and T_t and the technology parameters. For all cases, we observe that the total latency is not significantly higher than the minimum source-sink delay of 2739 ps (from Table I).

VI. Conclusion

Automated buffered routing is a necessity in modern very large-scale integration design. The contributions of this paper are two new problem formulations for buffered routing for single- and multiple-clock domains. Both of these formulations address problems that will become more prominent in future designs. Any computer-aided design (CAD) tools currently performing buffer insertion will eventually have to deal with synchronizer insertion. Furthermore, any SoC routing CAD tools will have to handle routing across multiple clock domains due to the increasing use of IPs.

We solve both problems optimally in polynomial time via the RBP and GALS algorithms that build upon the fast path algorithm of [17]. Experimental results validate the correctness and practicality of the two algorithms for an aggressive technology.

Acknowledgment

The authors would like to thank H. Zhou for supplying fast path code and also to M. Thiagarajan for help with the figures and researching the background material on the MCFIFOs.

References

[1] C. J. Alpert, G. Gandham, J. Hu, S. T. Quay, J. L. Neves, and S. S. Saptnekar, "Steiner tree optimization for buffers and blockages and bays," *IEEE Trans. Comput.-Aided Design*, vol. 20, pp. 556–562, Apr. 2001.

[2] L. Carloni, K. McMillan, A. Saldanha, and A. Sangiovanni-Vincentelli, "A methodology for correct-by-construction latency insensitive design," *Proc. IEEE Int. Conf. Computer-Aided Design (ICCAD)*, 1999.

[3] D. Chapiro, "Globally asynchronous locally asynchronous systems," Ph.D. dissertation, Stanford Univ., Stanford, CA, 1984.

[4] T. Chelcea and S. Nowick, "Robust interfaces for mixed-timing systems with application to latency-insensitive protocols," in *Proc. ACM/IEEE Design Automation Conf. (DAC)*, 2001, pp. 21–26.

[5] P. Cocchini, "Concurrent flip-flop and repeater insertion for high performance integrated circuits," *Proc. IEEE Int. Conf. Computer-Aided Design (ICCAD)*, pp. 268–273, 2002.

[6] J. Cong, "Timing closure based on physical hierarchy," in *Proc. Int. Symp. Physical Design*, 2002, pp. 170–174.

[7] J. Cong, J. Fang, and K.-Y. Khoo, "An implicit connection graph maze routing algorithm for ECO routing," *Proc. IEEE Int. Conf. Computer-Aided Design (ICCAD)*, pp. 163–167, 1999.

[8] J. Cong and Z. Pan, "Interconnect performance estimation models for design planning," *IEEE Trans. Comput.-Aided Design*, vol. 20, pp. 739–752, June 2001.

[9] S. Hassoun, "Optimal use of 2-phase transparent latches in buffered maze routing," *Proc. IEEE Int. Symp. Circuits Syst.*, 2003.

[10] A. Hemani, T. Meincke, S. Kumar, A. Postula, T. Olsson, P. Nilsson, J. Obert, P. Ellervee, and D. Lundqvist, "Lowering power consumption in clock by using globally asynchronous locally synchronous design style," presented at the *Proc. ACM/IEEE Design Automation Conf. (DAC)*, 1999.

[11] J. Muttersbach, T. Villiger, H. Kaeslin, N. Felber, and W. Fichtner, "Globally-asynchronous locally-synchronous architectures to simplify the design of on-chip systems," presented at the *Proc. 12th Annu. IEEE Int. ASIC/SOC Conf.*, 1999.

[12] M. Lai and D. F. Wong, "Maze routing with buffer insertion and wiresizing," in *Proc. ACM/IEEE Design Automation Conf. (DAC)*, 2000, pp. 374–378.

[13] R. Lu, G. Zhong, C. Koh, and K. Chao, "Flip-flop and repeater insertion for early interconnect planning," in *Proc. Design Automation Test Europe Conf. (DATE)*, 2002, pp. 690–695.

[14] J. Rabaey, *Digital Integrated Circuits*. Englewood Cliffs, NJ: Prentice-Hall, 1996.

[15] J.-N. Seizovic, "Pipeline synchronization," *Proc. IEEE ASYNC*, 1994.

[16] L. P. P. P. van Ginneken, "Networks for minimal Elmore delay," *Proc. IEEE Int. Symp. Circuits Syst.*, pp. 865–868, 1990.

[17] H. Zhou, D. F. Wong, I.-M. Liu, and A. Aziz, "Simultaneous routing and buffer insertion with restrictions on buffer locations," *IEEE Trans. Comput.-Aided Design*, vol. 19, pp. 819–824, July 2000.

Fast Computation of Symmetries in Boolean Functions

Alan Mishchenko, *Member, IEEE*

***Abstract*—Symmetry detection in completely specified Boolean functions is important for several applications in logic synthesis, technology mapping, binary decision diagram (BDD) minimization, and testing. This paper presents a new algorithm to detect four basic types of two-variable symmetries. The algorithm detects all pairs of symmetric variables in one pass over the shared BDD of the multioutput function. The worst case complexity of this method is cubic in the number of BDD nodes, but on typical logic synthesis benchmarks the complexity appears to be linear. The computation is particularly efficient when the functions have multiple symmetries or no symmetries. Experiments show that the algorithm is faster than other known methods, and in some cases achieves a speedup of several orders of magnitude.**

***Index Terms*—Binary decision diagrams (BDDs), Boolean functions, recursive procedures, symmetric variables, symmetry, zero-suppressed binary decision diagrams (ZBDDs).**

I. Introduction

The problem of symmetry detection in Boolean functions has a long history and many applications, such as functional decomposition in technology-independent logic synthesis [6], [9], [11], Boolean matching in technology mapping [12], [13], and binary decision diagram (BDD) minimization [21].

Early methods to detect symmetries are based on checking the equality of two-variable cofactors of the function $F_{01} = F_{10}$ and $F_{00} = F_{11}$. Decomposition charts [18] and truth tables [4] have been used to compute and compare cofactors. Representing functions using BDDs [1] improved the efficiency of cofactor computation. However, computing multiple cofactor pairs is still expensive for large functions because repeated cofactoring leads to creating and deleting a large number of intermediate BDD nodes. In this paper, cofactor checking is referred to as the *naïve method* to detect symmetries.

To bypass the cofactor computation, recent more sophisticated approaches use dynamic BDD variable reordering [19], generalized Reed–Muller forms [23], and analysis of shared BDDs [16]. The latter approach was recently successfully applied to BDD minimization in [21].

Here we review the symmetry detection algorithm described in [16] and [21] as the most computationally efficient. This algorithm is based on the observation that the absence of symmetry between a pair of variables can, in many cases, be discovered without computing and comparing the cofactors. To prove the absence of symmetry counting

Manuscript received July 16, 2002; revised December 23, 2002 and May 7, 2003. This work was supported by a research grant from Intel Corporation.

The author is with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720 USA (e-mail: alanmi@eecs.berkeley.edu).

Digital Object Identifier 10.1109/TCAD.2003.818371

minterms as well as several other specialized traversals of the shared BDDs are performed. These traversals are relatively fast because they only explore the available BDD structure, without building new BDD nodes. In the end, the naïve symmetry check is applied to those variable pairs for which the existence of symmetry could not be disproved. This method is faster than the naïve method applied to all cofactor pairs. However, it has the following limitations, which become noticeable for functions with many inputs and large BDDs.

- It involves multiple BDD traversals.
- It often requires a substantial number of cofactors to be checked using the naïve method.
- It takes a long time to process some functions to discover that they have no symmetries.
- It detects only one type of "classical" two-variable symmetry and does not detect other useful symmetry types (for example, single-variable symmetries [6] or skew symmetries [23]).

The main contribution of this paper is in proposing a new algorithm for symmetry detection from the shared BDDs. This method improves on the previous approaches in several ways.

- It reduces the number of BDD traversals to one main traversal and several additional traversals performed from the main traversal.
- It is particularly efficient for benchmarks with few symmetries and no symmetries, leading to several order of magnitude speedups, compared to other methods.
- It can simultaneously (in one pass) detect four basic symmetries defined in [23].

The differences between the present work and the previous work [16], [21] are the following.

- The previous work is based on "negative thinking." It first detects as many nonsymmetric pairs as possible, and then uses the naïve check to prove that the remaining pairs are symmetric. The present work is based on "positive thinking." If the symmetries exist, they are detected and added to the set of computed symmetries. There is no search for the nonsymmetric variable pairs and no application of the naïve check to individually computed cofactor pairs.
- As evidenced by Table II in [21, p. 92], the most efficient component of the previous work is Idea 1 (counting minterms in the cofactors of the functions and proving nonsymmetric those variable pairs that have different minterm counts). The present work does not rely on minterm counting at all, neither explicitly nor implicitly.
- The present work is based on new BDD traversal procedures and uses symmetry graphs, represented by zero-suppressed binary decision diagrams (ZDDs), to efficiently store and manipulate the symmetry information.

A straightforward application of the proposed method computes symmetries individually for each output. A minor modification of the algorithm allows for the computation of symmetries that are common to *all* outputs, which is particularly useful for BDD minimization by variable reordering.

Several generalizations of the classical two-variable symmetries were proposed [2], [5], [10]. These methods incorporate classical two-variable symmetries as a particular case and use them to compute other symmetries. Therefore, the proposed efficient algorithm to compute two-variable symmetries can also be used to enhance the computation of the generalized symmetries.

The rest of the paper is organized as follows. Section II introduces definitions and theoretical foundations. Section III presents the new algorithm. Section IV discusses implementation issues. Section V presents the experimental results. Section VI concludes the paper.

II. Background

Functions discussed in this paper are Boolean completely specified functions; variables are Boolean variables. The definitions and notations introduced in this section are based on [23].

The *support* of F, $\mathrm{supp}(F)$, is the set of all variables on which F depends.

A *cofactor* of a function $F(\ldots, x_i, \ldots, x_j, \ldots)$ with respect to variables x_i and x_j is the function resulting from the substitution into F of specific values for x_i and x_j. For example, the cofactor of F with respect to $x_i = 0$ and $x_j = 1$ is function $F(\ldots, 0, \ldots, 1, \ldots)$, which is denoted F_{01}.

The Shannon expansion represents function F in the form $F = \bar{x}F_0 + xF_1$, where F_0 and F_1 are the negative and positive cofactors of the function F with respect to variable x.

Definition: Two variables x_i and x_j of function $\mathrm{F}(\ldots x_i, \ldots, x_j, \ldots)$ are *symmetric* if the function does not change when the variables are swapped

$$\mathrm{F}(\ldots, x_i, \ldots, x_j, \ldots) = \mathrm{F}(\ldots, x_j, \ldots, x_i, \ldots).$$

This symmetry is known as the *classical symmetry*, or the *nonskew nonequivalence symmetry*, denoted $x_i \ \mathrm{NE}\ x_j$. The definition of this symmetry translates into the following requirements for the cofactors of the functions $F_{01} = F_{10}$.

Other relationships among the two-variable cofactors lead to other symmetry types. Requiring the equality of $F_{00} = F_{11}$ yields the *nonskew equivalence symmetry*, denoted $x_i\ E\ x_j$. Complementing one of the cofactors in the equalities $F_{01} = F_{10}$ and $F_{00} = F_{11}$ yields two skew symmetries: the *skew nonequivalence symmetry*, denoted $x_i\ !\mathrm{NE}\ x_j\ (F_{01} = \bar{F}_{01})$, and the *skew equivalence symmetry*, denoted $x_i\ !E\ x_j\ (F_{00} = \bar{F}_{11})$.

For completeness, two other ways of generalizing two-variable symmetries are mentioned. With four cofactors, F_{00}, F_{01}, F_{10}, and F_{11}, it is possible to create 12 different cofactor equalities (six equalities without complementation and six with complementation). Some of them simply state that the function does not depend on a variable. For example, $F_{00} = F_{01}$ means that the function does not depend on x_j. Such symmetries are called *single-variable symmetries* [6].

The above symmetries are defined using the relationship among cofactor pairs. Another generalization considers cofactor groups that may include more than two cofactors. This approach yields 30 different symmetries [5], [2], which subsume all previously defined two-variable symmetries. The concept of two-variable symmetries has also been extended to larger groups of variables [10], [15].

In this paper, we focus on the four basic two-variable symmetries: $\mathrm{NE}, E, !\mathrm{NE}, !E$. The proposed method finds all pairs of symmetric variables. Using this information, larger groups of symmetric variables can be constructed by applying the transitivity.

Theorem 1 [23]: In each group of conditions below, any two of the three conditions imply the third, as follows.

$$\begin{array}{llll}
\text{Group A:} & (1)\ x_i\ \mathrm{NE}\ x_j, & (2)\ x_j\ \mathrm{NE}\ x_k, & (3)\ x_i\ \mathrm{NE}\ x_k. \\
\text{Group B:} & (1)\ x_i\ \mathrm{E}\ x_j, & (2)\ x_j\ \mathrm{E}\ x_k, & (3)\ x_i\ \mathrm{NE}\ x_k. \\
\text{Group C:} & (1)\ x_i\ !\mathrm{NE}\ x_j, & (2)\ x_j\ !\mathrm{NE}\ x_k, & (3)\ x_i\ \mathrm{NE}\ x_k. \\
\text{Group D:} & (1)\ x_i\ !\mathrm{E}\ x_j, & (2)\ x_j\ !\mathrm{E}\ x_k, & (3)\ x_i\ \mathrm{NE}\ x_k. \\
\text{Group E:} & (1)\ x_i\ !\mathrm{E}\ x_j, & (2)\ x_j\ !\mathrm{NE}\ x_k, & (3)\ x_i\ \mathrm{E}\ x_k.
\end{array}$$

Theorem 1 does not hold for incompletely specified Boolean functions.

Another useful property follows from the canonicity of the Shannon expansion.

Theorem 2 [23]: Let F be a function and x_i, x_j, and x_k be three variables belonging to the support of F. Function F is symmetric in x_i

```
solution RecursiveProcedure( problem P )
{
Step 1:    If P is a trivial case, return the solution.
Step 2:    Cofactor P w.r.t. a variable to get subproblems $P_0$ and $P_1$.
Step 3:    Get the partial solutions, $S_0$ and $S_1$, of subproblem $P_0$ and $P_1$.
Step 4:    Derive S, the solution of P, from the partial solutions $S_0$ and $S_1$.
Step 5:    Return S.
}
```

Fig. 1. Pseudocode of a generic recursive algorithm.

and x_j with any symmetries belonging to the set $\{\mathrm{NE}, E, !\mathrm{NE}, !E\}$ iff both cofactors of F with respect to x_k are symmetric in x_i and x_j with the same symmetries.

Next we define the *symmetry graph*, $G_F(V, E)$, which is used in this paper as a canonical representation of the symmetry information. The vertices of the symmetry graph correspond to variables in the support of the function, while the edges correspond to the two-variable symmetries. Each symmetry type corresponds to an edge type. Thus, if two variables have several symmetries, the corresponding vertices are connected by as many edges of different type.

Several graph operations are used in the sequel. The *union* and *intersection* are defined and denoted similarly to the set-theoretic union $(\cup)$ and intersection $(\cap)$ of 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 vertices of variables in Y.

Two operations on variable sets are considered, the set difference $(-)$ and the number of elements in the set (denoted by vertical bars, for example $|\mathrm{S}|$). Thus, the number of support variable is denoted $|\mathrm{supp}(F)|$.

The concepts "symmetry graph," "the set of edges in the symmetry graph," "symmetry information," and "the set of pair-wise variable symmetries" are used interchangeably in this paper.

III. Symmetry Computation Algorithm

This section presents the main contribution of the paper, a recursive symmetry computation algorithm.

A. Generic Recursive Procedure

The symmetry detection algorithm falls into a general category of recursive algorithms, which work by cofactoring the problem with respect to a variable, solving the resulting subproblems, and then deriving the solution of the initial problem. The pseudocode of the generic recursive procedure is shown in Fig. 1.

If decision diagrams [1], [14] are used to represent the input parameters of a recursive procedure, the partial results are cached to prevent multiple calls with the same input parameters. Caching is responsible for the reduction of computational complexity from exponential to polynomial in the sizes of the representation of the parameters. The cache lookup is typically placed before Step 2 and the cache insertion is placed before Step 5. To keep the pseudocode short, these steps are omitted in this paper.

B. Computational Core

Recursive computation of symmetries is based on Theorem 2, which allows us to compute the symmetries of the function if the symmetries for the cofactors are known. The pseudocode shown in Fig. 2 illustrates this computation. The procedure *ComputeSymmetries* takes the function whose symmetries should be computed. The internal recursive procedure *ComputeSymmetries_rec* takes the function and a set of variables initialized to the support of the function. Both procedures return the symmetry graph, whose edges represent pair-wise symmetric variables.

```
symmgraph ComputeSymmetries( function F )
{
    return ComputeSymmetries_rec( F, supp(F) );
}

symmgraph ComputeSymmetries_rec( function F, varset V )
{
Step 1:    if ( F is a constant function )
               return CompleteGraph( V );
Step 2:    x = supp( F );
           ( $F_0$, $F_1$ ) = Cofactors( F, x );
Step 3:    RemainingVars  = supp( F ) – x;
           $S_0$ = ComputeSymmetries_rec( $F_0$, RemainingVars );
           if ( $S_0 = \varnothing$ )
               $S_1 = \varnothing$;
           else
               $S_1$ = ComputeSymmetries_rec( $F_1$, RemainingVars );
Step 4:    Y = SymmetricVars( $F_0$, $F_1$, RemainingVars );
           $S_2 = x \times Y$;
           $S_3$ = CompleteGraph( V – supp( F ) );
           $S = (S_0 \cap S_1) \cup S_2 \cup S_3$;
Step 5:    return S;
}
```

Fig. 2. Pseudocode of symmetry computation core.

Step 1) Procedure *ComputeSymmetries_rec* checks the function for being a constant. The constant function is symmetric in all its variables. Therefore, the procedure returns the *complete* symmetry graph with vertices representing variables in V. This graph has edges between all vertex pairs.

Step 2) Cofactoring F with respect to variable $x \in \mathrm{supp}(F)$ is a standard operation. When BDDs are used to represent the function, cofactoring with respect to the topmost variable in F is a constant time operation. Note that because F is cofactored using a variable in its support, the cofactors, F_0 and F_1, are different Boolean functions. This is important for procedure *SymmetricVars*, used later in the pseudocode.

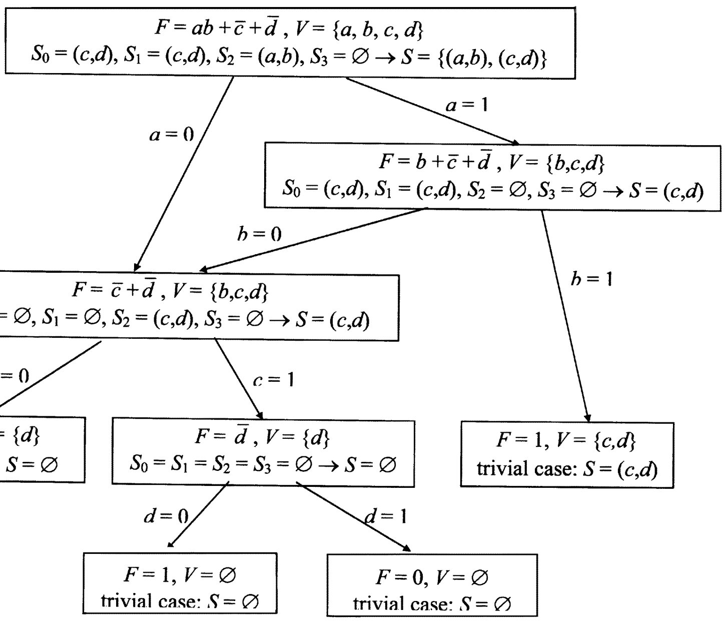
Step 3) Computing symmetries recursively is performed by calling *ComputeSymmetries_rec* with the cofactors and the variable set including the support of F without the cofactoring variable x. However, if the first call returns no symmetries, the second call is skipped, because, according to Theorem 2, if one of the cofactors does not have symmetries, the function does not have symmetries as well.

Step 4) The solution of the problem is found by augmenting the intersection, $S_0 \cap S_1$, with two symmetry graphs, S_2 and S_3. The intersection $S_0 \cap S_1$ consists of edges common to both subgraphs. According to Theorem 2, only the symmetries of *both* cofactors are the symmetries of the function.

The additional symmetries have two distinct origins. The symmetries in S_2 involve the cofactoring variable x. These symmetries are not included in S_0 and S_1 because x is not in the supports of the cofactors used in Step 3. Symmetries in S_2 are found by pairing x with all variables $y \in Y$, such that $F_0 |_{y=1} = F_1 |_{y=0}$. The set Y satisfying this condition is returned by procedure *SymmetricVars*. This step corresponds to finding all nonskew nonequivalence symmetries with variable x. The extension to compute other symmetries is discussed later.

The second part of the additional symmetries (S_3) is due to those variables in V that are not in the support of F. Pairs of these variables are symmetric as far as function F is concerned. Note that initially procedure *ComputeSymmetries_rec* is called with V equal to the support of F. In this case, the set of additional symmetries S_3 is empty. However, in later calls to *ComputeSymmetries_rec*, the support of a cofactor may not depend on all variables in *RemainingVars*. If set $V - \mathrm{supp}(F)$ includes at least two variables, the set of symmetries S_3 is not empty.

The algorithm in Fig. 2 is particularly efficient when applied to functions with multiple symmetries and no symmetries at all. The former is true because the BDDs of symmetric and nearly symmetric functions



$F = ab + \bar{c} + \bar{d}$, $V = \{a, b, c, d\}$
$S_0 = (c,d)$, $S_1 = (c,d)$, $S_2 = (a,b)$, $S_3 = \varnothing \rightarrow S = \{(a,b), (c,d)\}$
$a = 0$
$a = 1$
$F = b + \bar{c} + \bar{d}$, $V = \{b,c,d\}$
$S_0 = (c,d)$, $S_1 = (c,d)$, $S_2 = \varnothing$, $S_3 = \varnothing \rightarrow S = (c,d)$
$b = 0$
$b = 1$
$F = \bar{c} + \bar{d}$, $V = \{b,c,d\}$
$S_0 = \varnothing$, $S_1 = \varnothing$, $S_2 = (c,d)$, $S_3 = \varnothing \rightarrow S = (c,d)$
$c = 0$
$c = 1$
$F = 1$, $V = \{d\}$
trivial case: $S = \varnothing$
$F = \bar{d}$, $V = \{d\}$
$S_0 = S_1 = S_2 = S_3 = \varnothing \rightarrow S = \varnothing$
$F = 1$, $V = \{c,d\}$
trivial case: $S = (c,d)$
$d = 0$
$d = 1$
$F = 1$, $V = \varnothing$
trivial case: $S = \varnothing$
$F = 0$, $V = \varnothing$
trivial case: $S = \varnothing$

Fig. 3. An example of symmetry computation.

```
varset SymmetricVars( function G, function H, varset Y )
{
Step 1:     if ( G = H )
                 return Y;
            if ( G = const  and  H = const )
                 return ∅;
Step 2:     z = Var( Y );
            ( G0, G1 ) = Cofactors( G, z );
            ( H0, H1 ) = Cofactors( H, z );
Step 3:     R0 = SymmetricVars( G0, H0, Y – z );
            if ( R0 = ∅ )
                 R1 = ∅;
            else
                 R1 = SymmetricVars( G1, H1, Y – z );
Step 4:     R = R0 ∩ R1;
            if ( G1 = H0 )
                 R = R ∪ z;
Step 5:     return R;
}
```

Fig. 4. Computing the variables that are symmetric with a variable.

tend to be small and regular. This increases the cache hit rate and leads to faster processing. The situation when the function has no symmetries is detected early in the process. Thus, when the computation for one of the cofactors results in the empty set, the algorithm does not compute symmetries for the other cofactor. This shortcut leads to significant speedups for large benchmarks.

The worst case complexity of *ComputeSymmetries_rec* is cubic in the number of the BDD nodes, because the complexity of this procedure is linear while each call to *SymmetricVars*, performed inside *ComputeSymmetries_rec*, has the worst case quadratic complexity. However, for benchmark functions, the experimentally observed runtime is close to linear in the number of the BDD nodes.

Example: The symmetry computation algorithm is illustrated for Boolean function $F = ab + \bar{c} + \bar{d}$. The tree of recursive calls to *ComputeSymmetries_rec* is in Fig. 3. Initially, *ComputeSymmetries_rec* is called with F and its support: $\{a, b, c, d\}$. Cofactoring with respect to a leads to two calls, with functions $F_0 = \bar{c} + \bar{d}$ and $F_1 = b + \bar{c} + \bar{d}$, etc. Note that the trivial cases with one variable in V result in the empty set of symmetries, while the trivial calls with two variables in V result in returning the variable pairs.

C. Detecting Variables Symmetric With Variable

This section discusses procedure *SymmetricVars* with the pseudocode shown in Fig. 4. The procedure takes F_0 and F_1, the two cofactors of F with respect to x and the set of candidate variables Y (to avoid multiple indices, the cofactors are denoted G and H in the pseudocode). The procedure returns the subset of Y such that, for each variable z in this subset, $F_0|_{z=1} = F_1|_{z=0}$. It means that z is symmetric with x in the original function F.

Step 1) If functions G and H are equal, so are their cofactors. As a result, all variables in Y satisfy the condition. If both functions G and H are (nonequal) constants, then their cofactors are never equal. In this case, the procedure returns the empty set.

Step 2) Any variable z in Y is selected and the functions are cofactored with respect to this variable.

Step 3) For a variable to belong to the solution, it should belong to the solution for both cofactors (this, again, follows from the canonicity of the Shannon expansion). So if one of the subproblems has the empty solution, there is no need to attempt solving another one. This is another shortcut, which accounts for efficient processing of functions without symmetries.

Step 4) The resulting set of variables is the intersection of the variable sets for the cofactors. Now we recall that $G = F_0$ and $H = F_1$. Therefore, condition $F_{01} = F_{10}$ translates into $G_1 = H_0$. When this is true, variable z is added to the resulting set. This completes the recursive call.

D. Computing the Complete Graph

The complete graph can be computed using procedure *Tuples*(V, i), where $i = 2$. This procedure returns the set of subsets of V composed of exactly i elements. The pseudocode of this procedure can be found in [8, p. 66].

E. Detecting Multiple Symmetry Types

The extension to treat multiple symmetry types is straightforward. In the pseudocode of *ComputeSymmetries_rec*, instead of implicitly checking condition $F_{01} = F_{10}$ for the existence of the nonskew nonequivalence symmetry in *SymmetricVars*, other conditions can be checked as well. Thus, checking condition $F_{00} = F_{11}$ detects the nonskew equivalence symmetry, while checking conditions $F_{01} = \bar{F}_{01}$ and $F_{00} = \bar{F}_{11}$ detects the skew nonequivalence and the skew equivalence symmetries, respectively.

TABLE I
COMPUTATION OF SYMMETRIES OF LARGE MCNC BENCHMARKS

Benchmark Statistics				Symmetries		Runtime, s						
name	ins	outs	|BDD|	pairs	ratio,%	reading	naïve	old	new	[16]	[21]	[23]
alu2	10	6	231	4	3.17	0.01	0.01	0.01	0.01	-	-	-
alu4	14	8	1182	6	1.71	0.02	0.02	0.01	0.01	-	-	-
dalu	75	16	1402	982	7.83	0.55	0.92	0.16	0.07	32.5	0.60	-
des	256	245	3238	1264	6.95	1.57	0.51	0.27	0.05	52.5	5.60	3.60
frg2	143	139	1907	1353	9.32	0.18	0.23	0.10	0.02	5.3	2.70	14.15
i10	257	224	49387	3746	3.39	20.48	285.42	3.89	1.63	-	-	-
k2	45	45	1316	338	3.61	0.22	0.57	0.04	0.07	-	-	-
pair	173	137	4976	1910	5.24	3.29	2.08	0.65	0.08	-	3.80	-
rot	135	107	6004	364	1.87	1.78	7.12	0.32	0.14	-	5.70	-
too_large	38	3	839	17	0.92	0.18	0.20	0.01	0.02	3.8	0.50	-
C1355	41	32	29578	0	0.00	18.64	61.45	52.96	0.02	-	2.90	-
C1908	33	25	9519	248	2.23	1.13	4.71	2.08	0.06	1126.5	0.70	-
C2670	233	140	4971	1547	4.78	2.84	25.63	3.25	0.16	-	7.20	-
C3540	50	22	24475	81	0.60	16.43	27.36	1.14	0.67	-	3.00	-
C432	36	7	1226	0	0.00	0.11	0.86	0.04	0.01	23.2	-	-
C499	41	32	27472	0	0.00	3.59	57.03	51.18	0.02	-	4.00	-
C5315	178	123	2492	521	0.83	1.38	2.79	0.84	0.13	636.7	2.10	-
C7552	207	108	10674	1879	1.31	8.14	42.77	13.19	0.32	-	9.30	-
C880	60	26	17755	262	4.01	1.81	8.48	0.34	0.19	7.7	-	-
Total						82.35	528.16	130.48	3.68			
Ratio, %						16.0	100.0	24.7	0.7			

The only part of the pseudocode in Figs. 2 and 4 that depends on a symmetry type is Steps 1 and 4 of the procedure *SymmetricVars*. When checking for skew symmetries, Step 1 should be modified to check $G = \bar{H}$. Furthermore, in Step 4, a variable should be added to the resulting set R when $G_0 = H_1$ (to check for the nonskew equivalence symmetry) or when $G_1 = \bar{H}_0$ and $G_0 = \bar{H}_1$ (to check for the skew nonequivalence and the skew equivalence symmetries, respectively).

All four symmetry types can be computed by a modified version of *SymmetricVars*. In this case, the procedure returns four sets of variables symmetric with a variable. Each set corresponds to each of the four basic symmetry types. The next subsection shows an efficient implementation of the procedure simultaneously returning four sets of variables.

IV. Implementation Issues

In the implementation of symmetry computation, BDDs [1] represent Boolean functions and ZDDs [14] represent variable sets and symmetry graphs. This choice is motivated by the following practical considerations.

Using BDDs to represent the functions allows for a type of computation, which only explores the BDD structure without modifying it. The algorithms in Figs. 2 and 4 work without building new BDD nodes. This makes the proposed implementation very fast, as evidenced by a similar implementation of operator *ITE_constant* [7]. Additionally, using BDDs allows for efficient caching of the intermediate results across multiple recursive calls.

ZDDs, on the other hand, provide a canonical representation for the combinatorial sets. The set representation can be extended to represent symmetry graphs. A graph is seen as a set of pairs of vertices connected by edges. This representation is used to solve a variety of graph optimization problems [3]. Note that the ZDD representation of graphs corresponds to the graph representation by adjacency lists, while the better-known BDD representation of graphs by transition relations corresponds to the representation of graphs by the adjacency matrix. For symmetry computation, the ZDD representation is more natural because it does not require encoding of the graph vertices necessary for the construction of the transition relation. Also, symmetry graphs are typically sparse, which makes the sizes of representations using ZDDs significantly smaller.

In the ZDD representation, two variables are used for each primary input variable of the function. These variables stand for the positive and the negative literals. These two ZDD variables can be used to efficiently represent four variable sets. Given four types of symmetries among two primary input variables, the symmetry types can be encoded as follows: 11, 10, 01, and 00. The first number stands for the literal (positive or negative) of the first variable, similarly for the second number.

It should be noted that, although the algorithm does not build new BDD nodes, a small number of ZDD nodes is created to manipulate the symmetry graphs and the variable sets in the recursive procedures. However, the experiments show that the increase in the number of ZDD nodes is negligible, compared to the size of the shared BDDs of the original functions. The shared BDDs are constructed only once at the beginning, possibly with the use of dynamic BDD variable reordering. They are not modified during the symmetry computation.

V. Experimental Results

The proposed algorithm was implemented in C with CUDD Decision Diagram Package [22]. The source code is available as part of EXTRA Library [17]. Table I summarizes the experiments conducted using the largest MCNC benchmark circuits [24]. The program was run on a 933-MHz Pentium III PC under MS Windows 2000. The memory requirements of the algorithm amounted to less than 10% of the memory allocated by the CUDD package to construct the shared BDDs of the benchmark functions.

The following notation is used in Table I. The first four columns show the benchmark parameters: the name, the number of inputs and outputs, and the number of BDD nodes after reading and reordering by the sifting algorithm [22]. The next two columns show information about symmetries. Column "pairs" gives the total number of all classical nonequivalent symmetric variable pairs in all outputs. Column "ratio" lists the percentage of symmetric pairs to the total number of variable pairs in all outputs. The values in column "ratio" were the same for all the tested algorithms, and equal to the results reported in the last columns of Table II in [16]. The fact that the symmetry information computed using all the available algorithms is identical is the evidence of the correctness of the proposed implementation.

The right part of Table I compares the runtimes of several symmetry computation algorithms. Column "reading" gives the CPU time needed to read the benchmark file, construct the BDD, and perform the reordering. Other columns show the time needed to compute the symmetries from the shared BDD.

Column "naïve" shows the runtime of the naïve algorithm, which derives the two-variable cofactors for all variable pairs and checks their equality. Column "old" shows the runtime of an implementation using CUDD of the symmetry detection algorithm reported in [16]. Column "new" shows the runtime of the algorithm presented in this paper. These three algorithms have been run independently, to prevent the reuse of computed results stored in the internal cache of the CUDD package.

The last three columns show the runtime of symmetry detection reported in [16], [21], and [23] using SPARC Station 2, SPARC Station 20, and DEC5000, respectively. The dash in the column of the table means that the results are not available in the above publications.

Several words should be said about the fairness of these comparisons. The settings of the experiments conducted in [16] exactly correspond to our settings. However, a different BDD package was used, and the computer employed in 1993 (SPARC Station 2) was slower. In order to compare the performance using the same BDD package and the same computer, this algorithm was reimplemented with CUDD. The runtime of this implementation is reported in column "old."

The experiment conducted in [21] differs from the experiment in this paper. However, the difference does not devalue the comparison. The goal of symmetry detection in [21] is to minimize the BDD representation for partially symmetric functions. Therefore, only the symmetric variable pairs, which are common for *all* outputs, are computed and reported in [21]. Computing *common* symmetries is considerably simpler than computing symmetries for individual outputs (the way it was done by the proposed algorithm) because if a variable pair is proved to be nonsymmetric for one output, there is no need to test the same pair for other outputs.

Finally, the experiment in [23] was performed using small benchmarks. These benchmarks are not listed in Table I (except "des" and "frg2") because the proposed algorithm takes less than 0.01 s to process them. It should also be noted that runtimes in [23] are given for single outputs of the benchmark function (output #120 for benchmark "des," and output #134 for benchmark "frg2"), while runtimes of other algorithms refer to all outputs of these benchmarks.

VI. Conclusion

In this paper, a new algorithm is presented to compute pair-wise symmetries of completely specified Boolean functions. The algorithm can be briefly characterized as follows.

- It works on the shared BDD of multioutput functions and computes the symmetry information for each output individually. The worst case complexity of the algorithms is cubic in the number of BDD nodes, but close to linear for the practical benchmarks.
- It exploits the compactness and canonicity of the ZDD representation of the combinatorial sets to store the symmetry information computed for a node in the shared BDD.
- It computes four types of two-variable symmetries: the nonequivalence symmetries and equivalence symmetries; both can be skewed and nonskewed.
- It is particularly efficient when applied to multioutput Boolean functions with multiple symmetries or no symmetries at all (see benchmarks "c432," "c499," "c1355" in Table I).

Experimental results show that the overall performance of the algorithm is significantly better than other algorithms reported in the literature.

Future work in this area includes extensions of the algorithms to generalized symmetries [2], [5], [10] and applications to improve the quality of functional decomposition and speedup Boolean matching.

References

[1] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, pp. 677–691, Aug. 1986.

[2] M. Chrzanowska-Jeske, "Generalized symmetric variables," presented at the Int. Conf. Electronics, Circuits, and Systems, Msida, Malta, 2001.

[3] O. Coudert, "Solving graph optimization problems with ZBDDs," in *Proc. Eur. Design and Test Conf.*, 1997, pp. 224–228.

[4] D. L. Dietmeyer and P. R. Schneider, "Identification of symmetry, redundancy, and equivalence of Boolean functions," *IEEE Trans. Electron. Comput.*, vol. EC-16, pp. 804–817, Dec. 1967.

[5] B. T. Drucker, C. M. Files, M. A. Perkowski, and M. Chrzanowska-Jeske, "Polarized pseudo-Kronecker symmetry with an application to the synthesis of lattice decision diagrams," in *Proc. Int. Conf. Computational Intelligence and Multimedia Applications*, 1998, pp. 745–755.

[6] C. R. Edward and S. L. Hurst, "A digital synthesis procedure under function symmetries and mapping methods," *IEEE Trans. Comput.*, vol. C-27, pp. 985–997, Nov. 1978.

[7] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. Norwell, MA: Kluwer, 1996.

[8] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli, *Synthesis of Finite State Machines: Functional Optimization*. Norwell, MA: Kluwer, 1997.

[9] B.-G. Kim and D. L. Dietmeyer, "Multilevel logic synthesis of symmetric switching functions," *IEEE Trans. Computer-Aided Design*, vol. 10, pp. 436–446, Apr. 1991.

[10] V. N. Kravets and K. A. Sakallah, "Generalized symmetries in Boolean functions," in *Proc. Int. Conf. Computer Aided Design*, 2000, pp. 526–532.

[11] V. N. Kravets, "Constructive multi-level synthesis by way of functional properties," Ph.D. dissertation, Univ. Michigan, Ann Arbor, 2001.

[12] Y.-T. Lai, S. Sastry, and M. Pedram, "Boolean matching using binary decision diagrams with applications to logic synthesis and verification," in *Proc. Int. Conf. Computer Aided Design*, 1992, pp. 452–458.

[13] F. Mailhot and G. De Micheli, "Technology mapping using Boolean matching and don't care sets," in *Proc. Eur. Design Automation Conf.*, 1990, pp. 212–216.

[14] S. Minato, "Zero-suppressed BDDs for set manipulation in combinatorial problems," in *Proc. Design Automation Conf.*, 1993, pp. 272–277.

[15] J. Mohnke, P. Molitor, and S. Malik, "Limits of using signatures for permutation independent Boolean comparison," *Formal Methods Syst. Des.*, vol. 21, no. 2, pp. 167–191, Sept. 2002.

[16] D. Möller, J. Mohnke, and M. Weber, "Detection of symmetry of Boolean functions represented by ROBDDs," in *Proc. Int. Conf. Computer Aided Design*, 1993, pp. 680–684.

[17] A. Mishchenko. EXTRA library of DD procedures. [Online]. Available: http://www.ee.pdx.edu/~alanmi/research/extra.htm

[18] A. Mukhopadhyay, "Detection of total or partial symmetry of a switching function with the use of decomposition charts," *IEEE Trans. Electron. Comput.*, vol. 16, pp. 553–557, Oct. 1963.

[19] S. Panda, F. Somenzi, and B. F. Plessier, "Symmetry detection and dynamic variable ordering of decision diagrams," in *Proc. Int. Conf. Computer Aided Design*, 1994, pp. 628–631.

[20] T. Sasao, "A new expansion of symmetric functions and their application to nondisjoint functional decompositions for LUT-type FPGAs," in *Proc. Int. Workshop Logic Synthesis*, 2000, pp. 105–110.

[21] Ch. Scholl, D. Möller, P. Molitor, and R. Drechsler, "BDD minimization using symmetries," *IEEE Trans. Computer-Aided Design*, vol. 18, pp. 81–100, Feb. 1999.

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

[23] C.-C. Tsai and M. Marek-Sadowska, "Generalized Reed–Muller forms as a tool to detect symmetries," *IEEE Trans. Comput.*, vol. 45, pp. 33–40, Jan. 1996.

[24] S. Yang, "Logic synthesis and optimization benchmarks user guide (version 3.0)," Microelectronics Center of North Carolina, Research Triangle Park, NC, 1991.