Fast Spectrum Computation for Logic Functions using Binary Decision Diagrams

Masahiro Fujita* Jerry Chih-Yuan Yang† Edmund M. Clarke‡ Xudong Zhao‡ Patrick McGeer§

*Fujitsu Laboratories of America,
†Center for Integrated System, Stanford University,
‡School of Computer Science, Carnegie-Mellon University,
§EECS, University of California at Berkeley.

ABSTRACT

We show very efficient methods to compute Walsh spectrum for logic functions with large numbers of inputs (30 or more) using Binary Decision Diagrams. The BDD structure is extended to have any integer values as leaf (constant) nodes. The result is an efficient representation for integer vectors and integer matrices. The proposed procedure works directly on an extended BDD for the logic function, and computes the *full* Walsh spectrum in the form of an extended BDD. The algorithm presented is a more efficient version of the matrix-multiplication method proposed in [2]. Our method embeds the Walsh transform matrix implicitly into program code with recursive calls, which results in a significant speed improvement. Our algorithm has the same complexity as the fastest known Walsh algorithm, and utilizes a much more efficient data structure than traditional truth tables. Furthermore, in cases where the complete set of spectra coefficient is either infeasible or impractical, we also present a method to compute *subsets* of Walsh coefficients. We present experimental results showing that logic functions having more than 60 inputs which cannot be processed by other published methods [4, 6, 3] can be computed within 30 seconds on Sparc 2.

INTRODUCTION

Walsh spectrum computation [4] is a very important tool to analyze Boolean functions. For example, by analyzing coefficients of Walsh spectra for Boolean functions, we can efficiently classify them. These classifications can then be effectively used in problems logic synthesis. One problem that spectra-based techniques have been used to solve is that of Boolean matching[2, 6]. The matching problem is to detect whether two given Boolean functions are equivalent under input permutation and input/output polarity changes.

We first review some definitions for the meaning of Walsh coefficients[4]:

Definition 1 *For a Boolean function f with n input variables, each of the 2^n coefficients of its Walsh spectrum shows the closeness between f and one of the following functions:*

- **zeroth order coefficient** *(1 element):* 0,
- **first order coefficients** *(n elements):* $x_1, x_2, \ldots, x_n$,
- **second order coefficients** *($\binom{n}{2}$ elements:)* $x_1 \oplus x_2,\ x_1 \oplus x_3,\ \ldots,\ x_{n-1} \oplus x_n,\ \ldots,$
- **n^{th} order coeffient** *(1 element):* $x_1 \oplus x_2 \oplus, \ldots, \oplus x_n$

The symbol $\oplus$ denotes exclusive-OR operation. So, Walsh spectrum gives important information for exclusive-OR analysis of Boolean functions. It is therefore also used for the exclusive-OR based logic synthesis[5, 4].

Traditional approaches to compute Walsh spectrum is based on truth tables. The most effective truth table-based algorithm is the *fast Walsh transform* [4] which is similar to the butterfly algorithm for fast Fourier transforms. Clearly the main disadvantage for truth table-based techniques is that they cannot handle logic functions with large numbers of input variables.

Recently, a method to compute Walsh spectrum directly from sum-of-products representation has been proposed [3]. However, many practical logic functions cannot be represented in sum-of-products forms, because numbers of products can be too large.

One way to overcome the above difficulty is to use BDD to compute Walsh spectrum, since BDDs have been an effective data structure for Boolen function manipulation. If we extend BDD proposed by Bryant [1] by allowing any integer values as constant nodes, we can represent integer vectors and matrices in BDD. Consequently, we can perform multiplication between vectors and matrices using BDD operations as shown in [2]. In [2], experimental results on the computation of Walsh spectrums for logic functions with large numbers of inputs are reported. These spectra cannot be computed by other published methods, because either the number of inputs is too large for truth table representation, or the sum-of-product forms have too many product terms.

In this paper we improve the method proposed in [2] by embedding the Walsh matrix into program codes in a recursive way. As shown later, this gives a significant speed-up over the method in [2]. In fact, the algorithm has the same complexity as the fastest known Walsh transform. Our algorithm has the additional advantage of using BDDs, which offer significant storage savings. Our results show that Walsh spectra for logic functions with well over 60 variables can be computed within 30 seconds on Sparc 2.

For some logic functions it is not possible to compute the full set of Walsh coefficients, even by using BDDs. This is due to the large number of leaf (constant) nodes. In these cases, it is often desirable to compute a *partial*

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \\ 1 \\ -1 \\ -1 \\ -1 \end{bmatrix} = \begin{bmatrix} -2 \\ 6 \\ 2 \\ 2 \\ 2 \\ 2 \\ -2 \\ -2 \end{bmatrix}$$

Figure 1: Computation of Walsh spectrum for $x_1x_2+x_3$

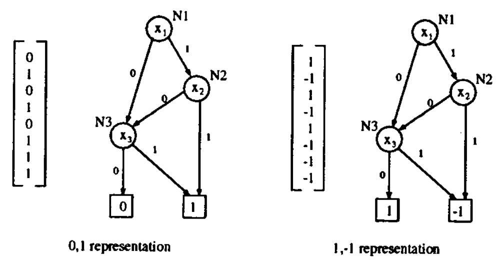


Figure 2: BDD for the logic function $x_1x_2 + x_3$

set of coefficients. For example, the n first-order coefficients often provide a good filter for Boolean function classification. Thus, it is desirable to develop an efficient method for extracting *subsets* of coefficients. We present a method for extracting a subset of coefficients using matrix and vector multiplication using BDDs. We show that for logic functions whose full spectra cannot be computed, their i^{th}-order coefficients can be easily computed using the same BDD framework.

This paper is organized as follows. In section 2, we review the original Walsh spectrum computation method. In section 3, we show the algorithm to compute the full set Walsh spectrum coefficients efficiently. Section 4 gives the partial coefficients computation procedure. We report on the experimental results in section 5, and we conclude in section 6.

WALSH TRANSFORMATION

The *Walsh matrix* is defined recursively as follows [5, 4]:

$$T_0 = 1 \qquad T_n = \begin{bmatrix} T_{n-1} & T_{n-1} \\ T_{n-1} & -T_{n-1} \end{bmatrix} \qquad (1)$$

In Walsh matrix computation, the logical 0 is encoded to 1, and logical 1 is encoded to -1. As a result, if two values are the same, their product is always 1 and if two values are different, their product is always -1.

The meanings of each row in Walsh matrix can be related to Definition 1. The first row in the matrix is the truth table for the constant logic 0, since it is encoded to 1, and correspond to the *zeroth*-order coefficient. The second row in the matrix is the truth table for the function x_1 and the third row is the function x_2. The fourth row is the function $x_1 \oplus x_2$, and so on.

We show the computation of Walsh spectrum for a Boolean function with the following example.

Example 1 *Given a Boolean function $f = x_1x_2 + x_3$, the Walsh spectrum is computed as follows:*

The matrix in the left hand side of Figure 1 is the Walsh matrix defined above (in this case, n is 3,since there are three variables, x_1, x_2 and x_3). The vector in the left hand side of Figure 1 is an encoded truth table representation of the logic function $x_1x_2+x_3$ with logical false *encoded to "1", and logical* true *encoded to "-1".*

The computed Walsh spectrum is shown in the right hand side in the figure. Each element in the Walsh spectrum shows the closeness between the logic function $x_1x_2 + x_3$ and the logic functions appeared in the rows of Walsh matrix. □

Please refer to [4] for detailed description of Walsh matrix properties.

As can be seen from this example, computing Walsh spectrum using truth tables is not applicable to logic functions with large numbers of inputs

In fact, using the matrix by vector method above, the computation of the transform involves summation of $2^n \times 2^n$ product terms. In [4], a fast Walsh procedure which utilizes common subsets of product terms can reduce the number of product terms involved to $2^n \times n$. Even with the fast algorithm, because the truth-table grows exponentially with n, it is unfeasible to compute Walsh spectrum for functions with more than 30 inputs.

BDDS AND WALSH COMPUTATION

Binary Decision Diagrams (BDDs) are an efficient way to represent logic functions [1]. Here we extend it by allowing any integer values as constant nodes instead of just 0 and 1. An example BDD which represents the logic function $x_1x_2 + x_3$ is shown in Figure 2. In the figure, both 0,1 representation and the encoded 1,-1 representation are shown. Since we allow any integer values as constant nodes, we can represent in BDD "1,-1 representation" of truth tables which is necessary to compute Walsh spectrums.

In [2], a method to compute Walsh spectrum in BDD by computing the matrix product directly. Although this method is very efficient compared to the previous approaches, such as [4, 6, 3], it can be further improved by embedding the Walsh matrix into program codes in the following way.

Computing Walsh spectrum is to compute the product of the matrix defined in (1) (section 2) and truth tables (values in truth tables are changed to 1 and -1 from 0 and 1). As can be seen from (1), given an encoded truth table of a function g, we can compute the Walsh matrix multiplied by the vector recursively as:

$$\begin{aligned} Walsh(g,n) &= T_n \cdot g \\ &= \begin{bmatrix} T_{n-1} & T_{n-1} \\ T_{n-1} & -T_{n-1} \end{bmatrix} \begin{bmatrix} g_{\bar{x}_1} \\ g_{x_1} \end{bmatrix} \\ &= \begin{bmatrix} T_{n-1} \cdot g_{\bar{x}_1} + T_{n-1} \cdot g_{x_1} \\ T_{n-1} \cdot g_{\bar{x}_1} - T_{n-1} \cdot g_{x_1} \end{bmatrix} \\ &= \begin{bmatrix} Walsh(g_{\bar{x}_1}, n-1) + Walsh(g_{x_1}, n-1) \\ Walsh(g_{\bar{x}_1}, n-1) - Walsh(g_{x_1}, n-1) \end{bmatrix} \quad (2) \end{aligned}$$

In the above equations, g_{x_i} is the cofactor of g with respect to x_i and $g_{\bar{x}_i}$ is the cofactor of g with respect to the complement of x_i.

Here *Walsh*(g,n) returns the Walsh spectrum for the logic function g assuming that there are n variables total.

From Eq. 2, we can derive the following algorithm for computing the Walsh spectrum of a function g. It can be seen as a recursive algorithm in which the Walsh matrix definition is implicitly embedded. No explicit matrix product is formed.

```
$Walsh(g, n)${
        if $(lookup(hash, g, n, \&result))$ return $result$;
        if ($g$ is a constant) return the appropriate
                constant vector;
        $i$ is the index of the top var of $g$;
        $W_x = Walsh(g_{x_i}, n-1)$;
        $W_{xb} = Walsh(g_{\overline{x}_i}, n-1)$;
        $result = yi * (W_x + W_{xb}) + yi' * (W_x$
                $- W_{xb})$;
        store $(hash, g, n, result)$;
        return $result$;
}
```

In the algorithm, y_i is defined as a vector having all 1's in the first half elements, and all 0's in the second half elements. Similarly, y_i' is a vector having all 0's in the first half elements and all 1's in the second half elements.

Please note that each operation in the above can be efficiently realized using BDD operations, which is a key point of our method. The recursive paradigm especially suits the recursive nature of BDD data structure.

In order to clarify how the procedure works, now we show an example.

Example 2 *The trace for the Walsh spectrum computation for the logic function $x_1x_2 + x_3$ using BDD are shown in Figure 3.*

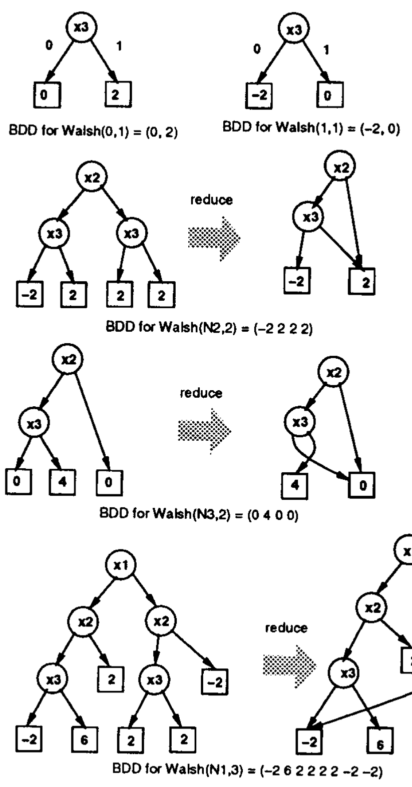
Node N_1 shows the logic function $x_1x_2 + x_3$, from which we want to obtain Walsh spectrum. The procedure descends BDD for the logic function $x_1x_2+x_3$ in a recursive way, i.e., nodes N_1, N_2, N_3, and the constant nodes are processed one by one. First we compute the Walsh spectrum for the constant 1 and -1. These are Walsh(0,1) and Walsh(1,1) in the figure. From these, we can compute Walsh spectrum for node, N_3, using the above procedure and get Walsh(N_3,2) in the figure. By continuing this process, we can get the BDD which represents the Walsh spectrum for node N1 as shown in the Figure. □

An interesting observation can be made regarding the similarity of *Walsh* algorithm and the fast Walsh transform reported in [4]. In the fast Walsh transform, the most elementary operation from which subsequent terms are shared in the "butterfly" flow-graph is exactly Equation 2. By inspection, it is clear that *Walsh* algorithm has the same computational complexity as the fast Walsh transform. However, it is clear our method is superior than previous methods since the use BDDs allows us to handle much larger Boolean functions.

The method of matrix by vector reported in [2] is less efficient by a constant factor because of the overhead needed to represent the Walsh matrix explicitly.

Computing partial Walsh coefficients

Sometimes it is impossible or unnecessary to compute the entire 2^n coefficients of a spectrum. Often, only a specific order of the coefficients is needed. For example, the *zeroth-* and *first-*order coefficients are good filters for Boolean matching applications[2]. In this section, we show how a subset of coefficients can be extracted using efficient BDD techniques.



x3
0
1
0
2
BDD for Walsh(0,1) = (0, 2)
x3
0
1
-2
0
BDD for Walsh(1,1) = (-2, 0)
x2
x3
x3
-2
2
2
2
reduce
x2
x3
-2
2
BDD for Walsh(N2,2) = (-2 2 2 2)
x2
x3
0
4
0
reduce
x2
x3
4
0
BDD for Walsh(N3,2) = (0 4 0 0)
x1
x2
x2
x3
2
x3
-2
-2
6
2
2
reduce
x1
x2
x2
2
x3
-2
6
BDD for Walsh(N1,3) = (-2 6 2 2 2 2 -2 -2)

Figure 3: Trace for the Walsh spectrum computation

Previously, we established that each row of the Walsh matrix of Eq. 1 can be seen as a Boolean function. When these rows are multiplied with the truth table vector, the coefficients result.

Assume that a subset m coefficient is to be extracted. The idea behind computing partial subsets of coefficients is as follows:

1. Select those rows representing the m coefficients from the Walsh matrix;
2. Represent the functions associated with those rows using a single BDD. The resulting BDD will require auxiliary variables to encode the different rows. Essentially, the BDD represents a matrix where each row represents a coefficient that is to be computed.
3. Perform a matrix by vector operation[2] to obtain a resulting BDD which contains m constants.

We illustrate the procedure above with an example.

Example 3 *Given $f = x_1x_2+x_3$. We wish to compute the first-order coefficients. From Figure 1, the rows corresponding to the first-order functions x_1, x_2, and x_3 are the first, second, and fourth rows. In matrix notation, the computation looks like:*

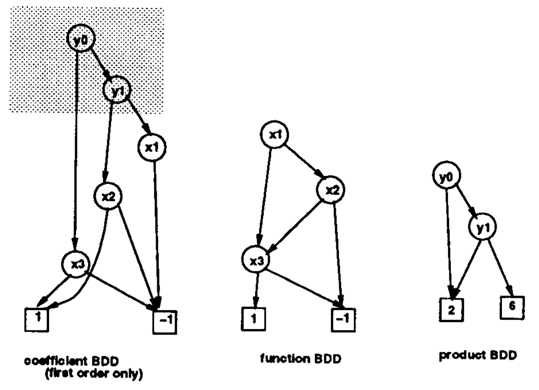


Figure 4: Extracting first-order coefficients.

Circuit Name	Output	No. of Inputs	$\|BDD\|$ (logic)	$\|BDD\|$ (Walsh)	CPU (sec)
alu4	r	14	220	4884	0.97
C1908	last	33	1261	9740	8.48
C5315	869	27	58	301	0.67
C5315	last	67	662	7354	25.75
C432	195	36	523	spaceout	—

Table 1: Full Walsh spectra computation results

$$\begin{bmatrix} 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \\ 1 \\ -1 \\ -1 \\ -1 \end{bmatrix} = \begin{bmatrix} 6 \\ 2 \\ 2 \end{bmatrix}$$

In Fig. 4, the leftmost BDD reflects selected rows of Walsh matrix. In this case, rows corresponding to first-order coefficients are selected. The shaded variables reflect the auxiliary varibles that are added to encode the rows. The middle BDD is of f. The rightmost BDD is the resulting BDD after taking the matrix by vector operation. The resulting first-order coefficients are 6, 2, 2 *for x_1, x_2, x_3 respectively.* □

It should be noted that in the limit, if this method is used to extract *all* coefficients, then the procedure will be the same as the matrix by vector method proposed in [2]. Therefore, this method is most effective when the number of coefficients to be extracted is relatively small, i.e. $m << 2^n$. Otherwise, procedure *Walsh* should be used.

EXPERIMENTAL RESULTS

We have implemented the procedure presented in the last section. First, we computed the full spectra for several experimental results, shown in Table 1. The machine used is SUN Sparc station 2.

As can be seen from the results, Walsh spectrum for complex logic functions, such as ones having more than 60 variables, can be computed within a minute on workstations. Please note that the largest example is not likely computed by any other methods, such as [4, 3], since it have more than 60 variables and it can not be represented in sum-of-products form due to excessive large number of products.

Circuit Name	Output	No. inputs	CPU time (seconds)
alu4	r	14	0.5
C1908	last output	33	5.5
C1355	1326gat	41	15.0
C432	195	36	1.7

Table 2: Partial Walsh spectra computation results

These results also indicate that the compiled procedure is much faster than the matrix multiplication method proposed in [2].

However, for some examples, such as C432, we were unable to complete the computation. The main reason is that the number of leaf (constant) nodes is so extraordinarily high that not even BDD data structure can represent it within reasonable space.

Table 2 shows some results by computing just the *first*-order coefficients. In particular, note that while it is not possible to compute the full spectrum for C432, it is very easy to compute the first-order coefficients.

CONCLUSIONS

We have shown a very efficient procedure to compute Walsh spectrum using BDD. It is much faster even compared to the matrix multiplication method in BDD which is proposed in [2]. As far as we know, this is the fastest method to compute Walsh spectrum or its related spectrum.

We also demonstrated a method for which a subset of coefficients can be computed. This is useful in cases where computing the full spectrum coefficients is unfeasible or unnecessary.

Since we can compute Walsh spectrum for fairly large logic functions, it can be a promising future direction to study on how Walsh spectrum and its related spectra can be used in logic synthesis, verification, and testing.

References

[1] R.E. Bryant. "Graph-based algorithms for boolean function manipulation". *IEEE Trans. Computer*, Vol. C-35, No. 8, pp. 667–691, Aug. 1986.

[2] E.M. Clarke, K.L. McMillan, X. Zhao, M. Fujita, and J. Yang. "Spectral Transforms for Large Boolean Functions with Application to Technology Mapping". In *Proc. 30th ACM/IEEE Design Automation Conf.*, June 1993.

[3] B.J. Falkowski, I. Schafer, and M.A. Perkowski. "Calculation of the Rademacher-Walsh spectrum from a reduced representation of Boolean functions". In *EURO-DAC '92*, pp. 181–186, 1992.

[4] S.L. Hurst, D.M. Miller, and J.C. Muzio. *"Spectral Techniques in Digital Logic"*. Academic Press, 1985.

[5] R.J. Lechner. "A transform approach to logic design". *IEEE Trans. Comput.*, Vol. C–19, No. 7, July 1970.

[6] J. Yang and G. De Micheli. "Spectral techniques for technology mapping". Technical Report CSL-TR-91-498, Stanford University, Dec. 1991.