Linear Cofactor Relationships in Boolean Functions

Jin S. Zhang[1] **Malgorzata Chrzanowska-Jeske**[1] **Alan Mishchenko**[2] **Jerry R. Burch**[3]

[1] Department of ECE, Portland State University, Portland, OR

[2] Department of EECS, UC Berkeley, Berkeley, CA

[3] Synopsys Inc. Hillsboro, OR

Abstract *- This paper describes linear cofactor relationships (LCRs), which are defined as the exclusive-sums of cofactors with respect to a pair of variables in Boolean functions. These relationships subsume classical symmetries and single-variable symmetries. The paper proposes an efficient algorithm to detect LCRs and discusses their potential applications in Boolean matching, minimization of decision diagrams, synthesis of regular layout-friendly logic circuits, and detection of support-reducing boundsets.*

1 Introduction

Many properties of a Boolean function can be expressed using cofactors. The cofactors are derived from the function by substituting constant values for the input variables. For example, the Boolean difference can be expressed as $f_0 \oplus f_1$, where $f_0 = f\,[x \leftarrow 0]$ and $f_1 = f\,[x \leftarrow 1]$ are the negative and positive cofactors of function f with respect to (w.r.t.) variable x. The Boolean difference equals 0 if f does not depend on variable x.

A Boolean function has symmetry if the function stays unchanged when several of its input variables are permuted. When the permutation involves two variables, it is called *classical symmetry* [1]. We use f_{00}, f_{01}, f_{10} and f_{11} to represent the cofactors of f w.r.t. two variables, say x_i and x_j. Equation $f(\ldots x_i,\ldots,x_j\ldots) = f(\ldots x_j,\ldots,x_i,\ldots)$ can be expressed as $f_{01} = f_{10}$ or equivalently, as the linear cofactor relationship $f_{01} \oplus f_{10} = 0$.

Many applications in electronic design automation exploit symmetries of functions to achieve better results and improve performance. Classical symmetries have a long history and many applications, such as functional decomposition in technology independent logic synthesis [1-4], technology mapping [5-7] and BDD minimization [8].

Single variable symmetries [1] are derived based on similar cofactor relationships between the other cofactor pairs: $f_{00} \oplus f_{01} = 0/1$, $f_{00} \oplus f_{10} = 0/1$, $f_{01} \oplus f_{11} = 0/1$ and $f_{10} \oplus f_{11} = 0/1$. These symmetries imply that function f remains unchanged or complemented under the constant assignment of one variable while the other variable is complemented, for example, $f(\ldots 0,\ldots x_j\ldots) = f(\ldots 0,\ldots x_j'\ldots)$.

Both classical and single variable symmetries are called first-order symmetries. The notion of symmetries can be further extended [9] to include symmetries defined as invariants of the functions under the permutation/complementation of arbitrary groups of variables (rather than variable pairs). These symmetries are called *higher-order symmetries*.

One of the benefits of classical symmetries is that they result in functionally equivalent cofactors, which lead to merging of nodes in Binary Decision Diagrams (BDD) [10], thereby reducing the BDD size. Our initial motivation for extending the notion of classical symmetries is that some functions can be more efficiently represented by Functional Decision Diagrams (FDD) [11, 12] and Kronecker Functional Decision Diagrams (KFDD) [13]. If we can detect similar relationships in Boolean functions, which result in isomorphic nodes in FDD and KFDD, we can minimize these decision diagrams.

In this paper, we study linear (EXOR-based) relationships among any non-empty subset of the four two-variable cofactors of a Boolean function. These linear cofactor relationships represent sufficient conditions for the minimization of all of the above mentioned decision diagrams (DD).

We show that LCRs can be computed as efficiently as the classical symmetries. However, they are more diverse and, therefore, give a more detailed characterization of Boolean functions. This is demonstrated, for example, in Section 5.1, by the comparison of classical symmetries and LCRs when applied to Boolean matching. Higher-order symmetries are an orthogonal generalization of classical symmetries in relation to LCRs because neither of them subsumes the other. However, the computation of higher-order symmetries is substantially harder because it requires manipulation of the matrices of cofactors of the Boolean function, which is more complex than the BDD traversal procedures proposed in this paper for the computation of LCRs.

The remainder of this paper is organized as follows. Background information on Boolean functions, decision diagrams, classical and single variable symmetries are introduced in Section 2. Section 3 presents the motivation, definitions and properties of LCRs. An efficient algorithm to detect LCRs is described in Section 4. Experimental results on MCNC benchmarks demonstrate a performance improvement of this algorithm over the naïve method. Section 5 discusses potential applications of LCRs in Boolean matching, PKFDD minimization, regular layout synthesis [14], and detecting support-reducing boundsets [15]. The conclusions are drawn in Section 6.

2 Background

2. 1 Boolean Functions and Decision Diagrams

We use $f(x_1,\ldots,x_n)$ to denote a Boolean function with input variables $x_1,\ldots,x_n$. In this work, we only consider completely specified Boolean functions and Boolean variables.

The variables that a function f depends on are called the *support* of f, denoted by $supp(f)$.

We write $f\,[x\leftarrow 0]$ to denote the function formed from f by replacing x with 0. This is also known as the *negative cofactor* of f w.r.t. variable x, denoted by $f_{x'}$. While x is not in the support of the resulting function, it is convenient to treat x as an input. The *positive cofactor*, f_x, is $f\,[x\leftarrow 1]$.

We denote multiple substitutions similarly. For example, $f\,[x_i \leftarrow 0,\ x_j\leftarrow 0]$ is a *cofactor* of a function $f\,(x_1,\ldots,x_n)$ w.r.t. variables x_i and x_j. If x_i and x_j are clear from context, then this cofactor can be denoted by f_{00}.

The Shannon expansion of a Boolean function f is:

$$f = x \bullet f_x + x' \bullet f_{x'} \qquad (1)$$

where "•" denotes conjunction and "+" denotes disjunction. If no confusion results, we may leave the conjunction operator implicit.

The positive Davio expansion [16] (denoted Davio I or pD) for a Boolean function f is:

$$f = f_{x'} \oplus (x \bullet (f_x \oplus f_{x'})) \qquad (2)$$

where "⊕" denotes the EXOR operation. $(f_x \oplus f_{x'})$ is the Boolean difference of f w.r.t. variable x. We refer to $f_{x'}$ as the constant moment and $(f_x \oplus f_{x'})$ as the linear moment. The negative Davio expansion (denoted Davio II or nD) is:

$$f = f_x \oplus (x' \bullet (f_x \oplus f_{x'})) \qquad (3)$$

Here f_x is the constant moment and $(f_x \oplus f_{x'})$ is the linear moment.

A binary decision diagram (BDD) for a Boolean function f is generated by successively applying Shannon expansions to all variables. A reduced ordered BDD (ROBDD) is a BDD with the constraint that input variables appear in the same order in every path from root to leaves, and each node represents a distinct function. Figure 1 (a) show the ROBDD for function $f = abc + abd + acd + bcd$. The dashed lines indicate the negative cofactor of each node.

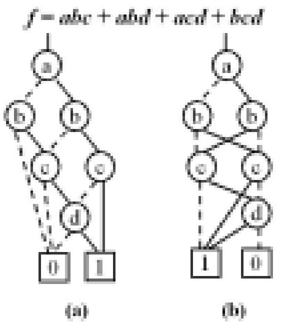


Figure 1. (a) ROBDD (b) FDD with nD expansions

A functional decision diagram (FDD) [11, 12] is generated by successively applying either positive or negative Davio expansions to all the variables, with one type of expansion per variable. Figure 1 (b) is the FDD for *f* with all negative Davio expansions. The dashed lines represent the constant moments for each node.

A Kronecker Function Decision Diagram (KFDD) [13] allows both Shannon and Davio expansions in generating the decision diagram, although only one type of expansion is allowed per variable. Figure 2 shows KFDD for the same function *f* and the associated functions at each node.

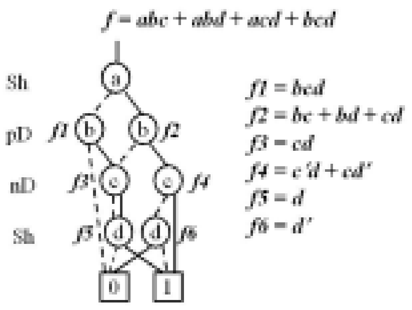


Figure 2. KFDD for *f*

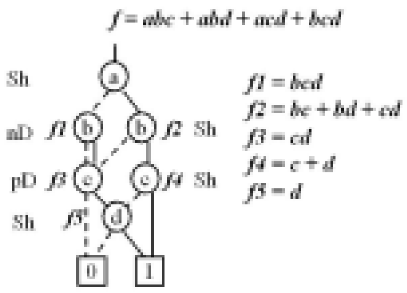


Figure 3. PKFDD for *f*

A Pseudo Kronecker Function Decision Diagram (PKFDD) [17] removes the constraint that only one type of expansion is allowed per variable for KFDD, thus providing more flexibility in representing a Boolean function. Figure 3 is the PKFDD for function *f* with the expansion type and the associated function at each node.

2.2. Classical and Single Variable Symmetry

The most basic notion of symmetry states that two variables, x_i and x_j, of function $f(..., x_i, ..., x_j, ...)$ are symmetric if the function remains invariant when the variables are swapped, i.e. $f(..., x_i, ..., x_j, ...) = f(..., x_j, ..., x_i, ...)$.

For a pair of variables x_i and x_j, there are four cofactors: f_{00}, f_{01}, f_{10}, and f_{11}. The above notion of symmetry can be expressed as $f_{01} = f_{10}$, or equivalently $f_{01} \oplus f_{10} = 0$. This symmetry is also called the *non-skew non-equivalent symmetry*, denoted by $x_i\mathrm{NE}x_j$. "Non-skew" refers to the right side of the equation being 0 rather than 1. "Non-equivalent" in this context means that, in each cofactor, the values of the two variables are not equivalent. *Non-skew equivalent symmetry*, denoted by $x_i\mathrm{E}x_j$, exists when $f_{00} \oplus f_{11} = 0$. This is a generalization where if the two variables are swapped, they must also be negated for the function to be preserved. *Skew symmetry* exists when two cofactors are complemented rather than equivalent to each other. For example, a Boolean function with *skew equivalent symmetry* is such that $f_{00} \oplus f_{11} = 1$. Skew symmetries are analogous to non-skew symmetries, except that the function is complemented, rather than preserved, when the variables are swapped. Taken together, all of the above symmetries are called *classical* symmetries [1].

For any two variables in a Boolean function, there are 6 possible cofactor pairs. *Single variable symmetries* [1] are defined when the function has equivalent (non-skew symmetry) or complement (skew symmetry) relationships of the other four cofactor pairs, not included in classical symmetries, for example, $f_{00} = f_{01}$. This non-skew relationship means that the negative cofactor of f w.r.t variable x_i does not depend on x_j. Table 1 summarizes classical and single variable symmetries, their functional invariance, cofactor relationships, names and symbol representations.

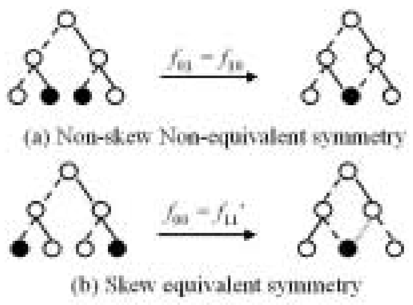
Table 1. Summary of Classical and Single Variable Symmetries.

Invariant of the function (non-skew/skew)	**Cofactor relationship**	**Name**	**Symbol**
$f(..., x_i, ..., x_j, ...) \oplus f(..., x_j, ..., x_i, ...) = 0/1$	$f_{10} \oplus f_{01} = 0/1$	Classical Symmetries	T_1 (NE)
$f(..., x_i, ..., x_j, ...) \oplus f(..., x_j', ..., x_i', ...) = 0/1$	$f_{00} \oplus f_{11} = 0/1$		T_2 (E)
$f(..., 0, ..., x_j, ...) \oplus f(..., 0, ..., x_j', ...) = 0/1$	$f_{00} \oplus f_{01} = 0/1$	Single Variable Symmetries	T_3
$f(..., 1, ..., x_j, ...) \oplus f(..., 1, ..., x_j', ...) = 0/1$	$f_{10} \oplus f_{11} = 0/1$		T_4
$f(..., x_i, ..., 0, ...) \oplus f(..., x_i', ..., 0, ...) = 0/1$	$f_{00} \oplus f_{10} = 0/1$		T_5
$f(..., x_i, ..., 1, ...) \oplus f(..., x_i', ..., 1, ...) = 0/1$	$f_{01} \oplus f_{11} = 0/1$		T_6

3. Linear Cofactor Relationships in Boolean Functions

3.1. Motivations

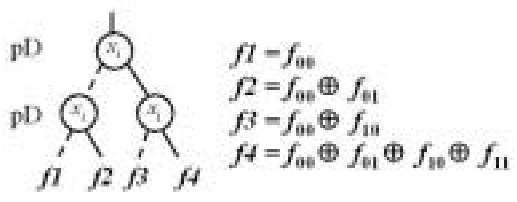
The existence of both classical symmetries and single-variable symmetries in a Boolean function results in shared or constant nodes in the corresponding ROBDD, as illustrated in Figure 4. The complemented edges represented by dotted lines are used in the decision diagram in the case of skew symmetries. This observation allows us to reduce the size of the ROBDD.



$f_{01} = f_{10}$
(a) Non-skew Non-equivalent symmetry
$f_{00} = f_{11}'$
(b) Skew equivalent symmetry

Figure 4. The effect of symmetries on ROBDD.

Boolean functions can be represented by FDDs using positive or negative Davio expansions. Figure 5 shows a fragment of an FDD with positive Davio expansions for both variables x_i and x_j. The functions at each node *f1, f2, f3, f4* are given in terms of the cofactors for variables x_i and x_j.



pD
pD
x_i
x_j
x_j
$f1$ $f2$ $f3$ $f4$
$f1 = f_{00}$
$f2 = f_{00} \oplus f_{01}$
$f3 = f_{00} \oplus f_{10}$
$f4 = f_{00} \oplus f_{01} \oplus f_{10} \oplus f_{11}$

Figure 5. Partial FDD with Positive Davio Expansions.

If variables x_i and x_j are symmetric with non-skew non-equivalent symmetry, then $f_{01} = f_{10}$. This relationship implies that *f2* = *f3*, so nodes *f2* and *f3* can be shared in Figure 5. Both classical non-skew non-equivalent and non-skew equivalent symmetries create shared nodes in the FDD.

Other node sharing possibilities lead to an additional set of cofactor relationships. For example, *f1* = *f4* if and only if (iff) $f_{00} = f_{00} \oplus f_{01} \oplus f_{10} \oplus f_{11}$, or equivalently, $f_{01} \oplus f_{10} \oplus f_{11} = 0$. This cofactor relationship involves three cofactors. Similarly, *f1* = *f2* iff $f_{00} = f_{00} \oplus f_{01}$, or $f_{01} = 0$. This is the constant cofactor relationship, which also reduces FDDs.

Since an FDD can be constructed with either positive or negative Davio expansions, there are four combinations of expansion types for two variables: pDpD, nDnD, pDnD, nDpD. Performing the same analysis above, we get the following eight cofactor relationships involving one or three cofactors [18,19]. Table 2 shows these relationships and the expansion combinations and equivalent cofactors that lead to these relationships (there could be multiple expansions that lead to the same relationship, only one is given in Table 2).

Table 2. Cofactor relationships involving 1 or 3 cofactors

Index	Cofactor relationship	Expansions	Equivalent cofactors
1	$f_{00} = 0$	pDnD	*f1* = *f2*
2	$f_{01} = 0$	pDpD	*f1* = *f2*
3	$f_{10} = 0$	pDpD	*f1* = *f3*
4	$f_{11} = 0$	pDnD	*f2* = *f4*
5	$f_{01} \oplus f_{10} \oplus f_{11} = 0$	pDpD	*f1* = *f4*
6	$f_{00} \oplus f_{10} \oplus f_{11} = 0$	pDnD	*f2* = *f3*
7	$f_{00} \oplus f_{01} \oplus f_{11} = 0$	nDpD	*f2* = *f3*
8	$f_{00} \oplus f_{10} \oplus f_{01} = 0$	nDnD	*f1* = *f4*

KFDDs allow Shannon, positive Davio, and negative Davio expansions to be used. Figure 6 shows the partial KFDD for variable x_i and x_j using Shannon and positive Davio expansions. The functions at each node *f1, f2, f3, f4* are given in terms of the cofactors for x_i and x_j.

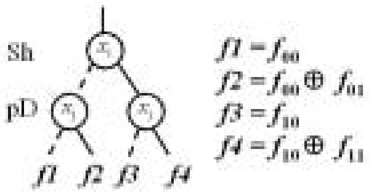


Figure 6. Partial KFDD with Shannon and positive Davio Expansions

It is easy to see that the cofactor relationships listed in Table 2 also reduce KFDDs. One condition that is missing is when *f*2 = *f*4, or equivalently $f_{00} \oplus f_{01} \oplus f_{10} \oplus f_{11} = 0$. This is a cofactor relationship that involves all four cofactors [18,19]. The other combinations of Shannon, as well as the positive and negative Davio expansions, lead to the same relationships.

Figures 7 and 8 show how FDDs and KFDDs can be reduced as a result of these cofactor relationships. In Figure 7, variables *a* and *b* in $f = a'c + a'b'd + ab'cd + abcd'$ do not have classical symmetries, therefore there is no node sharing in the

BDD for variables *a* and *b*. However, there is node sharing in the FDD because $f_{01} \oplus f_{10} \oplus f_{11} = 0$. Using this relationship in *f* between variables *a* and *b*, the FDD for *f* can be constructed with a specific variable order and expansion types to take advantage of this node reduction. Similarly, Figure 8 shows that variables *a* and *b* are not symmetric in function $f = a'c + a'bd' + acd + abc'd'$ with classical symmetries, therefore there is no node sharing in the BDD. However, $f_{00} \oplus f_{01} \oplus f_{10} \oplus f_{11} = 0$. This results in a shared node in the KFDD if variables *a* and *b* are adjacent in the variable order, and Shannon and Positive Davio expansions are used.

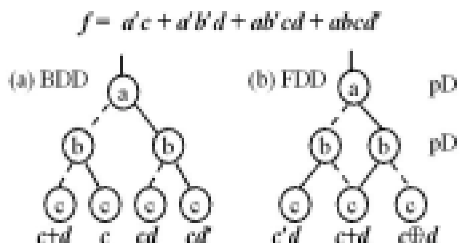


Figure 7. (a) Partial BDD (b) Partial FDD of function *f*

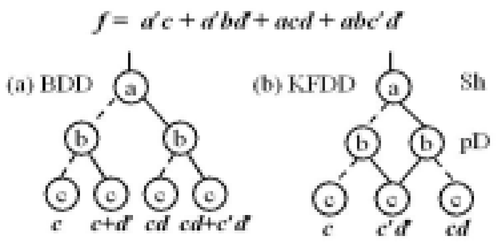


Figure 8. (a)Partial BDD (b) Partial KFDD of function *f*

3.2 Linear Cofactor Relationships

Definition 1. Let $f(x_1 \ldots x_n)$ be a Boolean function. Let *i* and *j* be integers, $0 \le i \le n$, $0 \le j \le n$. Let $g = \langle g_4, g_3, g_2, g_1, g_0 \rangle$ be a Boolean vector of length 5, such that at least one of g_3, g_2, g_1, g_0 is 1. Variables x_i and x_j have *linear cofactor relationship g* in function *f* iff:

$$\forall x_1 \ldots x_n \, [g_4 = g_3 f_{11} \oplus g_2 f_{10} \oplus g_1 f_{01} \oplus g_0 f_{00}].$$

We use symbol $LCR_g(f, (x_i, x_j))$ to denote linear cofactor relationships. If *f* and (x_i, x_j) are clear from context, we write LCR_g .

For example, if variables x_i and x_j have non-skew non-equivalent classical symmetry in function *f*, then $0 = 0 \cdot f_{11} \oplus 1 \cdot f_{10} \oplus 1 \cdot f_{01} \oplus 0 \cdot f_{00}$. Therefore $LCR_{\langle 0, 0, 1, 1, 0 \rangle}$, or in hexadecimal form LCR_{06}, holds. All classical and single variable symmetries are subsumed by linear cofactor relationships.

We use a don't care "-" in *g* to represent multiple LCRs. For example, $LCR_{\langle 0, 0, 0, 1, - \rangle}$ represents both $LCR_{\langle 0, 0, 0, 1, 0 \rangle}$ and $LCR_{\langle 0, 0, 0, 1, 1 \rangle}$.

Definition 2. A *linear cofactor class* contains all the LCRs that have the same sum of g_0, g_1, g_2, g_3. We use LCC_n to denote the linear cofactor class, where $n = g_0 + g_1 + g_2 + g_3$.

For example, both $LCR_{\langle 0, 0, 1, 1, 0 \rangle}$ and $LCR_{\langle 1, 1, 0, 0, 1 \rangle}$ belong to LCC_2, even though they have different skew type. $LCR_{\langle 0, 1, 0, 0, 0 \rangle}$ and $LCR_{\langle 0, 1, 1, 1, 0 \rangle}$ belong to LCC_1 and LCC_3 respectively.

Table 3 summarizes all the cofactor relationships discussed so far. We adopt the skew terminology when the EXOR of the cofactors is 1.

Table 3. Summary of Discussed Cofactor Relationships

index	**Linear Cofactor Relationships (non-skew / skew)**	**Linear Cofactor Class**	**Symbol (non-skew/skew)**
1	$f_{00} = 0/1$	LCC_1	LCR_{01} / LCR_{11}
2	$f_{01} = 0/1$		LCR_{02} / LCR_{12}

3	$f_{10} = 0/1$		LCR_{04} / LCR_{14}
4	$f_{11} = 0/1$		LCR_{08} / LCR_{18}
5	$f_{01} \oplus f_{10} = 0/1$	LCC_2	LCR_{06} / LCR_{16}
6	$f_{00} \oplus f_{11} = 0/1$		LCR_{09} / LCR_{19}
7	$f_{00} \oplus f_{01} = 0/1$		LCR_{03} / LCR_{13}
8	$f_{10} \oplus f_{11} = 0/1$		LCR_{0C} / LCR_{1C}
9	$f_{00} \oplus f_{10} = 0/1$		LCR_{05} / LCR_{15}
10	$f_{01} \oplus f_{11} = 0/1$		LCR_{0A} / LCR_{1A}
11	$f_{00} \oplus f_{01} \oplus f_{10} = 0/1$	LCC_3	LCR_{07} / LCR_{17}
12	$f_{00} \oplus f_{01} \oplus f_{11} = 0/1$		LCR_{0B} / LCR_{1B}
13	$f_{00} \oplus f_{10} \oplus f_{11} = 0/1$		LCR_{0D} / LCR_{1D}
14	$f_{01} \oplus f_{10} \oplus f_{11} = 0/1$		LCR_{0E} / LCR_{1E}
15	$f_{00} \oplus f_{01} \oplus f_{01} \oplus f_{11} = 0/1$	LCC_4	LCR_{0F} / LCR_{1F}

3.3 Statistics for linear cofactor relationships in MCNC benchmarks

Classical symmetries and single variable symmetries are common in Boolean functions. It is interesting to see how common the other linear cofactor relationships are to decide if it is worthwhile to detect these relationships in Boolean functions.

Table 4 gives the total number and ratio of non-skew and skew LCRs in each of the linear cofactor classes, for all MCNC multilevel combinational benchmark functions. The complete table can be found in [48], which shows that linear cofactor relationships exist in all MCNC benchmarks. It is obvious that more than 90% of the LCRs are non-skew and among all LCRs, 32.6% are classical and single variable symmetries.

Table 4. Statistics of linear cofactor relationships in MCNC Benchmark Functions

	LCC_1		**LCC_2**		**LCC_3**		**LCC_4**	
	Non-skew	Skew	Non-skew	Skew	Non-skew	Skew	Non-skew	Skew
Total	**30,610**	**11,005**	**114,305**	**8,652**	**85,927**	**2,063**	**124,681**	**540**
Ratio	**8.1%**	**2.9%**	**30.3%**	**2.3%**	**22.8%**	**0.6%**	**33%**	**0.1%**

Existing electronic design automation algorithms often make use of classical symmetries. It is possible that other linear cofactor relationships can be used to improve these algorithms. We will touch upon some potential applications in section 5.

4 Fast Computation of Linear Cofactor Relationships

While many algorithms have been proposed to detect classical symmetries efficiently [20-23], the only method to detect linear cofactor relationships had been the naïve method, which computes the four cofactors for each variable pair in the Boolean function and checks if the cofactors satisfy the relationships defined in Table 3. This method is straightforward, yet very inefficient for large circuits. In [21], an efficient method to compute classical symmetries was proposed, based on the theorem in [23]. Theorem 1 extends the scope in [23] to all linear cofactor relationships and thus allows the fast computation algorithm proposed in [21] be applied to the computation of all LCRs.

Theorem 1. Let $f(x_1,\ldots,x_n)$ be a Boolean function and let the variables x_i , x_j and x_k be variables in the support of f. x_i and x_j have LCR_g in f iff they have LCR_g in both cofactors of f w.r.t. x_k.

Proof: x_i and x_j have LCR_g in function f iff: $\forall x_1\ldots x_n\,[g_4 = g_3 f_{11} \oplus g_2 f_{10} \oplus g_1 f_{01} \oplus g_0 f_{00}]$. This equation holds true for all values of x_k, where $1 \le k \le n$, $k \neq i$ and $k \neq j$. That is:

$$\forall x_1\ldots x_{k-1}\, x_{k+1}\ldots x_n\,[g_4 = g_3 f_{11} \oplus g_2 f_{10} \oplus g_1 f_{01} \oplus g_0 f_{00}]\,[x_k \leftarrow 0] \text{ and}$$

$$\forall x_1\ldots x_{k-1}\, x_{k+1}\ldots x_n\,[g_4 = g_3 f_{11} \oplus g_2 f_{10} \oplus g_1 f_{01} \oplus g_0 f_{00}]\,[x_k \leftarrow 1].$$

Taking $[x_k \leftarrow 1]$ and $[x_k \leftarrow 1]$ into the equations, we have:

$$\forall x_1\ldots x_n\,[g_4 = g_3 f_{11}[x_k \leftarrow 0] \oplus g_2 f_{10}[x_k \leftarrow 0] \oplus g_1 f_{01}[x_k \leftarrow 0] \oplus g_0 f_{00}[x_k \leftarrow 0]\,], \text{ and}$$

$$\forall x_1\ldots x_n\,[g_4 = g_3 f_{11}[x_k \leftarrow 1] \oplus g_2 f_{10}[x_k \leftarrow 1] \oplus g_1 f_{01}[x_k \leftarrow 1] \oplus g_0 f_{00}[x_k \leftarrow 1]\,].$$

Therefore, x_i and x_j have the same LCR_g in both cofactors of f w.r.t. x_k. The reverse implication holds because x_k must be 0 or 1.

□

4. 1 Computational Core

Theorem 1 states that we can compute the LCRs of a function if we know the LCRs of the function's cofactors w.r.t. a variable. Therefore, we can recursively solve the sub-problems and derive the final solution from the partial solutions. There are two recursive procedures in the proposed algorithm: ComputeLCR and LCRVars. ComputeLCR detects variable pairs with linear cofactor relationships in f_x and $f_{x'}$, where x is the top variable in the BDD. LCRVars detect LCRs between variable x and the other variables in the support of f.

We use an *LCR graph* to represent the computed LCRs. The vertices of the LCR graph correspond to variables in the support of the function. The edges correspond to variable pairs with LCR. The graph operations *union* ($\cup$) and *intersection* ($\cap$) are similarly defined as the set *union* and *intersection* on the sets of edges. The *Cartesian product* ($\times$) of variable x by a set of variables Y results in a graph composed of edges connecting the vertex of variable x with the vertices of variables in Y. In the software implementation, the LCR graph is represented by Zero-Suppressed Binary Decision Diagrams (ZDD) [24], which give a canonical representation of the LCR information.

4.1.1 Top Level Recursion: ComputeLCR

Figure 9 shows the flow chart of ComputeLCR. It takes as inputs: function F, a set of variables V, and integer g. F is the function whose LCRs are being computed. V is initialized to be the support of F at the top level, but could become a super set of F's support in the subsequent recursive calls. g is an integer indicating the LCR to be computed, as given in Table 2. The program returns the LCR graph representing variable pairs with the particular LCR. It is worth noting that only variables that function F depends on participate in this computation. In the case of multi-output functions, the LCRs for each output are computed separately using the true support of the output function.

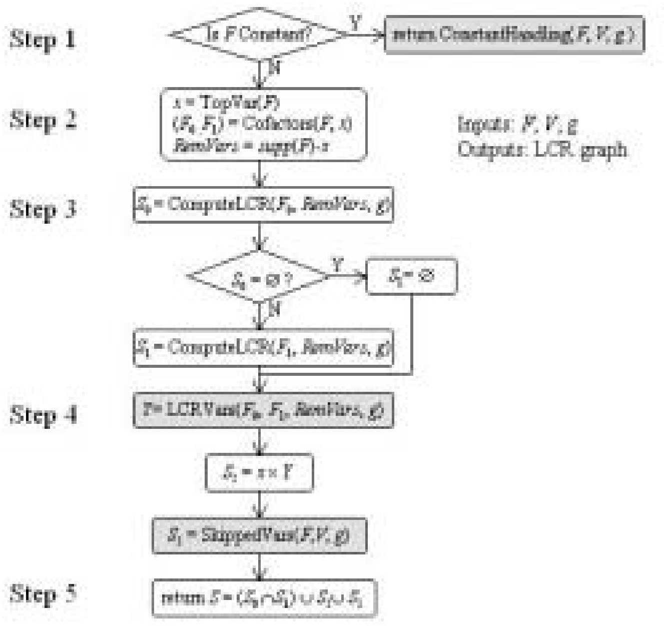

Step 1
Is F Constant?
Y
N
return ConstantHandling(F, V, g)
Step 2
x = TopVar(F)
(F0, F1) = Cofactors(F, x)
RemVars = supp(F)-x
Inputs: F, V, g
Outputs: LCR graph
Step 3
S0 = ComputeLCR(F0, RemVars, g)
S0 = ∅ ?
Y
N
S1 = ∅
S1 = ComputeLCR(F1, RemVars, g)
Step 4
Y = LCRVars(F0, F1, RemVars, g)
S2 = x × Y
S3 = SkippedVars(F, V, g)
Step 5
return S = (S0 ∩ S1) ∪ S2 ∪ S3

Figure 9. The flow chart of the top level recursion: ComputeLCR

The recursive steps are the same for each LCR. However the handling of Step 1 and the S_3 computation in Step 4 is different (indicated by shaded boxes in the flow chart) due to the difference in the cofactor relationships. The following paragraphs explain each step in the pseudo code. A brief discussion of classical symmetries is included here for completeness; the details can be found in [21].

Step 1. If F is a constant, then $F = F_{11} = F_{10} = F_{01} = F_{00}$. Checking if any variable pairs in V having linear cofactor relationships is equivalent to checking if there are any values of g that satisfy the following equation: $g_4 = (g_3 \oplus g_2 \oplus g_1 \oplus g_0) \bullet F$.

If F is constant 0, then each pair of variables in V has all 15 types of non-skew LCRs.

If F is constant 1, then each pair of variables in V has non-skew LCC_2, non-skew LCC_4, skew LCC_1 and skew LCC_3 LCRs.

When F is a constant, the program either returns a complete LCR graph, or an empty set, depending on the value of the function and the LCR to be computed.

Step 2. Compute the support of F and the cofactors of F w.r.t. the top variable x of the BDD. This results in two different Boolean functions, F_0 and F_1.

Step 3. The recursive calls are performed in this step. First, ComputeLCR is called with F_0 and $supp(F) - x$. Next the same procedure is repeated for the positive cofactor. If there is no LCR in the negative cofactor F_0, there is no need for the second recursive call, according to Theorem 1. As a result, the computation is very efficient in this case.

Step 4. The first part of the solution comes from the intersection of partial solution S_0 and S_1, because, according to Theorem 1, only the LCRs of both cofactors are LCRs of the function.

The set Y contains those variables y such that variables x (chosen in step 2) and y have LCR_g. This set is computed using another recursive procedure LCRVars, which will be discussed later.

The set S_3 contains LCRs involving variables that are in V but not in the support of F. This is relevant when a cofactor of the function does not depend on all the variables in the initial support of F less x. Figure 10 is a fragment of the BDD illustrating this scenario. The negative cofactor of F w.r.t. variable a only depends on variables d and e. Therefore, in the recursive procedure, variables *b* and *c* are skipped over. As a result, we need to add the LCRs involving the "skipped" variables. It is easy to see that between the skipped variables *b* and *c*, we have $F_{a'b'c'} = F_{a'b'c} = F_{a'bc'} = F_{a'bc}$, and between the skipped variable *b* and the variable *d* in the support of $F_{a'}$, we have $F_{a'b'd'} = F_{a'bd'}$ and $F_{a'b'd} = F_{a'bd}$.

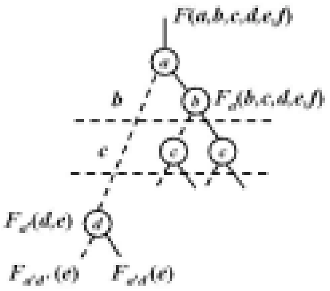


Figure 10. Illustrations of the skipped variables

- **LCRs among the skipped variables.** In this case: $F_{11} = F_{10} = F_{01} = F_{00}$. Checking if any skipped variable pairs have linear cofactor relationships is equivalent to checking if there are any values of *g* that satisfy the following equation: $g_4 = (g_3 \oplus g_2 \oplus g_1 \oplus g_0) \bullet F_{00}$. All non-skew LCC_2 and non-skew LCC_4 LCRs satisfy this equation.
- **LCRs between the skipped variables and the variables in the support of *F*.** In this case: $F_1 = F_0$, or equivalently, $F_{10} = F_{00}$ and $F_{11} = F_{01}$. Checking if any skipped variable having linear cofactor relationships to the variables in the support of *F* is equivalent to checking if there are any values of *g* that satisfy the following equation: $g_4 = (g_3 \oplus g_1) \bullet F_1 \oplus (g_2 \oplus g_0) \bullet F_0$. Different LCRs exist depending on the values and the relationships of F_0 and F_1. For example, if F_1 is complement to F_0, then $g_4 = (g_3 \oplus g_1) \oplus (g_3 \oplus g_2 \oplus g_1 \oplus g_0) \bullet F_0$. ⟨1, 1, 1, 0, 0⟩ and ⟨0, 1, 0, 1, 0⟩ are two of the LCRs that satisfy this equation. Table 5 gives the complete set of LCRs between the skipped variables and the variables in the support of *F* under all values and relationships of F_1 and F_0.

Table 5. Values & relationships of F_1 and F_0 with corresponding LCRs

F_1, F_0 values & relationships	Corresponding LCRs
$F_0 = 0$	All LCR_g, where g = ⟨0, 0, -, 0, -⟩ and ⟨0, 1, -, 1, -⟩.
$F_0 = 1$	All LCR_g, where g = ⟨0, 0, 1, 0, 1⟩, ⟨0, 1, 0, 1, 0⟩, ⟨0, 1, 1, 1, 1⟩, ⟨1, 0, 0, 0, 1⟩, ⟨1, 0, 1, 0, 0⟩, ⟨1, 1, 0, 1, 1⟩, ⟨1, 1, 1, 1, 0⟩.
$F_1 = 0$	All LCR_g, where g = ⟨0, -, 0, -, 0⟩ and ⟨0, -, 1, -, 1⟩.
$F_1 = 1$	All LCR_g, where g = ⟨0, 1, 0, 1, 0⟩, ⟨0, 0, 1, 0, 1⟩, ⟨0, 1, 1, 1, 1⟩, ⟨1, 0, 0, 1, 0⟩, ⟨1, 1, 0, 0, 0⟩, ⟨1, 0, 1, 1, 1⟩, ⟨1, 1, 1, 0, 1⟩.
$F_1 = F_0$	All non-skew LCC_2 and non-skew LCC_4 LCRs.
$F_1 = F_0'$	All LCR_g, where g = ⟨0, 0, 1, 0, 1⟩, ⟨0, 1, 0, 1, 0⟩, ⟨0, 1, 1, 1, 1⟩, ⟨1, 1, 0, 0, 1⟩, ⟨1, 1, 1, 0, 0⟩, ⟨1, 0, 1, 1, 0⟩, ⟨1, 0, 0, 1, 1⟩.

The complete solution $S = (S_0 \cap S_1) \cup S_2 \cup S_3$.

4.1.2 Inner recursion: LCRVars

Procedure LCRVars takes as inputs: F_0 and F_1, the two cofactors of F w.r.t. variable x, the set of candidate variables Y, and integer g. It returns the subset of Y such that x has LCRs with the variables in this subset. Figure 11 is the flowchart for this procedure. (Hereafter functions G and H are used to represent the cofactors F_0 and F_1 to avoid multiple subscripts.)

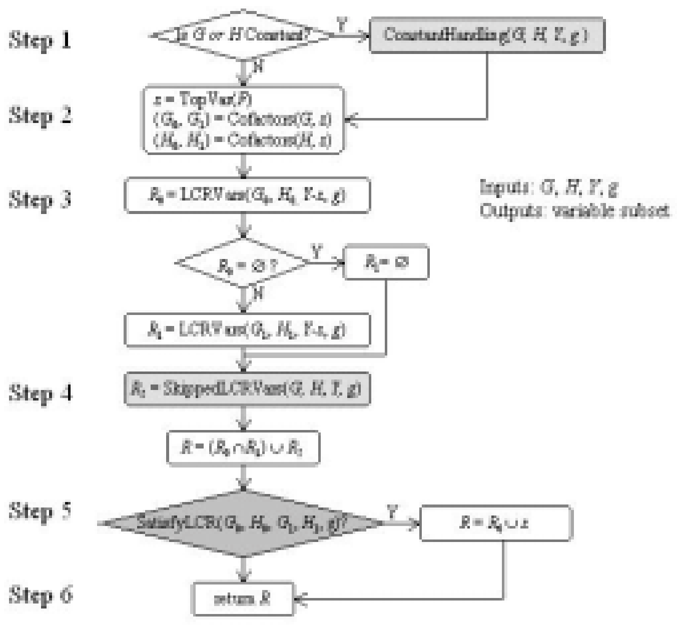

Step 1
Is G or H Constant?
Y
N
ConstantHandling(G, H, Y, g)
Step 2
x = TopVar(F)
(G0, G1) = Cofactors(G, x)
(H0, H1) = Cofactors(H, x)
Step 3
R0 = LCRVars(G0, H1, Y-x, g)
Inputs: G, H, Y, g
Outputs: variable subset
R0 = ∅ ?
R1 = ∅
R1 = LCRVars(G1, H0, Y-x, g)
Step 4
R2 = SkippedLCRVars(G, H, Y, g)
R = (R0 ∩ R1) ∪ R2
Step 5
SatisfyLCR(G0, H0, G1, H1, g)?
R = R0 ∪ x
Step 6
return R

Figure 11. The flowchart of inner recursion: LCRVars.

For each linear cofactor relationship, the program differs in step 1, and R_2 computation in step 4 and step 5 as indicated in the shaded boxes in the flowchart.

Step 1. Three scenarios are handled in this case: 1) G and H are both constant functions. 2) H is a constant function, but G is not. 3) G is a constant function, but H is not.

- ***G* and *H* are both constant functions.** In this case, $G = F_{01} = F_{00}$ and $H = F_{11} = F_{10}$. Checking if variable x has any linear cofactor relationships with the variables in Y is equivalent to checking if there are any values of g that satisfy the following equation: $g_4 = (g_3 \oplus g_2) \cdot H \oplus (g_1 \oplus g_0) \cdot G$. The existence of LCRs depends on the values of G and H. For example, if $H = 1$ and $G = 0$, we have $g_4 = (g_3 \oplus g_2)$. The following g satisfy this equation: $\langle 0, 0, 0, -, - \rangle$, $\langle 0, 1, 1, -, - \rangle$, $\langle 1, 0, 1, -, - \rangle$ and $\langle 1, 1, 0, -, - \rangle$. Therefore, there are 16 different LCRs under this value combination. Table 6 gives all fours combinations of H and G with the corresponding LCRs.

Table 6. Values of H/G and Corresponding Symmetries

H/G	Corresponding LCRs
00	All LCR$_g$: where $g = \langle 0, -, -, -, - \rangle$
01	All LCR$_g$, where $g = \langle 0, -, -, 0, 0 \rangle$, $\langle 0, -, -, 1, 1 \rangle$, $\langle 1, -, -, 0, 1 \rangle$, $\langle 1, -, -, 1, 0 \rangle$
10	All LCR$_g$, where $g = \langle 0, 0, 0, -, - \rangle$, $\langle 0, 1, 1, -, - \rangle$, $\langle 1, 0, 1, -, - \rangle$, $\langle 1, 1, 0, -, - \rangle$
11	All skew LCC$_1$ and LCC$_3$ LCR, non-skew LCC$_2$ and LCC$_4$ LCRs

- ***H* is a constant function, but *G* is not.** If H is constant 0, we have $g_4 = (g_3 \oplus g_2) \bullet 0 \oplus (g_1 \bullet F_{01} \oplus g_0 \bullet F_{00})$. There are three values of g that satisfy this equation: ⟨0, 0, 1, 0, 0⟩, ⟨0, 1, 0, 0, 0⟩ and ⟨0, 1, 1, 0, 0⟩. If H is constant 1, we have $g_4 = (g_3 \oplus g_2) \bullet 1 \oplus (g_1 \bullet F_{01} \oplus g_0 \bullet F_{00})$. ⟨0, 1, 1, 0, 0⟩, ⟨1, 0, 1, 0, 0⟩ and ⟨1, 1, 0, 0, 0⟩ satisfy this equation.
- ***G* is a constant function, but *H* is not.** If G constant 0, variable x has the following linear cofactor relationships with the variables in Y: ⟨0, 0, 0, 0, 1⟩, ⟨0, 0, 0, 1, 0⟩ and ⟨0, 0, 0, 1, 1⟩. If G constant 1, then we have ⟨0, 0, 0, 1, 1⟩, ⟨1, 0, 0, 0, 1⟩ and ⟨1, 0, 0, 1, 0⟩.

The variable subset Y is returned when both G and H are constants. In the last two scenarios, the program continues.

Step 2. A variable z in Y is selected and the functions are cofactored w.r.t. this variable.

Step 3. Based on Theorem 1, a variable belongs to the solution iff it belongs to the solutions of both cofactors. If one of the sub-problems returns an empty set, there is no need to solve the other one.

Step 4. Figure 12 is a fragment of a BDD illustrating the skipped variable situation. We want to check whether variable b has linear cofactor relationship with any variables in the support of $F_{a'}$. Since variable c is skipped over, we need to detect linear cofactor relationships between variables b and c. It is easy to see that $F_{a'b'c'} = F_{a'b'c}$ and $F_{a'bc'} = F_{a'bc}$.

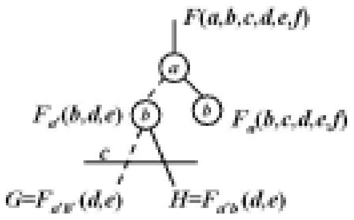


Figure 12. An illustration of skipped variables in *LCRVars.*

For skipped variables, $G = F_{01} = F_{00}$ and $H = F_{11} = F_{10}$, to check if variable x has any linear cofactor relationships to the skipped variables is equivalent to checking if there are any values of g that satisfy the following equation: $g_4 = (g_3 \oplus g_2) \bullet H \oplus (g_1 \oplus g_0) \bullet G$. This again, depends on the values and relationships of G and H, as shown in Table 7.

Table 7. Values & relationships of H and G with the corresponding LCRs

H, G values & relationships	Corresponding LCRs
$G = 0$	All LCR_g, where g = ⟨0, 0, 0, -, -⟩ and ⟨0, 1, 1, -, -⟩.
$G = 1$	All LCR_g, where g = ⟨1, 0, 0, 0, 1⟩, ⟨1, 0, 0, 1, 0⟩, ⟨0, 0, 0, 1, 1⟩, ⟨0, 1, 1, 0, 0⟩, ⟨1, 1, 1, 0, 1⟩, ⟨1, 1, 1, 1, 0⟩, ⟨0, 1, 1, 1, 1⟩.
$H = 0$	All LCR_g, where g = ⟨0, -, -, 0, 0⟩ and ⟨0, -, -, 1, 1⟩.
$H = 1$	All LCR_g, where g = ⟨1, 0, 1, 0, 0⟩, ⟨1, 1, 0, 0, 0⟩, ⟨0, 1, 1, 0, 0⟩, ⟨0, 0, 0, 1, 1⟩, ⟨1, 0, 1, 1, 1⟩, ⟨1, 1, 0, 1, 1⟩, ⟨0, 1, 1, 1, 1⟩.
$H = G$	All non-skew LCC_2 and LCC_4 LCRs.
$H = G'$	All LCR_g, where g = ⟨1, 0, 1, 0, 1⟩, ⟨1, 1, 0, 1, 0⟩, ⟨0, 1, 1, 1, 1⟩, ⟨1, 1, 0, 0, 1⟩, ⟨0, 1, 1, 0, 0⟩, ⟨1, 0, 1, 1, 0⟩, ⟨0, 0, 0, 1, 1⟩.

The resulting set is $R = (R_0 \cap R_1) \cup R_{2.}$

Step 5. This step checks to see if variable x has linear cofactor relationship to variable z, according to Table 3. If x has LCR to z, then z is added to the resulting set R.

The worst case complexity of the algorithm is cubic in the number of the BDD nodes, because the complexity of the procedure ComputeLCR is linear, while each call to LCRVars inside ComputeLCR has the worst case quadratic complexity. However, for the benchmark functions, the observed runtime is close to linear in the number of the BDD nodes.

4.2 Implementation of the Algorithm

The proposed algorithm was implemented in C with the CUDD decision diagram package [25] and the EXTRA library of DD procedures [26].

Binary decision diagrams (BDDs) are used to represent Boolean functions, while Zero-Suppressed Binary Decision Diagrams (ZDDs) are used to represent variable sets and LCR graphs. First, the shared BDD of a multi-output benchmark function is constructed. This BDD is not modified during the computation that follows. No additional BDD nodes are built, which makes the implementation very fast. Similar to other algorithms implemented using BDDs, partial results of computation are cached to prevent multiple calls with the same arguments. The calls to the caching procedures are omitted in the above pseudo-codes for simplicity. The cache lookups are performed before Step 2 and the cache insertions before returning the result in both ComputeLCR and LCRVars.

ZDDs provide a canonical representation of the LCR graph. The ZDDs are usually small, compared to the BDDs of the benchmark functions, because the LCR graphs are usually sparse. Even when a function has an LCR between each pairs of variables, the ZDD representation is still compact. In this case, the LCR graph is a clique, whose ZDD representation is quadratic in the number of variables.

Although the program doesn’t generate new BDD nodes, a small number of ZDD nodes are created to manipulate the LCR graph and the variables sets in the recursive procedures. However, experiments show that the increase in the number of ZDD nodes is still negligible, compared to the size of the shared BDDs of the original functions.

4.3 Experimental Results

The program was run on a 750MHz Pentium III PC under Red Hat Linux 7.3. We only compare our results with the naïve method because it is the only other method known to compute all LCRs.

In table 8, the MCNC benchmark function *cm151a* (12 inputs, 1 output) is used to demonstrate the efficiency of the program when there is no LCR in the circuit. All 15 non-skew LCRs are computed and reported individually using the naïve method and the algorithm proposed in the paper. The runtimes reported are accumulated over 100 runs to capture reliable data. They include only LCR computation time and do not include the time to read the benchmark file and construct the BDDs. No variable pairs in *cm151a* have LCR_{01}. Our algorithm is 10x faster than the naïve method in this case. For $LCR_{02,}$ which holds 11 variable pairs, the speedup is 3x. Similar phenomenon can also be observed for other linear cofactor classes.

Table 8. Benchmark results for cm151a

LCR	Num. of LCRs	New (sec)	Naïve (sec)	Performance gain
LCR_{01}	0	0.23	2.19	10
LCR_{02}	11	0.74	2.19	3

LCR_{04}	0	0.16	2.15	13
LCR_{08}	11	0.71	2.15	3
LCR_{06}	0	0.05	2.21	44
LCR_{09}	0	0.1	2.19	21
LCR_{03}	0	0.05	2.20	44
LCR_{0C}	0	0.03	2.14	71
LCR_{05}	24	0.55	2.17	4
LCR_{0A}	46	0.65	2.13	3
LCR_{07}	0	0.2	2.13	11
LCR_{0B}	0	0.1	2.11	21
LCR_{0D}	0	0.09	2.13	24
LCR_{0E}	0	0.12	2.14	18
LCR_{0F}	56	0.63	2.06	3

The larger the circuit, the greater the speedup. Table 9 gives the total LCRs and runtime information over a number of large MCNC benchmarks. The first three columns show the benchmark name, number of inputs and outputs, and the number of the BDD nodes after reading and reordering by the sifting algorithm [25]. The next column gives the total number of non-skew LCRs. Average runtime for both the fast and naïve methods are given in the two columns under "Performance". Each runtime is an average of 15 runs computing 15 different non-skew LCRs. Column "gain" records the performance speedup between the fast and naïve algorithm. Table 9 demonstrates a significant speedup of the fast algorithm over the naïve method for all benchmarks.

Table 9. Symmetry and performance data on MCNC Benchmark

Benchmark Statistics				**Total LCRs**	**Performance**		
Name	**Ins**	**Outs**	**|BDD|**		**Fast (s)**	**Naïve (s)**	**Gain**
alu4	14	8	1182	129	0.01	0.18	18
dalu	75	16	1407	11249	0.15	3.33	22.2
des	256	245	4291	15241	0.17	1.49	8.8
frg2	143	139	2089	19212	0.07	1.13	16.1
k2	45	45	1381	10104	0.14	1.63	11.6
pair	173	137	5487	22577	0.16	8.86	55.4
rot	135	107	6119	7988	0.19	20.45	107.6
too_large	38	3	815	738	0.02	0.88	44
C1355	41	32	29586	256	0.29	170.98	589.6
C1908	33	25	9519	3362	0.23	29.62	128.8
C2670	233	140	4488	15669	1.40	86.43	61.7
C3540	50	22	24504	6243	1.81	72.00	39.8
C432	36	7	1226	212	0.01	2.57	257

C499	41	21	28177	256	0.28	155.34	554.8
C5315	178	123	2903	51327	0.93	17.66	19
C7552	207	108	8439	53725	0.91	160.27	176.1

5 Applications of Linear Cofactor Relationships

Symmetry is an important characteristic of Boolean functions. Many applications in electronic design automation exploit classical symmetries to achieve better results and/or improve performance. Linear cofactor relationships, excluding classical symmetries and single variable symmetries, are relatively new. In this section, we touch upon some natural applications of linear cofactor relationships.

5.1 Boolean Matching

Boolean matching is a technique used to determine if a subfunction can be implemented with a specific cell from a given technology library. Two *n*-input Boolean functions *match* if one of them can be transformed into another by one or more of the following transformations: input permutation (P1), input negation (P2), output negation (P3). Functions are P-equivalent under P1, NP-equivalent under P1 and P2, and NPN-equivalent under all three transformations.

Classical symmetries have been used as signatures for Boolean matching [5,6,27,28]. The following theorems state the applicability of LCRs as signatures for Boolean matching (proofs are omitted due to space limitation).

Theorem 2. P1 and P2 transformations do not alter the total number of LCRs in each linear cofactor class.

Theorem 3. P3 transformation changes skew type for the LCRs in LCC_1 and LCC_3, and preserves the skew type for LCRs in LCC_2 and LCC_4. □

Theorems 2 and 3 state that the number of linear cofactor relationships in each linear cofactor class are preserved under NPN transformations, and therefore, can be used as signatures in Boolean matching. Table 10 compares the effectiveness of using all linear cofactor relationships vs. classical symmetries as signatures. The signature using linear cofactor relationships includes 8 integers, representing the total number of non-skew and skew LCRs in LCC_1, non-skew classical symmetries in LCC_2, skew classical symmetries in LCC_2, non-skew single variable symmetries in LCC_2, skew single variable symmetries in LCC_2, non-skew and skew LCRs in LCC_3, non-skew LCRs in LCC_4 and skew LCRs in LCC_4. Because both classical symmetries and single variable symmetries are closed under NPN transformations, we can separate them into different components of the signatures. Since output negation changes the skew type for LCC_1 and LCC_3, we use the sum of skew and non-skew LCRs in those cases. The signature using classical symmetries includes two integers, representing the total number of non-skew classical symmetries and the total number of skew classical symmetries. The column “Total” contains the total number of NPN equivalence classes for 2, 3, and 4 variables. The column “LCR” lists the number of distinct LCR signatures in these NPN equivalence classes, whereas the column “Classical” list the number of distinct classical signatures.

Table 10. Effectiveness of using LCRs as signatures

NPN Equivalence Class	Total	LCR	Classical
2-variable	4	3	3
3-variable	14	12	8
4-variable	222	172	20

The signatures fail to distinguish two 2-variable NPN equivalence classes $f = a'$ and $f = 1$ because the proposed computation algorithm only detects LCRs for variables in the support of the function. For these two functions, there is no variable pair in the support.

The above analysis shows that the distinguishing power of LCRs increases dramatically, compared to that of classical symmetries, as the number of variables in the NPN equivalence class increases. With the efficient computation algorithm proposed in this paper, LCRs can easily be incorporated as filters to quickly prune unnecessary tautology checks. Other techniques [5,6,27,28] can also be used in combination with LCRs to facilitate the task of Boolean Matching.

5.2 PKFDD Minimization

The size of decision diagrams is crucial in many CAD applications. For BDDs, the variable ordering is the only parameter available to reduce its size. Many algorithms have been proposed for BDD minimization [29,30]. Several algorithms exploit classical symmetries to create smaller BDDs [8, 22, 31].

PKFDD is the most general bit-level decision diagram for switching functions, since each node can be decomposed using either Shannon, positive Davio or negative Davio expansions. Canonicity is sacrificed in PKFDDs to achieve smaller decision diagrams.

Exact PKFDD minimization was discussed in [32]. Because both the variable order and the decomposition type affect the size of the decision diagram, it is hard to find a variable order and decomposition type at each node that result in the smallest PKFDD. Heuristic methods have been proposed in PKFDD minimization [33-35].

Linear cofactor relationships can be used to assist PKFDD minimization. While only classical and single variable symmetries help in reducing the size of BDDs, all LCRs can reduce the size of PKFDDs. Detecting these relationships can help decide both the ordering of the variables and the decomposition type to be applied in order to reduce the size of the decision diagram.

We use the following notation to represent the decomposition combination. The combination is denoted using 3 letters with S representing Shannon expansion, P representing positive Davio expansion and N representing negative Davio expansion. The order indicates the expansion type of the parent node, the left child, and the right child. For example, Figure 13 illustrates the expansion for function *f* with notation "SSP" as decomposition types.

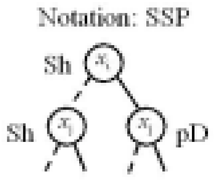


Figure 13. Illustration of PKFDD expansion notation.

In Table 11, we list all the non-skew and skew LCRs and the corresponding expansion combinations that have shared nodes due to these LCRs (skew LCRs result in complemented edges in the decision diagrams). In total, there are 27 different expansion combinations. Symbol "X" in the cell states that the LCR in the column results in a shared node for the expansion combination in the corresponding row.

Table 11. LCR types with corresponding expansions

Exp. Types	**Linear cofactor relationships**														
	LCR_{-1}	LCR_{-2}	LCR_{-4}	LCR_{-8}	LCR_{-6}	LCR_{-9}	LCR_{-3}	LCR_{-C}	LCR_{-5}	LCR_{-A}	LCR_{-7}	LCR_{-B}	LCR_{-D}	LCR_{-E}	LCR_{-F}

SSS					X	X	X	X	X	X					
SPS		X				X		X	X		X	X			
SNS	X				X			X		X	X	X			
SSP				X	X		X		X				X	X	
SSN			X			X	X			X			X	X	
SPP		X		X					X			X	X		X
SNN	X		X							X	X			X	X
SPN		X	X			X					X		X		X
SNP	X			X	X							X		X	X
PSS			X	X			X				X	X			X
PPS		X	X		X	X					X				X
PNS	X			X	X	X						X			X
PSP			X				X			X		X	X	X	
PSN				X			X		X		X		X	X	
PPP		X	X		X			X		X				X	
PNN	X			X		X		X	X				X		
PPN		X				X		X	X		X			X	
PNP	X				X			X		X		X	X		
NSS	X	X						X					X	X	X
NPS	X							X		X	X	X	X		
NNS		X						X	X		X	X		X	
NSP	X			X	X	X								X	X
NSN		X	X		X	X							X		X
NPP	X			X		X	X			X	X				
NNN		X	X		X		X		X			X			
NPN			X			X	X			X		X	X		
NNP				X	X		X		X		X			X	
SUM	10	10	10	10	12	12	10	10	10	10	12	12	12	12	10

Table 11 shows that, for each pair of variables with a linear cofactor relationship, there are many expansion combinations that could be assigned to create shared nodes. If the diagrams are mapped directly to circuit implementation, it is beneficial to choose the combination that uses more Shannon expansions to reduce the cost. Also, for each expansion combination, there are 6 different LCRs that reduce the size of PKFDD. Table 4 shows that linear cofactor relationships are abundant in benchmark functions. It is, therefore, promising to attempt PKFDD reduction taking advantage of these functional properties.

5.3 Regular Layout

Layout regularity is desirable because it offers predictability in circuit area and delay. It also significantly simplifies routing, reduces gate output load and improves testability. Regularity of layout is especially important when circuits are mapped into programmable or field programmable logic devices and gate arrays, since the majority of these devices have a large portion of their routing resources available as local and neighbor-to neighbor connections.

One approach to achieve layout regularity is to transform a Boolean function into a specialized type of decision diagram, which can be mapped directly into regular circuits composed of Shannon and Davio gates. Boolean functions are transformed into Pseudo-symmetric Binary Decision Diagram (PSBDD) through joint-vertices operations in [36-39]. This operation reintroduces the same variables at multiple levels, thereby increases the number of levels comparing to that of a ROBDD. The benefit is a regular symmetric array structure, which is usually triangular in shape. This technique was generalized to create Pseudo-symmetric Kronecker Functional Decision Diagram (PSKFDD) [40-42], which offers greater flexibility and increases the solution space. However the process of generating PSKFDD also creates repetition of some variables, resulting in circuits that are larger in the number of gates, though regular.

Detecting and utilizing linear cofactor relationships in Boolean functions can reduce the number of levels added to PSKFDDs as a result of repeating variables. First, we detect linear cofactor relationships between all pairs of variables in the circuit. Then we find the longest sequence of variables such that each pair of adjacent variables has a LCR. Next the expansions are assigned to each variable on the path in such a way that the corresponding reduction type (as shown in Table 10) allows for merging of sufficient nodes so that the planar layout is created on these levels [14]. This guarantees that for the variables included in the longest path, they will not be repeated in the decision diagrams. For functions with no LCRs at all, or if the longest path does not include all variables, the rest of the diagram is constructed using a heuristic algorithm, which could potentially introduce repetition of variables.

We compared results generated using LCRs with earlier works on PSBDDs and PSKFDDs to achieve regular layout. Figure 14 shows that the LCR approach results in fewer logic levels in PSKFDDs comparing with that of [42]. Figure 15 shows the level and node ratios of the LCR approach to that of [36]. The LCR approach consistently produces fewer levels on all 19 benchmarks, even though the number of nodes may be more in a few cases. These comparisons show the potential of using LCRs in achiever regular layout with less area. It is worth pointing out that while the resulting PSKFDDs contains less number of levels than that of PSBDDs, the logical complexity of each node is larger.

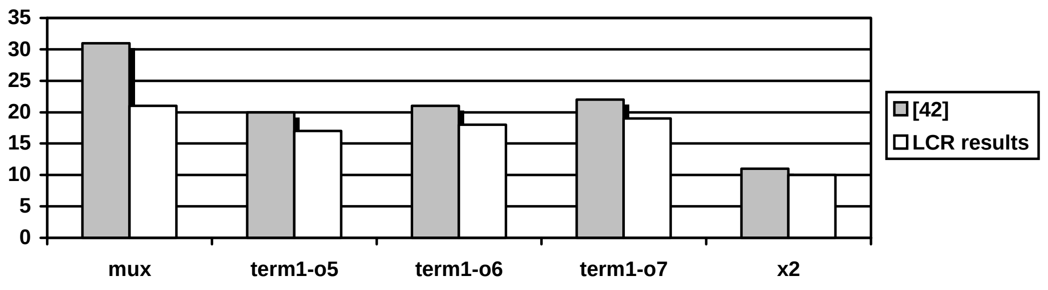


Figure 14. Comparison with [42] on the number of levels in the decision diagrams

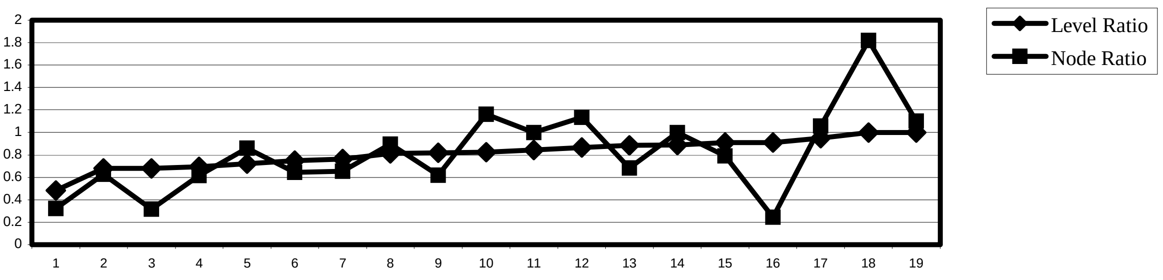


Figure 15. Level and node ratio: LCR results/[36]

5.4 Detecting Support-Reducing Bound sets

Detecting support-reducing bound sets is an important step in Boolean decomposition. It affects both the runtime and the quality of results of several applications in technology mapping and re-synthesis. Reference [43] proposed an efficient method to compute all support-reducing bound sets, that is, all groups of variables, which can lead to the decomposition resulting in the reduction of a function's support [49]. This method is quite efficient because it does not explicitly enumerate through all bound sets. Instead, it creates all bound sets implicitly and uses a cache to avoid repeated computations. Detailed profiling of the decomposition system has shown that the exhaustive bound sets detection is the most time-consuming task in the flow.

A bound set X_1 leads to an n-to-k *support-reducing* decomposition if k satisfies $[log_2\mu] \leq k < n$, where $n = |X_1|$ and μ is the number of distinct cofactors for variables in X_1 [44,45]. The presence of LCC_2 LCRs in a Boolean function results in shared nodes in the BDD for the function, thereby reducing the number of cofactors. Therefore we can use LCC_2 LCRs as heuristics to identify potential support-reducing bound set candidates [15].

Heuristic 1. We compute the set of variable pairs with two LCC_2 LCRs. These sets are 2-to-1 support-reducing bound sets. We then iteratively add another variable from the support of the function to form three-, four-, and five-variable support-reducing bound sets.

Heuristic 2. We compute the sets of three variables that contain one LCC_2 LCR in at least two of the variable pairs. This set is potentially 3-to-2 support-reducing. The four- and five-variable bound sets are formed by iteratively adding another variable from the support of the function to the existing set.

The experiments were conducted on a set of MCNC and ITC'99 benchmarks. The following notation is used in Table 12. Column "G/T" shows the ratio of true support-reducing bound sets found by the heuristics to the total number of support-reducing bound sets of a given size. Column "G/F" is the ratio of true support-reducing bound sets to the total number of candidates found by the heuristics. These two ratios characterize the efficiency of the heuristics from two different points of view. Column "Gain" represents the runtime improvement of the proposed method compared to the exhaustive method.

Table 12. Average G/T and G/F ratios and performance gain

Name	**3-var**		**4-var**		**5-var**		**Gain**
	G/T	G/F	**G/T**	G/F	**G/T**	G/F	
9symml	73%	88%	66%	95%	67%	99%	36
alu4	87%	99%	76%	99%	65%	89%	61
apex6	96%	84%	94%	92%	98%	95%	32
b15	99%	97%	98%	99%	98%	100%	38
b17	92%	91%	94%	96%	95%	99%	41
b20	96%	97%	90%	99%	91%	99%	41
c8	80%	99%	89%	100%	94%	100%	27
C2670	98%	61%	95%	80%	97%	95%	34
C3540	93%	94%	91%	96%	91%	99%	35
C5315	98%	73%	99%	81%	100%	93%	29
C7552	97%	85%	97%	89%	98%	96%	34
dalu	**60%**	**84%**	**61%**	**95%**	**65%**	**100%**	**147**

frg2	94%	97%	89%	97%	84%	99%	55
i10	98%	85%	96%	94%	98%	99%	34
k2	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**18**
pair	100%	60%	100%	71%	100%	84%	40
rot	80%	81%	83%	87%	88%	96%	37
Average	**90%**	**87%**	**89%**	**92%**	**90%**	**97%**	**43**

As shown in Table 12, the heuristics can detect about 90% of all support-reducing boundsets. The percentage of support-reducing bound sets detected is inversely proportional to the performance gain. Benchmark *dalu* has the worst "G/F" and "G/T" ratios among these benchmarks, but it has the highest performance gain of 147. On the other hand, the heuristics can detect all the support-reducing bound sets for *k2*, but the performance gain is only 18. Therefore, there is a tradeoff between the number of support-reducing bound sets detected vs. the performance improvement over the exhaustive method. Even though other heuristics can be employed to further improve the "G/F" and "G/T" ratios, we believe that achieving around 90% on G/F and G/T ratios and 40x performance gain is a good middle ground.

Experimental results show that the heuristic method is much faster than the exhaustive method [43], yet it finds most of the support-reducing bound sets of three, four, and five variables. The detected support-reducing bound sets typically result in simpler decomposition functions, compared to those that are not detected by the proposed method. As a result, the constructive decomposition, which constitutes an important step in technology mapping and re-synthesis, can be performed more efficiently.

6 Conclusions

This paper presents linear cofactor relationships (LCRs) defined for pairs of variables in a Boolean function. This notion subsumes classical and single variables symmetries. Experiments on MCNC benchmarks show that these relationships are common in Boolean functions.

An efficient algorithm is proposed to detect LCRs. The algorithm is characterized as follows:

- It works on the shared BDD of multi-output functions and computes the LCR information for each output.
- It exploits the compactness and canonicity of the ZDD representation to store the LCR information computed for a node in the shared BDD.
- It computes all 30 types of LCRs.
- It is particularly fast when applied to Boolean functions with no LCRs.

The experimental results show that the overall performance of the algorithm is significantly better than the naïve method.

The proposed efficient LCR detection enables the use of LCRs in CAD applications. Several such applications of LCRs are discussed in this paper: Boolean Matching, decision diagram minimization, synthesis of regular layout-friendly circuits, and detection of support-reducing bound sets in Boolean functions. We expect other applications of LCRs to emerge in VLSI IC design flow.

Acknowledgement: The authors wish to thank Prof. Robert Brayton for proposing the term "Linear Cofactor Relationship". The authors also acknowledge anonymous reviewers for their comments, which played an important role in shaping and clarifying the paper.

References:

[1] C. R. Edward and S. L. Hurst. A Digital Synthesis Procedure Under Function Symmetries and Mapping Methods. *IEEE Trans. On Computers*, vol. C-27, No.11, pp.985-997, November 1978.

[2] B.-G. Kim and D. L. Dietmeyer. Multilevel Logic Synthesis of Symmetric Switching Functions. *IEEE Trans. CAD*, 10(4), pp.436-446, April 1991.

[3] M. Chrzanowska-Jeske, W. Wang, J. Xia, and M. Jeske. Disjunctive Decomposition of Switching Functions Using Symmetry Information. *Proc. of IEEE SBCCI2000 International Symposium on Integrated Circuits and System Design*, pp. 67, September 2000.

[4] V.N. Kravets. Constructive Multi-level Synthesis by Way of Functional Properties. *Ph.D. Thesis.* University of Michigan, 2001.

[5] Y.-T. Lai, S. Sastry, and M. Pedram. Boolean Matching Using Binary Decision Diagrams with Applications to Logic Synthesis and Verification. P*roc. Intl. Conf. Computer Aided Design*, pp. 452-458, October 1992.

[6] F. Mailhot and G. De Micheli. Technology Mapping Using Boolean Matching and Don’t Care Sets. *Proc. European Design Automation Conference*, pp. 212-216, 1990.

[7] A. Mishchenko, X. Wang and T. Kam. A New Enhanced Constructive Decomposition and Mapping Algorithm *Proc. Design Automation Conference, 2003*, pp. 143-147, June 2003.

[8] Ch. Scholl, D. Möller, P. Molitor, and R. Drechsler. BDD Minimization Using Symmetries. *IEEE Trans. CAD*, 18(2) pp. 81-100, February 1999.

[9] V. N. Kravets and K. A. Sakallah. Generalized Symmetries in Boolean Functions. *Proc. Intl. Conf. Computer Aided Design*, pp. 526-532, November 2000.

[10] R. E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE. Trans. Comp.* Vol. C-35, No. 8, pp. 677-692, August 1986.

[11] U. Kebschull, E. Schubert, and W. Rosenstiel. Multilevel Logic Synthesis Based on Functional Decision Diagrams. *Proc. Euro-DAC*, pp. 43-47, 1992.

[12] U. Kebschull and W. Rosenstiel. Efficient Graph-based Computation and Manipulation of Functional Decision Diagrams. Proc. Euro-DAC, pp. 278-282, 1993.

[13] R. Drechsler, A. Sarabi, M. Theobald, B. Becker and M.A. Perkowski. Efficient Representation and Manipulation of Switching Functions Based on Ordered Kronecker Functional Decision Diagrams. *Proc. Design Automation Conference*, pp. 415-419, June 1994.

[14] M. Chrzanowska-Jeske, A. Mischenko, J. S. Zhang and M. Perkowski. Logic Synthesis for Layout Regularity using Decision Diagrams. *International Workshop on Logic Synthesis*, pp. 149-154, June 2004.

[15] J. S. Zhang, M. Chrzanowska-Jeske, A. Mischenko, and J. R. Burch. Detecting Support-reducing Bound Sets Using 2-Cofactor Symmetries. *Asia South Pacific Design Automation Conference*, January 2005.

[16] M. Davio, J.-P. Deschamps, and A. Thayse. Discrete and Switching Functions. *McGraw-Hill*, 1978.

[17] M.A. Perkowski. The Generalized Orthonormal Expansion of Functions with Multiple-valued Inputs and Some of Its Applications. *Intl. Symp. On Multi-Valued Logic*, pp. 442-450, 1992.

[18] M. Chrzanowska-Jeske. Generalized Symmetric and Generalized Pseudo-Symmetric Functions. *Proc. International Conference on Electronics, Circuits and Systems*, pp. 343-346, September 1999.

[19] M. Chrzanowska-Jeske. Generalized Symmetric Variables. *Proc. Intl. Conference on Electronics, Circuits, and Systems*. pp. 1147-1151, September 2001.

[20] D. Möller, J. Mohnke, and M. Weber. Detection of Symmetry of Boolean Functions Represented by ROBDDs. *Proc. Intl. Conf. Computer Aided Design*, pp. 680-684, November 1993.

[21] A. Mishchenko. Fast Computation of Symmetries in Boolean Functions. *IEEE Trans. CAD*, 22(11), pp.1588-1593, November 2003.

[22] S. Panda, F. Somenzi and B. F. Plessier Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams. *Proc. Intl. Conf. Computer Aided Design*, pp. 628-631, November 1994.

[23] C.-C. Tsai and M. Marek –Sadowaska. Generalized Reed-Muller Forms as a Tool to Detect Symmetries. *IEEE Trans. Computers,* C-45(1), pp. 33-40, January 1996.

[24] S. Minato. Zero-Suppressed BDDs for Set Manipulation in Combinational Problems. *Proc. Design Automation Conference*, pp. 272-277, 1993.

[25] F. Somenzi. *CUDD Package, Release 2.3.1*. http://vlsi.Colorado.EDU/~fabio/CUDD/cuddIntro.html

[26] A. Mishchenko. *EXTRA Library of DD procedures.* http://www.ee.pdx.edu/~alanmi/research/extra.htm

[27] C.-C. Tsai and M. Marek-Sadowska. Boolean Matching Using Generalized Reed-Muller Form. *Proc. Of Design Automation Conference*, pp. 339-344, June 1994.

[28] H. Savoj, M. J. Silva, R. K. Brayton and A. Sangiovanni-Vincentelli. Boolean Matching in Logic Synthesis. *Proc. European Design Automation Conference*, pp. 168-174, Feb. 1992.

[29] R. Drechsler, N. Drechsler, and W. Günther. Fast Exact Minimization of BDDs. *Proc. of Design Automation Conf.* pp. 200-205, June 1998.

[30] N. Ishiura, H. Sawada, and S. Yajima. Minimization of Binary Decision Diagrams Based on Exchange of Variables. *Proc. of Intl. Conf.* on CAD, pp. 472-475, 1991.

[31] D. Möller, P. Molitor, and R. Drechsler. Symmetry Based Variable Ordering for ROBDDs. *IFIP Workshop on Logic and Architecture Synthesis*, pp. 47-53, 1994.

[32] T. Sasao. Logic Synthesis and Optimization. *Kluwer Academic Publisher*, 1993.

[33] R. Drechsler and B. Becker. Dynamic Minimization of OKFDDs. *Proc. of Intl. Conf. on Computer Design*, pp. 602-607, October 1995.

[34] R. Drechsler, B. Becker and N. Göckel. Minimization of OKFDDs by Genetic Algorithms. *Intl. Symposium on Soft computing*, pp. B: 528-B:263, 1996.

[35] P. Lindgren, R. Drechsler and B. Becker. Improved Minimization Methods of Pseudo Kronecker Expressions. *Proc. Intl. Symposium on Circuit and Systems*, pp. VI:187-VI:190, 1998.

[36] M. Chrzanowska-Jeske, Y. Xu, and M. Perkowski. Logic Synthesis for a Regular Layout. *VLSI Design: An International Journal of Custom-Chip Design, Simulation and Testing*, Vol. 10, No. 1, 1999.

[37] W. Wang and M. Chrzanowska-Jeske. Generating Linear Arrays Using Symmetry Chain. *Proc. Intl. Workshop on Logic Synthesis*, pp. 115-119, June 1999.

[38] M. Chrzanowska-Jeske and Z. Wang. Mapping of Symmetric and Partially Symmetric Functions to CA-Type FPGAs. *Proc. of the IEEE Midwest Symposium on Circuits and Systems*, pp. 290-293, 1995.

[39] A. Mukherjee, R. Sudhakar, M. Marek-Sadowska, and S. I. Long. Wave Steering in YADDs: A Novel Non-Iterative Synthesis and Layout Technique. *Proc. Design Automation Conference*, pp. 446-471, 1999.

[40] M. Perkowski, M. Chrzanowska-Jeske, and Y. Xu. Lattice Diagrams Using Reed-Muller Logic. *Proc. Intl. Workshop Appl. Reed-Muller Expansions*, pp. 85-102, 1997.

[41] M. Chrzanowska-Jeske and J. Zhou. AND/EXOR Regular Function Representation. *Proc. of the IEEE Midwest Symposium on Circuits and Systems*, pp. 1034-1037, 1997.

[42] P. Lindgren, R. Drechsler and B. Becker. Synthesis of Pseudo-Kronecker Lattice Diagrams. *Proc. Intl. Workshop Appl. Reed –Muller Expansions*, pp. 197-204, 1999.

[43] A. Mishchenko, X. Wang, and T. Kam. A new Enhanced Constructive Decomposition and Mapping Algorithm. *Proc. Design Automation Conference*, pp. 143-147, June 2003.

[44] R. L. Ashenhurst. The Decomposition of Switching Functions. *Computational Lab*, Harvard University, Vol. 29, pp. 74-116, 1959.

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

[46] M. Chrzanowska-Jeske, W. Wang, J. Xia, and M. Jeske. Disjunctive Decomposition of Switching Functions Using Symmetry Information. *Proc. IEEE SBCCI*, pp. 67, September 2000.

[47] J. S. Zhang, M. Chrzanowska-Jeske, A. Mishchenko, and J. R. Burch. Fast Computation of Generalized Symmetries in Boolean Functions. *Proc. IWLS '04*, pp. 424-430, June 2004.

[48]http://www.ece.pdx.edu/~alanmi/research/lcr/index.htm

[49] V. N. Kravets and K. A. Sakallah. "Constructive library-aware synthesis using symmetries", *Proc. DATE '00*, pp. 208-216, 2000.