Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping [*]

E. M. Clarke[†] K. L. McMillan[†] X. Zhao[†] M. Fujita[‡] J. Yang [§]

Abstract

The Walsh transform has numerous applications in computer-aided design, but the usefulness of these techniques in practice has been limited by the size of the boolean functions that can be transformed. Currently available techniques limit the functions to less than 20 variables. In this paper, we show how to compute concise representations of the Walsh transform for functions with several hundred variables. We have applied our techniques to *boolean technology mapping* and, in certain cases, we obtained a speed up of as much as 50% for the matching phase.

1 Introduction

The Walsh transform [3] has numerous applications in computer aided design, particularly in the synthesis and testing of combinational circuits. Unfortunately, the usefulness of these techniques in practice has been limited by the size of the boolean functions that can be handled by the transform. Since this transform is given as a vector with length of 2^n where n is the number of variables in the function, currently available techniques limit the functions to less than 20 variables. In this paper, we show how to compute concise representations of the transform for functions with several hundred variables. Our technique is based on the use of binary decision diagrams (BDDs) [1] to represent large recursively defined integer matrices like the Walsh matrix and to perform standard operations on these matrices. The basis for the implementation of the matrix operations is an efficient algorithm for performing arithmetic on functions that map large boolean vectors into the integers.

We treat an integer matrix with dimension $2^m \times 2^n$ as a function that maps boolean vectors of length $m + n$ into the integers. Since various matrix operations can be performed by operations on the corresponding integer functions, we can reduce the problem of computing the Walsh transform to a problem about integer-valued functions that can be solved using BDDs. We present two BDD-based representations for integer functions. The first treats an integer as a boolean vector and encodes the integer function as an array of boolean functions, each of which can be expressed as a BDD. The other method represents such functions by a generalization of a BDD in which the terminal nodes can have integer values instead of just 0 and 1. The Walsh matrix has a simple recursive definition and can be encoded by BDDs with size linear in the number of variables using each of the two methods. Other operations on vectors and matrices that are needed for tasks like synthesis and testing can be implemented using this approach as well. Similar techniques can also be used to compute the Reed-Muller transform of boolean functions [3] with large numbers of variables.

We demonstrate the power of our techniques by applying it to *boolean technology mapping*, an important problem in CAD. We say that two boolean functions are *matchable* if they are equivalent under permutation and complementation of inputs and complementation of outputs. In boolean technology mapping it is necessary to check frequently whether two functions are matchable or not. This can be quite time consuming. In order to make the process more efficient, it is important to have a good filter that can immediately reject unmatchable functions. The Walsh transformation can be used as the basis of such a filter. By computing the Walsh transformation using BDDs, we are able to perform technology mapping using cell libraries in which the individual cells have a large number of inputs. In this paper, we describe a modification of *Ceres* [5], a technology mapping system developed at Stanford, to use the Walsh transformation as a filter. Our experimental results show that this filter is quite good. In fact, it rejected all unmatchable functions that we encountered. Consequently, every function which passed the Walsh filter really matched a cell in the technology library.

Our paper is organized as follows: The second section briefly reviews the basic properties of BDDs that are needed in the paper. The third section describes the two ways of representing integer functions usings BDDs. The implementation of various arithmetic operations on these functions

[*]This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597 and in part by the National Science Foundation under Grant no. CCR-8722633 and in part by U.S.-Japan cooperative research from the National Science Foundation under Grant no. INT-90-16694 and in part by the Semiconductor Research Corporation under Contract 92-DJ-294

[†]School of Computer Science, Carnegie Mellon University, Pittsburgh, PA15213, U.S.A.

[‡]Fujitsu Laboratories Ltd., 1015 Kamikodanaka, Nakahara-ku, Kawasaki 211, Japan

[§]Center for Intergrated Systems,Stanford University,Stanford, CA 94309

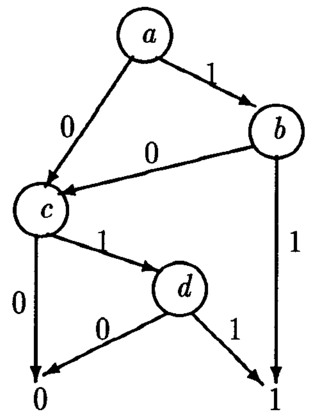

Figure 1: A BDD representing $(a \wedge b) \vee (c \wedge d)$

is also presented in this section. Section four shows how the results of the previous section can be used to implement standard operations like addition and multiplication of very large integer matrices. Section five describes how BDDs can be obtained for recursively defined integer matrices like the Walsh matrix and tells how to compute the Walsh transform for boolean functions. In this section we also illustrate the power of the two representations by computing the transforms of several very large boolean functions. Section six describes the application of these ideas in boolean technology mapping. The paper concludes in Section seven with a discussion of possible directions for future research.

2 Binary decision diagrams

Ordered binary decision diagrams (BDDs) are a canonical representation for boolean formulas described by Bryant [1]. They are often substantially more compact than traditional normal forms such as conjunctive normal form and disjunctive normal form, and they can be manipulated very efficiently. Hence, they have become widely used for a variety of CAD applications, including symbolic simulation, verification of combinational logic and, more recently, verification of sequential circuits. A BDD is similar to a binary decision tree, except that its structure is a directed acyclic graph rather than a tree, and there is a strict total order placed on the occurrence of variables as one traverses the graph from root to leaf. Consider, for example, the BDD of Figure 1. It represents the formula $(a \wedge b) \vee (c \wedge d)$, using the variable ordering $a < b < c < d$. Given an assignment of boolean values to the variables a, b, c and d, one can decide whether the assignment makes the formula true by traversing the graph beginning at the root and branching at each node based on the value assigned to the variable that labels the node. For example, the assignment $\{a \leftarrow 1, b \leftarrow 0, c \leftarrow 1, d \leftarrow 1\}$ leads to a leaf node labeled 1, hence the formula is true for this assignment.

Bryant also showed that given a variable ordering, there is a canonical BDD for every formula. The size of the BDD depends critically on the variable ordering. Bryant gives algorithms of linear complexity for computing the BDD representations of $\neg f$ and $f \vee g$ given the BDDs for the formulas f and g. The only other operations which we require for the algorithms that follow are quantification over boolean variables and substitution of variable names. Bryant gives an algorithm for computing the BDD for a restricted formula of the form $f|_{v=0}$ or $f|_{v=1}$, i.e., f with the variable v set to 0 or 1. The restriction algorithm allows us to compute the BDD for the formula $\exists v[f]$, where v is a boolean variable and f is a formula, as $f|_{v=0} \vee f|_{v=1}$. The substitution of a term w for a variable v in a formula f, denoted $f[v \leftarrow w]$ can be accomplished using quantification:

$$f[v \leftarrow w] = \exists v[(v \Leftrightarrow w) \wedge f].$$

More efficient algorithms are possible, however, for the case of quantification over multiple variables, or multiple renamings. In the latter case, efficiency depends on the ordering of variables in the BDDs being the same on both sides of the substitution.

3 Integer Operations

Let D_n be the set $\{0, \ldots, 2^{n+1} - 1\}$ of integers that can be represented with $n+1$ bits, and let B be the set consisting of the boolean values 0 and 1. Let $f : B^m \to D_n$ be a function that maps boolean vectors of length m into the set D_n. We describe two ways of representing f using Binary Decision Diagrams.

3.1 Representation as an array of BDDs

The function f is expressed as a summation

$$f(\bar{x}) = \sum_{i=0}^{n} f_i(\bar{x}) \cdot 2^i,$$

where each f_i has value 0 or 1 and is represented as a BDD. For example, the function $f(x_1, x_2) =$ if $x_1 \vee x_2$ then 3 else 4 may be represented using three BDDs as shown in Figure 2.

Arithmetical operations on such integer valued functions can be implemented in terms of logical operations on BDDs. For example, each h_i in

$$h(\bar{x}) = f(\bar{x}) + g(\bar{x}) = \sum_{i=0}^{n+1} h_i(\bar{x}) \cdot 2^i$$

are computed by the same logical operations that are used in the full adder circuit shown in Figure 3. The product of two integer functions can be computed by a series of additions. The steps involved in this process can be implemented using standard BDD operations. Computing products of integer-valued functions in this way may be very expensive in certain cases. However, we are able to avoid these cases in this paper.

Integer valued functions that take both positive and negative values can be handled in a similar manner. Let D_n^* denote the set of integers $\{-2^n, \ldots, 2^n - 1\}$. A function $f : B^m \to D_n^*$ can be represented by an array of $n+1$ BDDs, where each BDD gives one bit in 2's complement notation

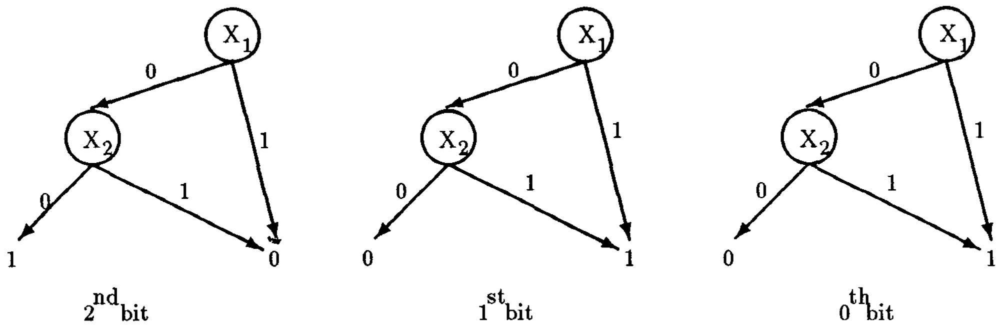

Figure 2: An array of BDDs representing if $x_1 \vee x_2$ then 3 else 4

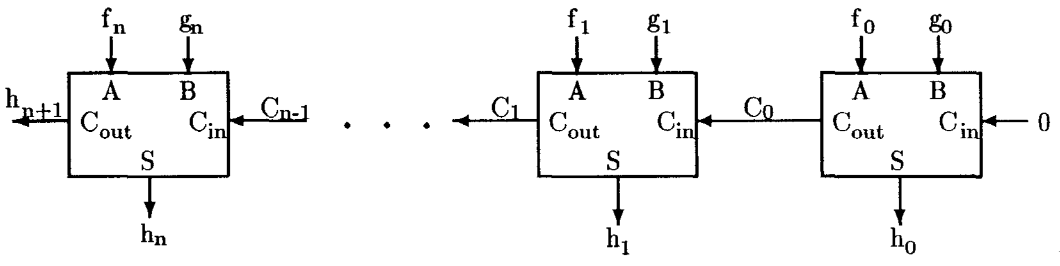

Figure 3: A full adder

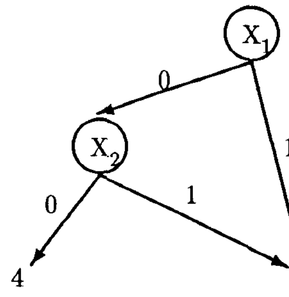

Figure 4: An extended BDD for if $x_1 \vee x_2$ then 3 else 4

for the value of the function. Multiplication and addition of these functions are implemented so that the operations on the BDDs are performed in the same way as bit operations in 2's complement arithmetic.

3.2 Representation as a BDD with integer terminal nodes

Suppose $\{n_1, \ldots, n_N\}$ are the possible values of f. The function f partitions the space B^m of boolean vectors into N sets $\{S_1, \cdots, S_N\}$, such that $S_i = \{\, \bar{x} \mid f(\bar{x}) = n_i \,\}$. Let f_i be the characteristic function of S_i, we say that f is in *normal form* if $f(\bar{x})$ is represented as $\sum_{i=1}^{N} f_i(\bar{x}) \cdot n_i$. This sum can be represented as a BDD with the possible values as its terminal nodes. For example, the function if $x_1 \vee x_2$ then 3 else 4 is represented as in Figure 4.

Any arithmetic operation $\odot$ using this representation is performed in the following way.

$$\begin{aligned} h(\bar{x}) &= f(\bar{x}) \odot g(\bar{x}) \\ &= \sum_{i=1}^{N} f_i(\bar{x}) \cdot n_i \odot \sum_{j=1}^{N'} g_j(\bar{x}) \cdot n'_j \\ &= \sum_{i=1}^{N} \sum_{j=1}^{N'} f_i(\bar{x}) g_j(\bar{x}) (n_i \odot n'_j) \\ &= \sum_{k=1}^{N''} \bigvee_{n_i \odot n'_j = n''_k} f_i(\bar{x}) g_j(\bar{x}) n''_k \end{aligned}$$

We now give an efficient algorithm for computing $f(\bar{x}) \odot g(\bar{x})$.

- If f is a leaf, which is an integer, apply $(f\odot)$ to each leaf of g.
- If g is a leaf, apply $(\odot g)$ to each leaf of f.
- Otherwise, f and g have the form in Figure 5, and the BDD for $f \odot g$, depending on the relative order of x_i and x_j, is given in figure 6.

The resulting diagram may not be in normal form. In order to convert it into normal form, a *minimization* phase is needed. The algorithm for this phase is essentially identical to the minimization phase in Bryant's algorithm for constructing BDDs [1] .

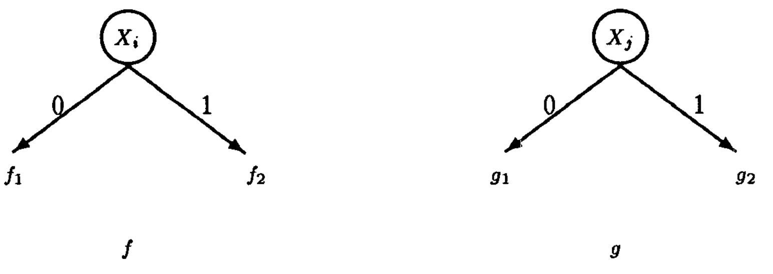

Figure 5: BDDs for f and g

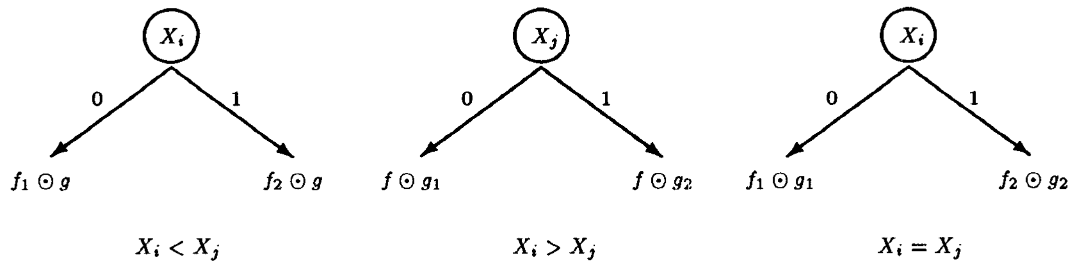

Figure 6: BDD of $f \odot g$

4 Matrix Operations

Let M be a $2^k \times 2^l$ matrix over D_n^*. It is easy to see that M can be represented as a function $M : B^{k+l} \rightarrow D_n^*$, such that $M_{ij} = M(\bar{x}, \bar{y})$, where $\bar{x}$ is the bit vector for i and $\bar{y}$ is the bit vector for j. Therefore, matrices with integer values can be represented as integer valued functions using both representations in Section 3. In this section, we will show how to implement some standard matrix operations in the two BDD representations that are needed for computing the Walsh transform. As examples, we will consider ***absolute value***, ***scalar multiplication***, ***addition***, ***summation over one dimension***, ***sorting a vector of integers***, and finally ***matrix multiplication***.

- Absolute value
  Given a $2^k \times 2^l$ matrix M, we wish to obtain another matrix M', such that for all i, j, $M'_{ij} = |M_{ij}|$. Since all of the operations in the identity

  $$|a| = \text{if } a < 0 \text{ then } -a \text{ else } a$$

  can be performed in terms of BDD operations, it is easy to obtain the BDD representation for this function using either representation for integer function. However, if we use the second representation, we can perform the operation in a more efficient manner. Suppose we have an extended BDD that represents M, then we can get the extended BDD representation for M' by first replacing the terminal nodes in M by their absolute values and then minimizing the resulting BDD. This operation will take time $O(S_M)$, where S_M is the size of the extended BDD for M.

- Scalar multiplication

  In this case we want to obtain a BDD representation for $M'_{ij} = cM_{ij}$. This can be easily done by multiplying two integer functions, where one represents the constant c and the other represents M. For each representation, the cost of this operation is $O(S_M)$.

- Matrix addition

  Suppose two matrices M^1 and M^2 with the same dimensions are given in one of the BDD representations. We want to obtain the sum of these two matrices $M'_{ij} = M^1_{ij} + M^2_{ij}$. This can be done in the obvious way by representing the result matrix by the integer function $M'(\bar{x}, \bar{y}) = M^1(\bar{x}, \bar{y}) + M^2(\bar{x}, \bar{y})$. The addition operation for both representations has already been discussed in the previous section. If we use the second representation, the complexity of this operation is the cost of adding two integer functions. In worst case this is $O(S_{M^1} \cdot S_{M^2})$.

- Summing matrices over one dimension

  It is sometimes desirable to obtain a 2^n vector from a $2^n \times 2^m$ matrix that each element in the vector is the summation of the corresponding column, i.e. $M'_i = \sum_{j=0}^{2^m-1} M_{ij}$. When the matrices are expressed in terms of integer valued functions, the equations becomes $M'(\bar{x}) = \sum_{\bar{y}} M(\bar{x}, \bar{y})$, where $\sum_{\bar{y}}$ means "sum over all possible assignments to $\bar{y}$". In practice, $\sum_{\bar{y}} M(\bar{x}, \bar{y})$ can be computed as:

$$\begin{aligned}
&\sum_{y_1 y_2 \cdots y_m} M(\bar{x}, y_1, y_2, \ldots, y_m) \\
= &\sum_{y_1 y_2 \cdots y_{m-1}} \sum_{y_m} M(\bar{x}, y_1, y_2, \ldots, y_m) \\
= &\sum_{y_1 y_2 \cdots y_{m-1}} (M(\bar{x}, y_1, y_2, \ldots, y_{m-1}, 0) \\
&\qquad + M(\bar{x}, y_1, y_2, \ldots, y_{m-1}, 1))
\end{aligned}$$

In this way, each variable in $\bar{y}$ is eliminated by performing an addition.

This operation can also be used to sum the elements of a vector and to obtain a two dimensional matrix from a three dimensional matrix by summing over one dimension. Although this operation works well in many cases, the worst case complexity can be exponential in the number of variables.

- Sorting vectors

  Frequently, it is useful to rearrange the elements in a vector so that they are in non-decreasing order. When the number of different values in the vector is not very large, the sorted vector can be represented as a list with length m, where m is the number of different values. Each element in the list contains a possible value of the function and the number of occurrences of that value.

  When the second representation for integer functions is used, it is easy to find the set of different values, since it is only necessary to collect all of the terminal nodes in the extended BDD. The number of occurrences N_k of a possible value C_k can be calculated as $N_k = \sum_{i=0}^{2^n-1}(\text{if } M_i = C_k \text{ then } 1 \text{ else } 0)$. The operation of summation over a vector discussed previously can be applied to compute this sum. Although, in general, the complexity of the summation operation does not have a satisfactory upper bound, summation over a vector takes time linear to the size of the BDD representing the vector. Thus the complexity of the sorting operation is $O(m \cdot S_M)$.

- Matrix multiplication

  Suppose that two matrices A and B have dimensions $2^k \times 2^l$ and $2^l \times 2^m$, respectively. Let $C = A \times B$ be the product of A and B, C will have dimension $2^k \times 2^m$. If we treat A and B as integer valued functions, we can compute the product matrix C as

  $$C(\bar{x}, \bar{z}) = \sum_{\bar{y}} A(\bar{x}, \bar{y}) B(\bar{y}, \bar{z})$$

  using the summation operation discussed above. In general, the complexity of this operation can also be exponential in the number of variables.

5 Spectral transformations of boolean functions

One of the most commonly used transformations in digital circuit design is the Walsh transform [3]. In this section, we will show how the BDD based techniques described previously can be used to compute concise representations of the spectra for this transformation.

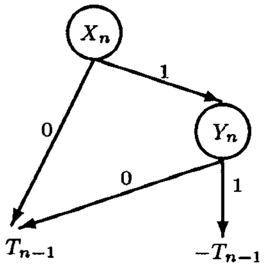

Figure 7: BDD for T_n

The Walsh matrix T_n has the recursive definition:

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

Each element of the matrix is determined by its row and column coordinates. We will encode the 2^n columns by variables $y_n, \ldots, y_1$ and the 2^n rows by the variables $x_n, \ldots, x_1$. T_n can be represented as an integer valued function:

$$\begin{aligned}
&T_n(y_n, \ldots, y_1, x_n, \ldots, x_1) \\
= \;&T_{n-1}(y_{n-1}, \ldots, y_1, x_{n-1}, \ldots, x_1) \\
&\cdot(\text{if } x_n y_n = 1 \text{ then } -1 \text{ else } 1)
\end{aligned}$$

The above recursive definition can be expressed by a BDD as shown in Figure 7.

In expressing the Walsh transform, it is convenient to encode the boolean 0 by the integer 1 and boolean 1 by integer -1. In general, the boolean vector $v \in B^n$ will be replaced by $v' \in \{-1, 1\}^n$, where $v' = 1 - 2v$. For example, the column vector $[0,1,1,1,1,0,0,0]^T$ is encoded as $[1,-1,-1,-1,-1,1,1,1]^T$. The Walsh transform maps an encoded boolean vector f with length 2^n to an integer vector of length 2^n, denoted by W_f, in which each component is between -2^n to 2^n. The transform can be easily expressed using the Walsh matrix, $W_f = T_n f$. [3] For example, the vector encoded by $[1,-1,-1,-1,-1,1,1,1]^T$ is mapped into $[0,0,0,0,-4,4,4,4]^T$. When the number of variables is large, this computation can be performed by representing both the matrix and the vector as BDDs and computing the product as described in Section three and Section four.

For a boolean function f with n variables, the kth order Walsh spectrum for f, denoted by W_f^k, is defined as the sequence of elements $W_f(\bar{y})$ in the Walsh spectrum such that $\sum_{j=1}^{n} y_j = k$. Its BDD representation can be obtained from the BDD for W_f using the following formula

$$W_f^k(\bar{y}) = \text{if } \Sigma\bar{y} = k \text{ then } W_f(\bar{y}) \text{ else } D$$

where D is some default number. Most of the matrix operations can be applied to W_f^k exactly as described in the previous section. In the case of sorting, however, we only need the list of values that are distinct from the default value D.

example circuit					1st rep.		2nd rep.		distinct
circuit	|input|	output	# of gates	$\lvert BDD \rvert$	|BDD|	time	|BDD|	time	coefficients
c1355	41	1326gat	546	9451	9961	860	5102	134	4
c1908	33	9	880	3607	3742	112	1850	44	18
c3540	50	361	1669	520	49940	4901	15985	171	39
c5315	178	854	2307	210	1416	53	925	57	8
50-bit adder	100	C_{50}	250	151	3180	56	7456	23	100
100-bit adder	200	C_{100}	500	301	11184	456	29906	128	200

Table 1: Experimental results for Walsh transformations

To illustrate the power of these techniques, we have computed the Walsh transformation for some large combinatorial circuits, including two adders and some of the ISCAS benchmarks (Table 1). The examples were run on a DEC-5000 and run time is shown in seconds. Because the first representation needs arithmetic operations that must be performed bit by bit, in most cases it requires more time to compute. However, when the number of distinct Walsh coefficients is large, the first representation may use less space than the second.

6 Boolean technology mapping

Technology mapping [5, 7] is a crucial step in logic synthesis, since it greatly influences the quality of the final synthesized circuit. This process is usually divided into three phases. In the first phase the entire circuit is partitioned into single-fanout sub-circuits. In the second phase a logic function is extracted from each sub-circuit. Finally, an existing cell in the technology library is assigned to each logic function. The obvious implementation for the last step is to compare the extracted function with the boolean functions for all of the cells in the library. We may have to check many cells before we find one that matches, so this phase is the most time consuming part of technology mapping.

When matching a cell to a function, we concentrate on permutation of inputs, complementation of inputs and complementation of the function. In the following discussion we use some simple terminology from group theory to describe these operations on boolean functions [2, 4]. Let $\mathcal{B}_n$ denote the set of boolean function with n variables. First, we consider *complementation of inputs.* $\mathcal{C}_2 =< \{0,1\}, \oplus, 0 >$ will be the group defined by the operation of addition modulo 2. $\mathcal{C}_2^n = \mathcal{C}_2 \times \ldots, \times \mathcal{C}_2$ will denote the direct product of n copies of $\mathcal{C}_2$. The group $\mathcal{C}_2^n$ acts on $\mathcal{B}_n$ in the obvious way. If $\rho = (r_1, \ldots, r_n) \in \mathcal{C}_2^n$ and $f \in \mathcal{B}_n$, then ρf is defined by $(\rho f)(x_1, \ldots, x_n) = f(x_1 \oplus r_1, \ldots, x_n \oplus r_n)$. *Permutation of inputs* is handled in a similar manner. Let $\mathcal{S}_n$ be the group of all permutations on n-elements (i.e. the symmetric group on n-elements). The action of $\mathcal{S}_n$ on $\mathcal{B}_n$ is defined so that if $\sigma \in \mathcal{S}_n$ and $f \in \mathcal{B}_n$, then $(\sigma f)(x_1, \ldots, x_n) = f(x_{\sigma(1)}, \ldots, x_{\sigma(n)})$. Finally, the group $\mathcal{N}$ for *function complementation* consists of two elements, the identity **I** and the negation operator **N**. These operators act on $\mathcal{B}_n$ in the expected way: $\mathbf{I}f = f$ and $(\mathbf{N}f)(x_1, \ldots, x_n) = \neg f(x_1, \ldots, x_n)$.

The group that we are concerned with in this section is

$$\mathcal{Y}_n = \{(n, \rho, \sigma) | n \in \mathcal{N}, \rho \in \mathcal{C}_2^n, \sigma \in \mathcal{S}_n\}.$$

$\mathcal{Y}_n$ acts on $\mathcal{B}_n$ by $((n, \rho, \sigma)f)(x_1, \ldots, x_n) = (nf)(x_{\sigma(1)} \oplus r_1, \ldots, x_{\sigma(n)} \oplus r_n)$. The multiplication operation of this group is defined so that $(n_1, \rho_1, \sigma_1) * (n_2, \rho_2, \sigma_2) = (n_1 n_2, \rho_1 \oplus \sigma_1(\rho_2), \sigma_1 \sigma_2)$. The *matching procedure* for two given functions f and g will attempt to find some $\gamma \in \mathcal{Y}_n$ so that $\gamma f = g$. The goal of the matching phase is easy to state using this terminology. Given a boolean function f, apply the matching procedure to the cells in the library to find a $\gamma \in \mathcal{Y}_n$ and a boolean function g for some cell in the library such that $\gamma f = g$.

Since the matching phase is expensive, it is desirable to reduce the number of cells we must match against the given function. Because the number of cells that match a given function is usually very small, it is important to find *filters* for limiting the number of cases in which the matching procedure must be used. Any necessary condition for boolean matching can be used as such a filter. The *Ceres* technology mapping program [5] developed at Stanford as part of the *Olympus Synthesis System* [6] uses two filters. One is based on unateness of inputs; the other is based on symmetry. It is obvious that if the two functions have different numbers of unate inputs, they cannot be matched. Likewise, if the two functions have different numbers of inputs which are symmetric with one another, they are not matchable. *Ceres* counts the number of unate inputs and the number of symmetric inputs and uses these numbers in filters. These two filters eliminate a large fraction of the unmatchable functions. However, a significant number of unmatchable functions still pass through both of these filters.

By using the Walsh transformation, we can obtain a more powerful necessary condition for matching boolean functions. If $W_f(\bar{y})$ denotes the Walsh coefficients of $f \in \mathcal{B}_n$, then the following properties hold:

$$W_{Nf}(\bar{y}) = -W_f(\bar{y})$$

$$W_{\rho f}(\bar{y}) = W_f(\bar{y}) \prod_{i=1}^{n} (-1)^{r_i y_i}$$

$$W_{\sigma f}(\bar{y}) = W_f(\sigma(\bar{y}))$$

For each function f, let $|W_f^k|$ denote the sequence obtained by sorting the absolute values of the k^{th} order Walsh coefficients W_f^k. From the above properties, it is obvious that

Circuit	Depth 3				Depth 5				Depth 7			
	Walsh		Ceres		Walsh		Ceres		Walsh		Ceres	
	M_w	time	M_c	time	M_w	time	M_c	time	M_w	time	M_c	time
C1355	532	687	968	372	639	840	1096	509	641	1068	1102	628
C3540	1423	1993	3550	1714	1447	3506	4401	4046	1570	7006	4855	16245
C432	272	367	613	336	288	615	740	619	302	1503	813	1755
C880	429	616	973	453	447	1214	1170	1170	460	2600	1329	4471
alu2	697	926	2374	1150	718	1544	3234	2394	1398	7564	4168	8298
alu4	1195	1510	4047	1964	1214	2350	5400	3981	2263	10699	6827	13230
apex6	718	876	1682	835	730	1063	1778	1227	735	1621	1808	2533
f51m	329	442	1158	568	341	788	1550	1351	637	2920	1968	4550
x1	2135	2489	8493	4269	2157	4309	13526	9288	5997	29244	17819	24309
z4ml	265	325	933	417	274	682	1281	1033	608	2657	1700	3225

Table 2: LSI Library Mapping Results

$|W_f^k|$ is invariant over the operators in $\mathcal{Y}_n$. Thus the equivalence of $|W_f^k|$ and $|W_g^k|$ can be used as a necessary condition for determining whether f and g match and hence as a filter in the matching phase. In our experiment we used $|W_f^0|$ and $|W_f^1|$ as filters because they were easy to compute.

The results of our experiments are summarized in table 2. The *depth* of a circuit is the maximum number of gates from the circuit's inputs to its output. The program attempts to match the sub-circuits obtained by partitioning with library functions of depth three, five and seven. For each depth, M_w gives the number of calls to the match procedure after the Walsh filter is applied, and M_c gives the corresponding number after the *Ceres* filters are applied. The *time* column measures the runtime required for the *matching phase* and does not include the time for other phases in the synthesis process. The examples were run on a DEC 5000/240, and the runtime is given in milliseconds. When compared to *Ceres*, the Walsh transform method requires considerably fewer matches. This is due to effective filtering of candidate circuits. After applying the Walsh filter, *all* calls to the matching procedure are successful. On the other hand, *Ceres* does not provide completely effective filters and must perform many unnecessary matches. The runtime performance with the BDD-based Walsh filter is usually lower for most of the circuits that we considered. As the *depth* increases, our filter becomes increasingly superior to the filters used by *Ceres*. This is expected since BDDs can deal with large circuits better than conventional methods.

7 Directions for future research

In this paper, we have developed techniques for representing functions that map large boolean vectors into the integers. We have also shown how to implement arithmetic operations efficiently on such functions. By using these methods we are able to perform many standard operations on integer matrices. These techniques are applied to the problem of computing a concise representation for the Walsh spectrum of a boolean function. We have demonstrated the power of this technique on several very large examples. In addition, we have shown how these techniques can be used to speed up boolean technology mapping.

In subsequent papers, we plan to investigate how the ideas in this paper can be applied to other problems in the design and testing combinational circuits. We also believe that the methods for performing matrix operations will be useful for a wide range of problems. Many problems in graph theory can be expressed using matrix operations and hence can be solved using BDD technique. For example, the all–pairs–shortest–path problem for extremely large directed graphs with weighted edges can be clearly be solved using these techniques.

Acknowledgements

We would like to thank David Long of CMU for his help in the implementation of the techniques discussed in the paper and for his help in writing the paper.

References

[1] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[2] M. A. Harrison. *Introduction to switching and automata theory.* McGraw-Hill, 1965.

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

[4] R. J. Lechner. A transform approach to logic design. *IEEE Transactions on Computers*, C-19(7), 1970.

[5] F. Mailhot and G. De Micheli. Technology mapping using boolean matching and don't care sets. In *Proceedings of the 1990 European Design Automation Conference*, 1990.

[6] G. De Micheli, David Ku, F. Mailhot, and T.K. Truong. The olympus synthesis system for digital design. *IEEE design and test of computers*, October 1990.

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