A Transform-Parametric Approach to Boolean Matching

Giovanni Agosta†, Francesco Bruschi†, Gerardo Pelosi‡, Donatella Sciuto†
† Politecnico di Milano, Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci 32 – 20133 Milano, Italy
Email: {agosta,bruschi,sciuto}@elet.polimi.it
‡ Università degli Studi di Bergamo
Dipartimento di Ingegneria dell'Informazione e Metodi Matematici
Viale Marconi 5 – 20044 Dalmine (BG), Italy
Email: gerardo.pelosi@unibg.it

***Abstract*— In this paper, we face the problem of P-equivalence Boolean matching. We outline a formal framework that unifies some of the spectral and canonical form-based approaches to the problem.**

As a first major contribution, we show how these approaches are particular cases of a single generic algorithm, parametric with respect to a given linear transformation of the input function.

As a second major contribution, we identify a linear transformation that can be used to significantly speed up Boolean matching with respect to the state of the art. Experimental results show that, on average, our approach is five times faster than the main competitor on 20-variables input functions, and scales better, allowing to match even larger components.

I. Introduction

In the synthesis of Boolean functions using a target technology, a problem known as *technology mapping*, one of the main issues is to determine whether a library contains a component that can implement a given function. Each component implements a set of functions that are all equivalent when permuting or negating the input lines.

To determine whether a given library cell C, represented by a Boolean function f_C, can implement a function f, it is therefore necessary to determine whether there is a permutation π of the input variables of f_C such that $f_C \circ \pi = f$. This problem is known as P-equivalence *Boolean matching* [4], and it is a key step in any technology mapping synthesis process because the time needed to match a function with a component of a library significantly impacts on the synthesis time.

Over the past 30 years, many methods and algorithms have been defined in literature to face this problem [4]. Nevertheless, new effective approaches have recently been developed, and have gained the interest of the scientific and industrial community, proving that the problem is still challenging.

In this paper, we propose a novel approach that tries to unify some of the most effective Boolean matching approaches based on *canonical forms*, ranging from those based on spectral function analysis [10], defined and developed in the '70s, to the most recent, based on functions representation by means of cofactors [1].

This paper is an extended version of [2]

The present work does not consider the N-equivalence problem, since efficient canonical form-based solutions for this problem are already stated in [5], and can be integrated with P-equivalence canonical forms [7].

We point out that the previous approaches are particular cases of a more general framework, parametric with respect to a linear function transformation ρ. We give a formal description of a set of linear transformations ρ that can be successfully employed in Boolean matching. Moreover, we identify a particular ρ that yields better performance than the previous approaches, by combining the ability to identify the canonical form using only the values of the transformed function in few points with an efficient, on-demand algorithm to individually compute those values.

A comparison with the existing state of the art in the field has been carried out implementing our algorithms by means of the well known *CUDD* Binary Decision Diagrams (BDD) manipulation package [16], since BDDs are the most flexible representation format for Boolean functions, and are widely adopted both in the industry and in research works. For an accurate and fair comparison, we reimplemented the most performant previous approach [1], and performed tests on both random function sets and specific functions that are considered worst cases.

The paper is organized as follows. In Section II we address the state of the art in the field. In Section III we introduce the unified framework for the computation of some canonical forms, as well as a new transformation that can be applied to Boolean functions to speed up the computation of the canonical form. In Section IV we provide experimental evidence that shows how our technique improves over the state of the art in terms of performance. Finally, we draw some conclusions and outline future works in Section V.

II. Related Works

The problem of Boolean matching has been the subject of many research works. [4] provides a survey of the main approaches. There are three main classes of matching algorithms: spectral methods, signature-based methods, and canonical form-based methods.

Spectral methods [3], [8], [14] exploit the fact that NPN-equivalence in the time domain translates to equivalence under coefficient permutations in the sequence domain of the Walsh Transform. These methods suffer from exponential complexity of the average case, since the cost of computing a Fast Walsh Transform is polynomial in the size of the input set (the last column of the truth table), which is of size 2^n with respect to the number of input variables. Little optimization is possible, except parallelizing the transformation algorithm, since all coefficients are computed at the same time, even when a few of them could be used to detect non-equivalent functions.

Signature-based methods use a compact representation of the Boolean functions. The signature is usually not a complete representation of the function, but it is supposed to include sufficient information to allow non-equivalent functions to be detected. In [15], it is noted that signature-based methods fail in detecting alias groups. A survey of signatures used in Boolean matching can be found in [17].

Canonical form-based methods use functions C that map any Boolean function f to a $C(f)$ such that $f' \sim f \Leftrightarrow C(f') = C(f)$. [5] introduces a canonical form for N-equivalence, and a semi-canonical form for P-equivalence. More recent developments have led to canonical forms for P-equivalence as well [1], [7], [11]. In this paper, we compare our approach to the most performant of these previous works [1].

Other recent approaches such as [12] focus on symmetry detection. These works are orthogonal to our own, and may be usefully integrated with it.

Finally, other approaches such as [6], [9] rely on a preliminary exploration of the function space that allows the precomputation of minterm positions. This information is then saved into huge lookup tables. [6] improves the approach by pruning the search tree, using signatures (including first order cofactors) and symmetry checks. These approaches are slower than canonical form based methods [1], except for very small numbers of input variables, but their main drawback is the large memory requirements, which effectively limit its applicability to functions with up to a maximum of dozen variables [6], [9]. Canonical form-based approaches, on the other hand, scales well up to over twenty input variables.

III. P-Equivalence Canonical Form

In this section, we introduce a unified approach to canonical form-based boolean matching. First, we define a generalized canonical form, and discuss the property of the function representation needed to compute it. Then, we describe the variable ordering algorithm that will be used in the experiments.

A. Generalized Lexicographic Canonical Form

The Boolean matching problem can be formally stated as follows: consider two Boolean functions f' and f'', of n variables each. Consider now all the possible permutations P of the input variables of f' (a variable permutation can be represented as a bijective function that maps an ordered variable sequence onto itself):

$$P = \left\{ \pi \mid (x_{n-1}, \ldots, x_0) \mapsto \left(x_{n-1}^{\pi(n-1)}, \ldots, x_0^{\pi(0)} \right) \right\}$$

If we consider π as an operator that can be applied to a function, we can denote the function f with permutated inputs with the expression $f \circ \pi$. f' is then P-equivalent (or *it matches*) to f'' if and only if there exists a $\pi \in P$ such that $f' \circ \pi = f''$.

One of the classes of approaches for the solution of the boolean matching problem relies on the use of *canonical representations* of functions. A canonical form is obtained by the application of a transformation C that maps any Boolean function f to a $C(f)$ in such a way that any other function f' P-equivalent to f has the same canonical form $C(f)$: $f' \sim f \Leftrightarrow C(f') = C(f)$. A canonical form for Boolean functions can be naturally introduced by representing functions as strings composed of their ordered output values. The lexicographic comparison of such strings induces an ordering and allows the definition of the canonical form as the lexicographic maximum of the set of P-equivalent functions:

$$C(f) = \underset{\pi \in P}{max} \{ f \circ \pi \}$$

It is straightforward that $C(f)$ has the properties of a canonical representation of f with respect to P-equivalence.

Note that the lexicographical maximum can be defined for all those function families that have a codomain on which an order relation is defined. Also, the above definitions can be restated for the dual case of the lexicographical minimum. In the rest of the paper, we will consider only the case of lexicographical maximum.

A trivial way to identify the lexicographical maximum of a P-equivalence class would be to apply all the possible permutations and to choose the maximal. However, this method incurs in an $O(n!)$ complexity, where n is the number of input variables. An efficient algorithm to find the lexicographical maximum of a given function is presented in [1]. This algorithm still has to face the issue of dealing with functions that have a codomain of small cardinality ($\{0, 1\}$), and therefore cause frequent collisions when trying to discriminate variables. In fact, the more a function exhibits different output values, the easier it is for the algorithm to assess a variable ordering, and the ability to exhibit different output values depends on the cardinality of the codomain.

That is why in [1] the values of a Boolean function f are not directly used as the input of the algorithm. Instead, a bijective linear transformation is first applied to the function. The transformation computes, for each element I of the powerset of the set of input variables $\{x_{n-1}, \ldots, x_0\}$, the number of minterms in the cofactor associated with that element, that is $|f_I|$. Of course, the *cofactor transformation* is not the only linear transformation that can be employed: it is effective because it allows the application of the canonization algorithm by comparing only a small number of values of the transformed function, and there is an efficient way to compute these values.

B. Variable Ordering Algorithm

Algorithm 4 gives the picture of a generalized canonizing algorithm in a high-level pseudo-code. For the sake of clarity, implementation-dependent optimization details (e.g.,

BDD cache optimizations) are not reported. The algorithm takes as input a Boolean function of n variables $f^b(x_{n-1}, x_{n-2}, \ldots, x_1, x_0)$, represented by an ordered sequence $W = (W_0, W_1, \ldots, W_{2^n-1})$ of $|W| = 2^n$ coefficients. The values of these coefficients depend on the representation chosen for the Boolean function. For instance, the Boolean function may be represented by the last column of its truth table.

The goal of the algorithm is to produce a list L of candidate canonizing permutations of the input variables. The permutation in L corresponding to the lexicographical maximum (see line 19 in Algorithm 4) is selected as the one that generates the canonical form of the assigned function f^b [1]. If more permutations lead to the canonical form, it is sufficient to identify any one canonizing permutation.

Algorithm 4: Generalized algorithm for the computation of the canonizing permutation.

Input: $V = \{0, \ldots, n-1\}$ as the set of input variables indexes, ($v = \{v_0, \ldots, v_{n-1}\}$);
$W = (W_0, \ldots, W_{2^n-1})$as coefficients corresponding to a n-variate Boolean function.
Output: List of candidate canonizing permutations, L.
Data: Let W^i be the coefficients of i-th order, $0 \leq i \leq n$, in W.

```
1  begin
2      C ← {V / ~S}
3      G ← Sort_{W^1}(C / =_{W^1})
4      L ← {G}
5      foreach i ∈ [0, |C|) do
6          foreach G ∈ L do
7              if |g_i| ≠ 1, g_i ∈ G then
8                  L ← L \ G
9                  foreach C_m ∈ g_j do
10                     G^(m) ← G \ g_j ∪ {C_m} ∪ g_j \ C_m
11                     L ← L ∪ G^(m)
12         /* |g_i| = 1, ∀g_i ∈ G ∧ ∀G ∈ L */
13         foreach G ∈ L do
14             L ← L \ G
15             foreach g_k ∈ G : k > i do
16                 G_k ← Sort_{W^2_{g_i}}(g_k / =_{W^2_{g_i}})
17                 G ← G \ g_k ∪ G_k
18             L ← L ∪ G
19     return max(L)
20 end
```

Every coefficient composing the representation of the function can be uniquely associated to one element of the input variables power-set. Let $v = \{x_{n-1}, x_{n-2}, \ldots, x_1, x_0\}$ be the set of input variables and $V = \{0, 1, \ldots, n-1\}$ be the set of input variables indexes. Given $D = \{0, 1, \ldots, 2^n - 1\}$ as the set of indexes corresponding to the positions of coefficients W, the following relation ψ defines a bijective map between the power-set of V, $\mathcal{P}(V)$, and the set D:

$$\psi : \mathcal{P}(V) \rightarrow D$$

$$\psi(I) = \begin{cases} \sum_{x \in I} 2^x & \text{if } I \in \mathcal{P}(V) \setminus \emptyset \\ 0 & \text{if } I = \emptyset \end{cases}$$

The inverse of the ψ relation can also be expressed as:

$$\varphi : D \rightarrow \mathcal{P}(V)$$

$$\varphi(m) = \begin{cases} \{0 \leq i < \lceil \log_2 n \rceil \mid [m/2^i] \text{ is odd } \} & \text{if } m \neq 0 \\ \emptyset & \text{if } m = 0 \end{cases}$$

Using the positional indexes in D, it is possible to define an equivalence relation amid coefficients in W as follows:

$$\forall\, i, j \in D, \quad W_i \sim W_j \Leftrightarrow \mid \varphi(i) \mid = \mid \varphi(j) \mid$$

where W_i and W_j are a pair of coefficients in W.

If $W_i \sim W_j$ we say that W_i and W_j have the same order $k = \mid \varphi(i) \mid = \mid \varphi(j) \mid$.

This way, we can partition the set of coefficients W into a series of disjoint sets $W^0 \ldots W^n$. We denote the generic element W^k as the set of coefficients of order k, where $k \in [0, n]$ and $\mid W^k \mid = \binom{n}{k}$.

A total ordering relation $\prec$ amid the coefficients in W is naturally inferred from the ordering of the binary n-uples identified with the elements of $\mathcal{P}(V)$ by the φ and ψ maps. In each binary tuple an element is 1 if the corresponding variable in v is included in the selected set of $\mathcal{P}(V)$, 0 otherwise. Thus, $\prec$ defines the dyadic ordering of the coefficients W. In the following, representations of the Boolean function f^b are assumed to be sorted according to $\prec$.

Now, let us introduce the notion of *symmetry equivalence relation* $\overset{S}{\sim}$ on the set of input variables v. Given an ordered set of coefficients $W = (W_0, \ldots, W_{2^n-1})$ representing the values of a Boolean function, any two input variables v_i and v_j, with $i, j \in [0, n-1]$ and $i \neq j$, are considered *equivalent* if and only if the sequence of coefficients values resulting after the exchange of the variables v_i and v_j is indistinguishable from the original one.

This notion of symmetry equivalence is consistent with the one adopted in [1], although other types of symmetries can be defined. The validity of Algorithm 4 is orthogonal to the problem of symmetry detection in the sense that different assumptions about the notion of symmetry can reduce the number of equivalence classes providing possibly a performance improvement.

Given $k = \mid V / \overset{S}{\sim} \mid$ as the number of cosets inferred by the relation $\overset{S}{\sim}$, the set of input variables can be thought as the disjoint union of the corresponding symmetric equivalence classes:

$$V \leftarrow \bigcup_{j=0}^{k-1} C_j$$

where $C_j = \{v_i \in v : v_i \overset{S}{\sim} c_j, 0 \leq i < n\}$, $0 \leq j < k$ being $c_j \in C_j$ the representative element of the generic equivalence class.

Now, the application of a transposition $\tau = \binom{0 \ldots h \ldots k \ldots n-1}{0 \ldots k \ldots h \ldots n-1}$ on the input variables of the Boolean function f^b corresponds to the operation of swapping all the pairs of coefficients W_i and W_j $\forall\, i, j$ with $i < j$ such that:

$$\varphi(i) \setminus \{k, h\} = \varphi(j) \setminus \{k, h\} \wedge$$
$$(k \in \phi(i) \wedge h \in \varphi(j)) \vee (k \in \varphi(j) \wedge h \in \varphi(i))$$

In general, a permutation σ can be decomposed into a functional product of transpositions $\sigma = \tau_0 \circ \tau_1 \ldots \circ \tau_t$, so the above correspondence can be naturally extended from transpositions to permutations.

Algorithm 4 classifies the set of input variables indexes V into a set $\mathcal{C}$ of $\overset{S}{\sim}$-equivalence classes. Subsequently, it collects the classes of $\mathcal{C}$ that have the same first order coefficient values (W^1) into groups g_j; the set of these groups is denoted as $\mathcal{G}$. More specifically, the generic group $g_j = \{C_{i_1}, C_{i_2}, \ldots, C_{i_t}\} \in \mathcal{C} / =_{W^1}$ includes all equivalence classes such that $W_{\psi(\{c_{i_1}\})} = \ldots = W_{\psi(\{c_{i_t}\})}$. The set $\mathcal{G}$ is then sorted according to values of W^1, making it an ordered sequence.

Initially, there is a single $\mathcal{G}$ in L, possibly containing groups g_i such that $|g_i| > 1$. The canonizing method proceeds examining the first group g_0 in this $\mathcal{G}$.

If g_0, is composed of a single $\overset{S}{\sim}$-equivalence class, i.e. $g_0 = \{C_{i_t}\}$, we consider such group *resolved*, which means that, with respect to the final canonizing permutation, a position for the corresponding input variables has been detected.

The second order coefficients $W^2_{g_0} \subset W^2$, with

$$W^2_{g_0} = \{\, W_k \in W^2 \,:\, \forall\, C_{i_t} \in g_0,\; C_{i_t} \subseteq \varphi(k),\, |\varphi(k)| = |C_{i_t}| + 1 \,\}$$

will then be used to try and order the subsequent (possibly *unresolved*) groups g_k, $k > 0$, as shown in lines 13÷18 of Algorithm 4.

In next iterations of the external loop (line 5), $g_1, \ldots g_{j-1}$ are considered, all such that $|g_i| = 1$. Thus, remaining unresolved groups may be ordered using second order coefficients.

On the other hand, when the considered group g_j is composed of more than one $\overset{S}{\sim}$-equivalence class C_{i_t}, the algorithm must exhaustively try to split g_j into a resolved group containing a single C_{i_t}, and a second, possibly unresolved, group $g_j \setminus \{C_{i_t}\}$. Lines 6÷11 of Algorithm 4 show that new possible solutions $\mathcal{G}$ are added to L in this case.

The algorithm will then try to resolve $g_j \setminus \{C_i\}$ using the second order coefficients $W^2_{\{C_i\}}$, as it did when the group was composed by a single symmetry equivalence class (lines 13÷18).

From the multiple solutions generated, we choose the single canonical form using higher order coefficients (line 19). Since these are complete solutions, we simply compare higher order coefficients in dyadic order, which amounts to selecting the lexicographical maximum among the solutions.

Example 3.1: Let us consider the following Boolean function: $f^b = (x_0 \oplus x_1)(x_0\overline{x}_1 + x_2)$, represented by the last column of the associated truth table $W = (0,\, 1,\, 0,\, 0,\, 0,\, 1,\, 1,\, 0)$. For this function, $V = \{0,\, 1,\, 2\}$, $n = |V| = 3$, $\mathcal{P}(V) = \{\emptyset,\, \{0\},\, \{1\},\, \{0,1\},\, \{2\},\, \{0,2\},\, \{1,2\},\, V\}$.

This can be easily seen by considering the entire truth table of f^b:

$x_2x_1x_0$	f^b
0 0 0	0
0 0 1	1
0 1 0	0
0 1 1	0
1 0 0	0
1 0 1	1
1 1 0	1
1 1 1	0

The computation of the first and second order coefficients assigns $W^1 = \{W_1, W_2, W_4\} = \{1,\, 0,\, 0\}$ and $W^2 = \{W_3,\, W_5,\, W_6\} = \{0,\, 1,\, 1\}$, respectively. The n proper $\overset{S}{\sim}$-equivalence classes (i.e., excluding the empty set $\emptyset$) are:

$$C = \{C_0 = \{0\},\, C_1 = \{1\},\, C_2 = \{2\}\}$$

The first collection of equivalence classes is computed by considering the first order coefficients W^1:

$$\mathcal{G} = \{g_0 = \{C_0\},\, g_1 = \{C_1,\, C_2\}\}$$

Sorting the variables with respect to the first order coefficients, the algorithm is able to discover that, in the canonical permutation, variable x_0 precedes variables x_1 and x_2, because $W_1 > W_2$ and $W_2 = W_4$.

Since the first group $g_0 = \{C_0\}$ is constituted by a single $\overset{S}{\sim}$-equivalence class, it is resolved, and the second order coefficients $W^2_{g_0} = \{W_3, W_5\}$ can be used to try to solve group $g_1 = \{C_1,\, C_2\}$. In our case, $W_3 < W_5$, so within g_1 class C_2 can be sorted before class C_1. The canonical permutation for f^b is therefore $\pi = \binom{0\,1\,2}{0\,2\,1}$.

C. Linear Transformations compatible with the Canonization Algorithm

The coefficients derived from the cofactor transformation employed in [1] are not the only possible choice: other works have employed, e.g., the Walsh coefficients instead of the cofactors [3]. To minimize the canonization time, it is important to be able to select the most performant transformation. To this end, we first define a set of compatible linear transformations that can be employed in conjunction with Algorithm 4. Then, we explore the set of compatible linear transformations and identify a specific transformation that is strictly more powerful than the cofactor form employed in [1].

Consider a generic multivariate Boolean function $f^b(x_{n-1}, \ldots, x_0)$ and a linear operator ρ with its associated matrix R. Let us consider all those ρ that are invertible and commutative with respect to any permutation π of the input variables of f^b. The idea is that, in general, it can be easier to maximum $f^b \circ \rho$ than f^b. To do so, we must show that finding the lexicographical maximum of $f^b \circ \rho$ leads to a canonical form $C(f^b)$, i.e. a uniquely identified representative element of the P-equivalence class of f^b.

To this end, for some class of transformations ρ, we need to prove that the following equality holds:

$$f^b \circ \rho \circ \pi = f^b \circ \pi \circ \rho \quad \forall\, \pi \in P$$

Definition 3.1: A *dyadic transposition matrix* Σ is a transposition matrix such that its application to the vector of values assumed by a Boolean function is equivalent to the application of a transposition π to the input variables of the same function.

Given any Boolean function, sketched by its truth table, with the minterms in the usual dyadic order, if we transpose variables v_i and v_j, the last column of the truth table changes according to the right product by a Σ matrix.

By construction, Σ keeps into account the sequential dyadic ordering of the truth table. As a consequence, Σ is a bisymmetric permutation matrix, because each of its columns contains a single one and globally the following equality chain holds: $\Sigma = \Sigma^T = \Sigma^{-1} = J\Sigma J$, where J is the exchange matrix ($J^2 = I$).

Theorem 3.1: Consider a Boolean function f^b represented as the vector of its values in the dyadic order of its domain. Let R be the matrix associated with a linear transformation ρ of the Boolean function f^b, and Σ be a dyadic transposition matrix. A sufficient condition for the equality ${f^b}^T R\Sigma = {f^b}^T \Sigma R$ to hold is that R is *definite positive* and either symmetric or persymmetric.

Proof:

$$ {f^b}^T R\Sigma = {f^b}^T \Sigma R \Leftrightarrow R\Sigma = \Sigma R $$

If R is persymmetric, then $R = JR^TJ$. Let us consider the following derivation:

$$ R\Sigma = JR^TJJ\Sigma J = JR^T\Sigma J $$

$$ \Sigma R = J\Sigma JJR^TJ = J\Sigma R^TJ $$

$$ R\Sigma = \Sigma R \Leftrightarrow JR^T\Sigma J = J\Sigma R^TJ \Leftrightarrow R^T\Sigma = \Sigma R^T $$

Therefore, $R\Sigma = \Sigma R$ is equivalent to:

$$ (R + R^T)\Sigma = \Sigma(R + R^T) $$

where $R + R^T$ is bisymmetric by construction. So, in the rest of the proof, we will assume that R is a symmetric matrix, without prejudice for the generality of our proof.

Due to the well-known theorem of simultaneous diagonalization [13], being the two matrices R and Σ both symmetric, and R being definite positive (that is, its eigenvalues are all positive), there exists a common orthogonal diagonalization matrix Q ($Q^T = Q^{-1}$), such that $Q^TRQ = \Lambda_R$ and $Q^T\Sigma Q = \Lambda_\Sigma$.

Then, since diagonal matrices always commute, the following equalities hold:

$$ R\Sigma = Q\Lambda_R Q^T Q\Lambda_\Sigma Q^T = Q\Lambda_R\Lambda_\Sigma Q^T = $$

$$ = Q\Lambda_\Sigma\Lambda_R Q^T = Q\Lambda_\Sigma Q^T Q\Lambda_R Q^T = \Sigma R $$

■

Corollary 3.1: Let Σ' be a permutation matrix corresponding to the subsequent application of several dyadic transpositions $\Sigma' = \Sigma_1 \cdot \Sigma_2 \ldots \Sigma_m$. Then, a linear transformation matrix R and Σ' commute.

Proof:

$$ R\Sigma' = R\left(\Sigma_1 \ldots \Sigma_m\right) = \Sigma_1 R \ldots \Sigma_m = \left(\Sigma_1 \ldots \Sigma_m\right) R = \Sigma' R $$

■

The most popular transformations used in canonical form Boolean matching are the Walsh and the cofactor transformations. The Walsh transform [3] is defined by means of the Hadamard matrix. The coefficients of the cofactor representation [1], on the other hand, are computed as the onset cardinalities of the cofactors of the Boolean function.

The Hadamard matrix H_n of rank $n = 2^k, k \in \mathbb{N}^+$ can be expressed as:

$$ H_1 = [1] $$

$$ H_{2n} = \begin{bmatrix} H_n & H_n \\ H_n & -H_n \end{bmatrix} $$

We observe that the computation of the cofactor onset sizes $|f_I|$, where I ranges in the power-set of the set of input variables, can be performed by means of a matrix C_n defined much like the Hadamard matrix:

$$ C_1 = [1] $$

$$ C_{2n} = \begin{bmatrix} C_n & 0 \\ C_n & C_n \end{bmatrix} $$

Since the matrix C_n has all the eigenvalues equal to 1, Theorem 3.1 can be applied. Thus, we can conclude that the cofactor representation is suitable for canonical form computation.

As a major result of this work Theorem 3.1 defines an entire class of transformations that can be employed in canonical form Boolean matching using the same approach as that used with the cofactor representation.

Note that in the case of the Hadamard matrix, we cannot directly apply Theorem 3.1. However, the Walsh transformation is actually obtained by a Walsh matrix, which is the Hadamard matrix with its lines reordered by applying bit reversal and Gray code sequential ordering [3]. Indeed, the resulting Walsh matrix W is symmetric, and W^2 is diagonal (specifically, $W^2 = nI$), and we can prove that the Walsh matrix commutes with dyadic trasponsition matrices.

Theorem 3.2: Let $W_{2^n \times 2^n},\ n \geq 1$ be a Walsh matrix, and Σ be a compatible dyadic transposition matrix. Then

$$ W\Sigma = \Sigma W $$

Proof: Proving that $W\Sigma = \Sigma W$ is equivalent to proving that $(W\Sigma)(\Sigma W) = (\Sigma W)(W\Sigma)$. The following derivation can be used to this end:

$$ \Sigma W W\Sigma = \Sigma nI\Sigma = nI = WW = W\Sigma\Sigma W $$

since $\Sigma^2 = I$. ■

The statement of Corollary 3.1 holds for the Walsh matrix as well, proving that it commutes with generic dyadic permutation matrices. Theorem 3.1 and Theorem 3.2 define two sets of acceptable linear transformations compatible with the generalized variable ordering algorithm described in Section III-B.

D. Generalization to Alternative Approaches

In this Section, we survey the approaches to the exploration of the permutations tree presented in [7], [11]. We show that, when the same representation of Boolean functions is employed, such algorithms bring to the same canonical forms

as Algorithm 4, and can benefit from representations that allow performace improvements in that algorithm.

In [7] the authors define a canonical form based upon a cost function associated with the onset representation of a function. Given an n-variate Boolean function f, represented by its k minterms $M = \{m_0, ...m_{k-1}\}$, the canonical form is constructed by minimizing a cost function defined as the integer number corresponding to the classical binary encoding of the string obtained by juxtaposing the minterms resulting from a permutation of the input variables. The authors, after defining the canonical form, provide an effective branch and bound procedure to compute it.

However, it is possible to show that the canonical form defined in [7] is the same as the one produced by Algorithm 4 when applied to a function described by means of its truth table.

The cost function is affected by both the juxtaposition order of the minterms and the permutation of variables. However, given a permutation of the input variables, there is a single juxtaposition order of the minterms that minimizes the cost function. It is immediate to see that the minimizing sequence of minterms is the increasing lexicographical order of function output values. An input variables permutation, on the other hand, changes the onset elements.

Let us consider the truth table of f, where the rows are in dyadic order. For a 3-variate function, the truth table is given by the sequence: $f(000)$, $f(001)$, $f(010)$, $f(011)$, $f(100)$, $f(101)$, $f(110)$, $f(111)$.

The generalized canonical form computation defined by Algorithm 4, applied to f, lexicographically maximizes the string of function outputs. Intuitively, this *shifts* the 1s of the Boolean function towards the minterms that have a lower contribution to the cost function of [7]. Thus, it reasonable that the two operations (lexicographical maximization and cost minimization) lead to the same result.

For a more formal proof, consider a function f, represented by its onset $M = \{m_0, ...m_{k-1}\}$. Its minimized cost function is obtained by juxtaposing the minterms starting from the lexicographically smallest, in ascending order. Any other sequence would correspond to a greater cost.

Let $c(f \circ \pi')$ be the cost function associated with the minimizing permutation π' of the input variables according to [7].

Let us now consider the lexicographical maximum $f \circ \pi''$ of function f obtained from Algorithm 4, along with the canonicizing permutation π''. We want to show that $f \circ \pi'$ and $f \circ \pi''$ are the same.

Assume $f \circ \pi' \neq f \circ \pi''$ and consider the first output value for which $f \circ \pi'$ and $f \circ \pi''$ differ. If this value is 1 for $f \circ \pi'$ and 0 for $f \circ \pi''$, then π'' is not the permutation that generates the canonical form according to Algorithm 4 – since $f \circ \pi'$ is lexicographically greater. On the other hand, if the value is 0 for $f \circ \pi'$ and 1 for $f \circ \pi''$, then π' is not the permutation that generates the canonical form according to [7]. Since we are considering the first difference between $f \circ \pi'$ and $f \circ \pi''$, there is a common prefix to their cost functions $c(f \circ \pi')$ and $c(f \circ \pi'')$. The first minterm of $f \circ \pi''$ after the common prefix corresponds to the first difference in the output values, while that of $f \circ \pi'$ corresponds to a later output value. The minterm in $c(f \circ \pi'')$ is lexicographically smaller than the corresponding one in $c(f \circ \pi')$, as the output values are considered in dyadic order. Thus, $c(f \circ \pi'') < c(f \circ \pi')$ and π' does not generate the canonical form of [7].

To give a better intuition of the reasoning, let us introduce the following example.

Example 3.2: Consider the Boolean function $f^b = (x_0 \oplus x_1)(x_0\overline{x}_1 + x_2)$ from Example 3.1 and its minterm representation:

$$M = \{m_1, m_5, m_6\} = \{001, 101, 110\}$$

According to the algorithm in [7], we first choose m_1, without any variable permutation (it is already minimal). However, we just learnt that x_0 will not be permuted at all, as the choice of m_1 implies a partition of the minterms in two subsets, and no permutations will happen between variables of different subsets, as such a permutation would replace m_1 with a different minterm of higher cost. The rightmost subset of m_1 only contains x_0, so that its position is fixed in the canonizing permutation. Then, we consider m_5, and replace it with m_3, i.e. we permute x_2 and x_1. This has no effect on the other minterms – the algorithm guarantees it will have no effect on minterms already sorted, and m_6 has the same bits for x_2 and x_1.

Thus, the canonicizing permutation is $\pi = \binom{0\,1\,2}{0\,2\,1}$, just as obtained using Algorithm 4.

Conversely, the same reasoning can be applied to the two algorithms when a different representation, e.g., the one obtained through the cofactor transformation, is used. To this end, it is sufficient to replace lexicographic ordering of strings on the alphabet $\{0, 1\}$ with a total ordering relation on the set of numerical coefficients.

A previous approach using a similar exploration of the permutations tree is reported in [11]. In this case, the canonical form is explicitly defined as the lexicographic maximum of the bit strings representing the truth table. The canonizing permutation is constructed by exploring the permutations tree and pruning it when a lexicographic non-maximum prefix is found.

An important conclusion is that a different representation than the truth table could also be applied to the algorithms proposed in [7], [11]. The truth table is not especially efficient: intuitively, if at a given depth the permutation tree has been pruned so that all the prefixes of surviving branches are equal, then moving to the next level will only allow to find two sets of subtrees with different prefixes.

We will show in the following section how to obtain efficient representations, and measure their effectiveness at reducing the cost of computing the canonizing permutation.

E. The Shifted Cofactor Transformation

Given the definition of a family of acceptable linear transformations, we now define one specific transformation that allows us to achieve better performance in computing the canonical form than the previous works, the *shifted cofactor* representation.

We start from the cofactor representation, which proved very effective [1]. As shown previously, the canonical form computation algorithm works better if the representation of the function $f \circ \rho$ has a smaller number of equal values.

Given a function $f(x_{n-1}, \ldots, x_0)$ and an assigment to its variables $\overline{X} = (\overline{x}_{n-1}, \ldots, \overline{x}_0), \overline{x}_k \in \{0, 1\}$, the corresponding element of the cofactor representation is obtained as follows. Consider the set I of input variables that are assigned a value of 1 in $\overline{X}$, $I = \{x_k \in \{x_{n-1}, \ldots, x_0\} | \overline{x}_k = 1\}$, then summing all the output values of the cofactor of f with respect to I, f_I:

$$(f \circ \rho_c)(\overline{X}) = \sum_{m=0}^{2^{n-|I|}-1} f_I(m) \tag{1}$$

Note that the output values of f_I contribute to $(f \circ \rho_c)(\overline{X})$ with a value of 0 or 1.

Let us consider a second representation of function f, $f \circ \rho'$ that satisfies Theorem 3.1. If the condition

$$(f \circ \rho')(\overline{X}) = (f \circ \rho')(\overline{Y}) \Rightarrow (f \circ \rho_c)(\overline{X}) = (f \circ \rho_c)(\overline{Y}) \tag{2}$$

holds for all pairs of assignment tuples $\overline{X}$ and $\overline{Y}$ corresponding to sets of input variables I and J with the same cardinality ($|I| = |J|$), then computing the canonical form from $f \circ \rho_c$ will take at least as many operations as from $f \circ \rho'$.

The condition in Equation 2 is not needed for all pairs $\overline{X}$ and $\overline{Y}$ such that $|I| \neq |J|$, because in that case $(f \circ \rho')(\overline{X})$ and $(f \circ \rho')(\overline{Y})$ are coefficients of different order, and thus are never compared in Algorithm 4.

Thus, a transformation ρ' that satisfies Equation 2 has a chance to improve canonization performance. Equation 2 and Theorem 3.1 limit the space of transformations ρ that can be employed in canonization. One important consideration is that the truth table does not satisfy the condition of Equation 2, and therefore it is not a good representation for use in Boolean matching algorithms. Algorithms such as those in [7], [11] could, on the other hand, effectively employ the cofactor transformation to more aggressively prune the permutations tree.

We now consider the addition of a weight $w_m \in \mathbb{N}^+$ to each addendum in Equation 1 to define another transformation ρ_{cw}:

$$(f \circ \rho_{cw})(\overline{X}) = \sum_{m=0}^{2^{n-|I|}-1} w_m f_I(m) \tag{3}$$

The selection of values of w_m in such a way that $\rho' = \rho_{cw}$ satisfies Theorem 3.1 and Equation 2, can be obtained by setting $w_m = a^{H(m)}$, where $H(m)$ is the Hamming weight of the binary expansion of m (referred to the cofactor f_I), and $a \in \mathbb{N}$ is chosen so that in any $(f \circ \rho_{cw})(\overline{X})$ sum, the binary representations of the partial sums of operands with equal Hamming weights do not overlap (i.e., the bitwise AND of their binary representations is 0).

This condition imposes that two values of $\rho_{cw} \circ f$ with equal order be computed using the same set of weights applied to different values of $f_I(m)$.

Let us consider the computation of a generic coefficient $(f \circ \rho_{cw})(\overline{X})$, and focus on the partial sum of terms with equal order:

$$(f \circ \rho_{cw})(\overline{X}) = \sum_{k=|I|}^{n} S^k \tag{4}$$

where

$$S^k = \sum_{\substack{0 \leq j < 2^{n-|I|} \\ H(j)=k}} w_k f_I(j) \tag{5}$$

Since the binary representation of each S^k does not overlap with others, the computation of Equation 4 does not involve any carry bit, which means that each addendum brings an independent contribution to the sum, and there is no loss of information in substituting $\{ S^k \,|\, \forall k \in [|I|, n] \}$ with their sum $(f \circ \rho_{cw})(\overline{X})$.

Thus, a violation of Equation 2 cannot depend on a carry bit that modifies a term S^k. Such violation must then correspond to a partial sum S^k that is different for $(f \circ \rho_c)(\overline{X})$ and $(f \circ \rho_c)(\overline{Y})$ but not for $(f \circ \rho_{cw})(\overline{X})$ and $(f \circ \rho_{cw})(\overline{Y})$.

Let us now consider the computation of the partial sum S^k shown in Equation 5. Since $f_I(j)$ terms of equal order are multiplied by the same weight w_k, a different number of non-zero $f_I(j)$ terms necessarily leads to a different value of the S^k, thus avoiding any loss of information. We can conclude that Equation 2 holds under the hypotheses.

However, a good representation must not only be able to effectively distinguish variable orders in the canonization algorithm – it must also be possible to efficiently compute its coefficients. We found that $w_m = 2^{H(m)}$ leads to an efficient implementation while introducing only limited aliasing, thus ensuring a good performance improvement over the cofactor representation, as will be shown in Section IV. We call this $f \circ \rho_{cw}$ the *shifted cofactor* representation of f. We will show in Section III-G how the shifted cofactor representation can be computed efficiently.

The formal verification of the applicability of Theorem 3.1 to the shifted cofactor transformation can be performed more easily if its definition is restated in matrix form. A careful analysis of the definition of the shifted cofactor transformation leads to the following relation with the cofactor matrix defined in Section III-C:

$$C_n^w = (C_n)^2$$

The proof proceeds by induction. The induction base is:

$$C_1 = C_1^2 = [1], C_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}, C_2^2 = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}$$

The induction step can be expressed as:

$$C_{n+1}^2 = \begin{bmatrix} C_n & 0 \\ C_n & C_n \end{bmatrix}^2 = \begin{bmatrix} C_n^2 & 0 \\ 2C_n^2 & C_n^2 \end{bmatrix}$$

where C_{n+1}^2 is persymmetric and block-triangular. Recalling that the eigenvalues of a triangular matrix can be read on its main diagonal, we can infer that C_{n+1}^2 has all positive eigenvalues if C_n^2 has the same property. Since the eigenvalue of C_1^2 is unitary, so are all the eigenvalues of C_n^2, for all n, and therefore C_n^w is definite positive, fulfilling all the conditions of Theorem 3.1.

TABLE I

TRUTH TABLE, COFACTOR AND SHIFTED COFACTOR REPRESENTATION OF THE 2:1 MULTIPLEXER $f(x_2, x_1, x_0) = x_0\overline{x_2} + x_1x_2$

$x_2x_1x_0$	f^b	$f \circ \rho_c$	$f \circ \rho_{cw}$
0 0 0	0	4	18
0 0 1	1	3	7
0 1 0	0	3	8
0 1 1	1	2	3
1 0 0	0	2	6
1 0 1	0	1	2
1 1 0	1	2	3
1 1 1	1	1	1

F. Canonization Example

In this section we exemplify the computation of some of the canonical forms previously presented. In particular, we derive the canonical representation of a three input multiplexer with respect to truth table, cofactor and shifted cofactor representation.

Let us consider a multiplexer with two data inputs x_0 and x_1 and one control input x_2, $f(x_2, x_1, x_0) = x_0\overline{x_2} + x_1x_2$.

1) Truth Table: The truth table of f is shown in column f^b of Table I. Algorithm 4, applied to f, first considers the values of the function for the input configurations (001), (010), (100) (associated with variables x_0, x_1, x_2, respectively). Since the value associated with x_0 is 1, while those associated with x_1 and x_2 are 0, the first-order coefficients suffice to draw a partial ordering: $\{x_0\}, \{x_1, x_2\}$, that is x_0 will be the first variable in the canonical permutation. To order x_1 and x_2, it is necessary to examine the second-order values associated with x_0x_1 and x_0x_2: $f(011) = 1$ and $f(101) = 0$. Since the value associated with $x_0\,x_1$ is greater than that of $x_0\,x_2$, x_1 preceeds x_2 in the canonical ordering (that is the permutation of input variables corresponding to the canonical form). The ordering is then x_0, x_1, x_2. In total, five values of the function have been employed to identify the canonical ordering.

2) Cofactors: Table I reports the values of the cofactors and shifted cofactors for the 2:1 multiplexer, sorted in dyadic binary order. E.g., the entry relative to the 001 configuration reports the cardinality of the onset of cofactor f_{x_0}. Conventionally, the first entry, 000, corresponds to the cardinality of the function onset, $|f|$.

First order cofactors associated with the input variables are: $|f_{x_0}| = 3$, $|f_{x_1}| = 3$, $|f_{x_2}| = 2$. Then, x_2 is the last variable and it is necessary to compute the second-order coefficients to discriminate between x_0 and x_1. Since $|f_{x_0x_2}| = 1$, $|f_{x_1x_2}| = 2$, x_0 comes after x_1, and the canonical ordering is: x_1, x_0, x_2. Five coefficients are employed to identify the canonical ordering.

3) Shifted Cofactors: First order shifted cofactors associated with the input variables are: $c_{x_0} = 7$, $c_{x_1} = 8$, $c_{x_2} = 6$, as shown in Table I. For example, for c_{x_0}, we have that $X = \{x_0\}$, $f_X = x_1 + \overline{x_2}$, so the minterms in its onset correspond to the variable assignments in the original function (001), (011) and (111). Therefore, $c_{x_0} = 2^{(1-1)} + 2^{(2-1)} + 2^{(3-1)} = 7$. The variable ordering of the canonical representation is then established by exploiting only the three first-order coefficients.

G. Efficient Computation of the Shifted Cofactor Transformation

In this Section, we tackle the issue of efficiently implementing the computation of the shifted cofactor transformation, when the Boolean functions are represented as BDDs, which is the most common case in modern synthesis tools.

Given a ROBDD for a Boolean function f rooted in node N, with a specified variable ordering $\overline{v} = (v_1, \ldots, v_n)$, the cardinality of the onset of f can be efficiently computed through the following induction rule.

In the base case, the Boolean function consists of a single node. Then, the cardinality of its onset $| f |=| N |$, is $| N |= 1$ if N is the constant one node, $| N |= 0$ if N is the constant zero node. The induction step is

$$| N |=| N_T |\ 2^{|v(N_T)-v(N)|-1} + | N_E |\ 2^{|v(N_E)-v(N)|-1}$$

where N_T and N_E are the roots of the *then* and *else* parts of the BDD rooted in N, and $| v(N_1) - v(N_2) |$ is the distance between the two variables tested in N_1 and N_2, $v(N_1)$ and $v(N_2)$, in V.

The terms $2^{|v(N_T)-v(N)|-1}$ and $2^{|v(N_E)-v(N)|-1}$ take into account the fact that some variables may not appear in one of the branches of the tree, so that $v(N_E)$ and $v(N_T)$ may not be adjacent to $v(N)$ in the input variable ordering $\overline{v}$.

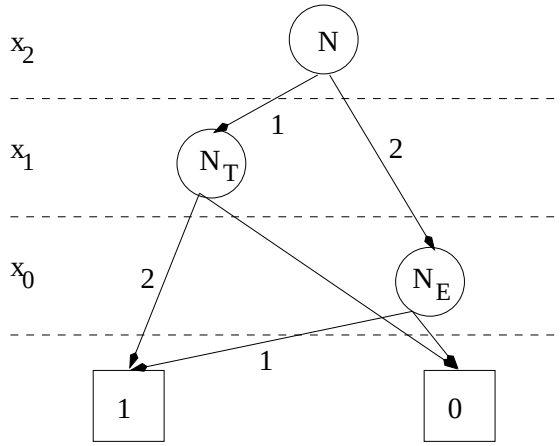

Fig. 1. ROBDD for the 2:1 multiplexer. Arcs are labeled with weights $2^{|v(N_1)-v(N_2)|-1}$, indicating the missing nodes for variables that are indifferent for a certain subtree.

For example, let us consider the computation of $| f |$, where f is the 2:1 multiplexer in Figure 1. Each arc (N_2, N_1) is labeled with the value of $2^{|v(N_1)-v(N_2)|-1}$. The value of $| f |$ is computed as $| N |=| N_T | +2 | N_E |= 2 + 2 \cdot 1 = 4$.

This induction rule, which is efficiently implemented in the *CUDD* package [16], can be extended to cover the shifted cofactor computation.

In the shifted cofactor transformation, the onset elements of the function are weighted by the Hamming weight of the corresponding input configuration, so in the BDD representation this computation rule can be translated by differently weighting the *then* and *else* subtrees. Specifically, a *then* arc between two nodes corresponding to adjacent variables has its weight doubled, while the *else* arc has a weight of one.

In the proof of the modified induction rule, the base case is unchanged, $\mid N \mid = 1$ if N is the constant one node, $\mid N \mid = 0$ if N is the constant zero node. The induction step is

$$\mid N \mid = 2 \mid N_T \mid 3^{|v(N_T)-v(N)|-1} + \mid N_E \mid 3^{|v(N_E)-v(N)|-1}$$

The new weight $3^{|v(N_1)-v(N_2)|-1}$ is given by the sum of the weights of a set of $\mid v(N_1) - v(N_2) \mid$ variables that do not affect the considered subtree:

$$\sum_{k=0}^{|v(N_1)-v(N_2)|-1} \binom{|v(N_1) - v(N_2)| - 1}{k} 2^k = 3^{|v(N_1)-v(N_2)|-1}$$

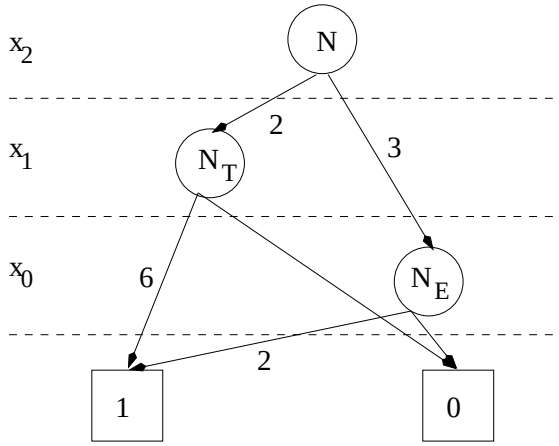

Fig. 2. ROBDD for the 2:1 multiplexer. Arcs are labeled with quantities that take into account the weights used in the computation of the shifted cofactor transformation.

The computation of the zero-order shifted cofactor is shown for the same 2:1 multiplexer example in Figure 2. The value $f \circ \rho_{cw}(000)$ is computed as $\mid N \mid = 2 \mid N_T \mid + 3 \mid N_E \mid = 2 \cdot 6 + 3 \cdot 2 = 18$.

IV. Experimental Results

In order to provide experimental evidence supporting the effectiveness of the *shifted cofactor* representation introduced in Section III, we have reimplemented the algorithm proposed in [1], and we then applied our representation within the same canonical form computation framework. The canonical form computation framework uses *binary decision diagrams* (BDD) as the most efficient representation format for boolean function manipulation, and relies on the *CUDD* BDD manipulation package [16].

The following results have been obtained using a Pentium4 processor at 3.20 GHz.

Figure 3 shows the overall results of our approach by comparing the average execution times for the algorithm using the cofactor representation [1] and the *shifted cofactor representation*, over a large set of randomly generated Boolean functions ranging from 3 to 24 input variables The shifted cofactors outperform the previous approach by five times.

To better understand the reasons of this success, let us consider the information provided by Figures 4 and 5. The former plots the number of solutions ($|L|$) produced by Algorithm 4 – and gives therefore a measure of the effectiveness of the two methods in identifying the canonizing transformation. The latter measures the usage of second order coefficients by Algorithm 4. Since both cofactor and shifted cofactor computations can be performed very efficiently on demand, the ability to reduce the number of cofactors used positively affects the overall performance.

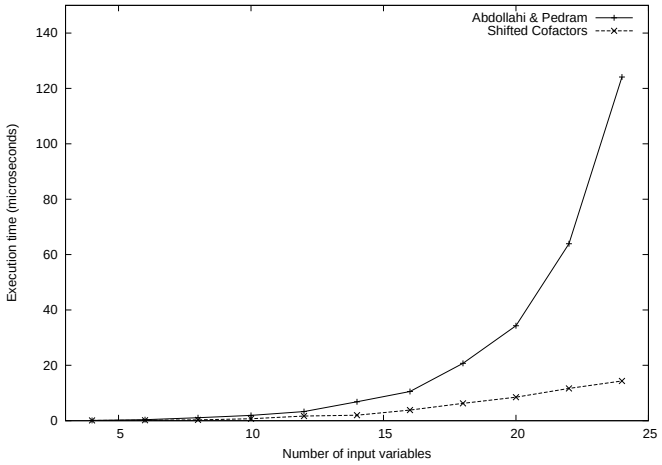

Fig. 3. Compared execution times as a function of the input set size

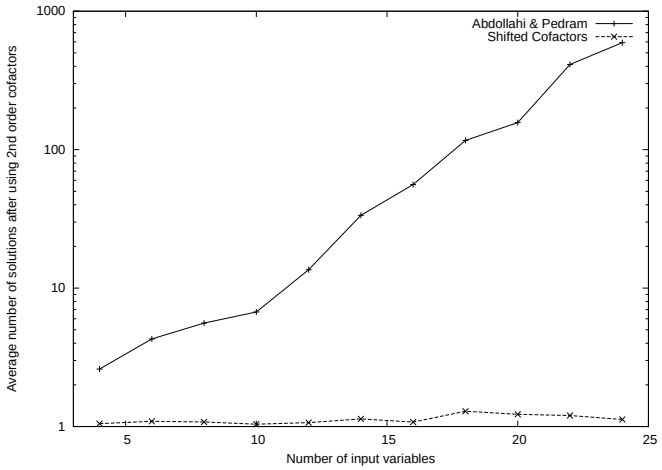

Fig. 4. Number of solutions produced by Algorithm 4 as a function of the number of input variables

This is not the case for Walsh coefficients, for example, because in that case the on-demand computation is slower than the Fast Walsh Transform methods that compute all the coefficients at once. This is the main reason why Walsh coefficients cannot be effectively employed to solve the Boolean matching problem. Indeed, the number of coefficients needed to compute the canonical form using the Walsh transform is not significantly different from that of the cofactor transform.

Note that, for both metrics considered in Figure 4 and Figure 5, the shifted cofactors technique outperforms the cofactor method by more than one order of magnitude.

In order to provide a better comparison with the previous work in [1], Table II also reports in the results for multiplexer functions, which were indicated as worst case within the test set of [1]. We report time metrics for multiplexers with 3, 4 and 5 selectors (11, 20 and 37 total input variables,

TABLE II
COMPARISON OF EXECUTION TIMES ON MULTIPLEXER BOOLEAN FUNCTIONS (TIMES IN MICROSECONDS)

Selectors	Cofactors	Shifted Cofactors
3	3.1 μs	3.1 μs
4	47.0 μs	18.8 μs
5	1337.7 μs	573.7 μs

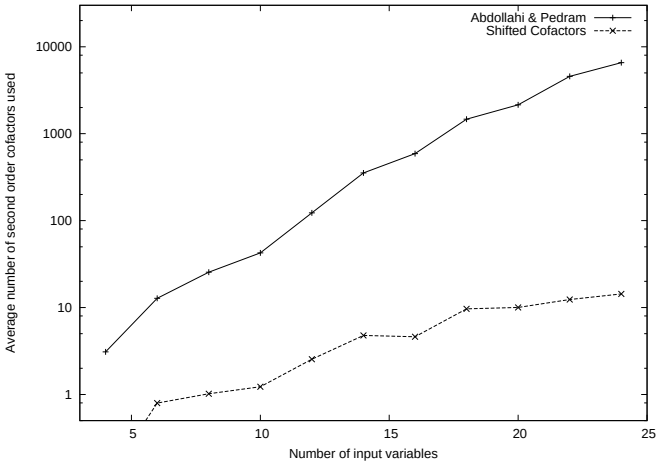

Fig. 5. Number of second order cofactors used by Algorithm 4 as a function of the number of input variables

respectively). All input multiplexers are represented using the variable ordering that ensures minimal BDDs. It can be seen that the shifted cofactors outperform the cofactors in the case of 20 and 37 input variables multiplexers, consistently with the results of the random functions test.

V. CONCLUDING REMARKS

In this paper, we have proposed a unified approach to boolean matching under P-equivalence based on canonical forms, building over existing spectral and cofactor-based techniques. From the theoretical insights obtained from the unification of previous approaches, we derived a new canonical form that significantly reduces computation times with respect to the state of the art.

Future directions include the extension of the family of linear transformations, the inclusion of NPN-canonical forms within the formal framework, and the extension to the problem of Boolean matching with *don't care* conditions.

REFERENCES

[1] Afshin Abdollahi and Massoud Pedram. Symmetry detection and boolean matching utilizing a signature-based canonical form of boolean functions. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(6):1128–1137, June 2008.

[2] Giovanni Agosta, Francesco Bruschi, Gerardo Pelosi, and Donatella Sciuto. A unified approach to canonical form-based Boolean matching. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 841–846, New York, NY, USA, 2007. ACM.

[3] Ken G. Beauchamp. *Applications of Walsh and Related Functions*. Academic Press, 1984.

[4] Luca Benini and Giovanni De Micheli. A survey of Boolean matching techniques for library binding. *ACM Trans. Design Autom. Electr. Syst.*, 2(3):193–226, 1997.

[5] Jerry R. Burch and David E. Long. Efficient Boolean function matching. In *ICCAD '92: Proceedings of the 1992 IEEE/ACM international conference on Computer-aided design*, pages 408–411, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[6] Donald Chai and Andreas Kuehlmann. Building a better Boolean matcher and symmetry detector. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 1079–1084, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[7] Jovanka Ciric and Carl Sechen. Efficient canonical form for Boolean matching of complex functions in large libraries. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 22(5):535–544, 2003.

[8] E. M. Clarke, K. L. McMillan, X Zhao, M. Fujita, and J. Yang. Spectral transforms for large boolean functions with applications to technology mapping. In *DAC '93: Proceedings of the 30th international conference on Design automation*, pages 54–60, New York, NY, USA, 1993. ACM.

[9] Debatosh Debnath and Tsutomu Sasao. Efficient computation of canonical form for Boolean matching in large libraries. In Masaharu Imai, editor, *ASP-DAC*, pages 591–596. IEEE, 2004.

[10] C. R. Edwards and S. L. Hurst. A Digital Synthesis Procedure Under Function Symmetries and Mapping Methods. *IEEE Trans. Comput.*, 27(11):985–997, 1978.

[11] Uwe Hinsberger and Reiner Kolla. Boolean matching for large libraries. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 206–211, New York, NY, USA, 1998. ACM.

[12] Victor N. Kravets and Karem A. Sakallah. Generalized symmetries in boolean functions. In *ICCAD '00: Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, pages 526–532, Piscataway, NJ, USA, 2000. IEEE Press.

[13] Serge Lang. *Linear Algebra*. Addison Wesley, 1966.

[14] D. M. Miller. A spectral method for boolean function matching. In *EDTC '96: Proceedings of the 1996 European conference on Design and Test*, page 602, Washington, DC, USA, 1996. IEEE Computer Society.

[15] Janett Mohnke, Paul Molitor, and Sharad Malik. Limits of using signatures for permutation independent Boolean comparison. In Isao Shirakawa, editor, *ASP-DAC*. ACM, 1995.

[16] Fabio Somenzi. CUDD: CU Decision Diagram Package. http://vlsi.colorado.edu/ fabio/CUDD/.

[17] Kuo-Hua Wang. Exploiting k-distance signature for boolean matching and g-symmetry detection. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 516–521, New York, NY, USA, 2006. ACM.