Methods for calculating autocorrelation coefficients

J. Rice and J. C. Muzio
Department of Computer Science
University of Victoria
PO Box 3055
Victoria, B.C., Canada
V8W 3P6
email: jrice@csc.uvic.ca and jmuzio@csr.uvic.ca

Abstract

Calculation of a function's autocorrelation coefficients is usually done either by brute force application of the defining equation, or by using a fast method to calculate the interim spectral coefficients and applying a transform to these coefficients. Both of these methods require time exponential in the number of inputs to calculate each coefficient. We present in this paper various methods for more efficient calculation of a function's first order autocorrelation coefficients. The first method is based on a disjoint cube list, and requires on the order of p^2 time for each coefficient, where p is the number of cubes in the disjoint list. The other methods are based on ROBDD representations of the function(s). Advantages and disadvantages of each method is discussed, and possibilities for other techniques are introduced.

1 Introduction

Boolean functions are functions for which the inputs and the output(s) are restricted to the Boolean domain. If a transformation matrix $\mathbf{T}^n$ is premultiplied by the output vector of the function, the result is a representation of the function in the spectral domain, usually referred to as the *spectral coefficients* of a function. An alternative view of the function, also in the spectral domain, is the *autocorrelation coefficients* of the function.

A function's spectral coefficients give a measure of similarity of the function to one of its inputs, or, to a combination of its inputs. The spectral coefficients of a function have been used in many areas of design, optimization, and testing of digital circuits [4, 6, 7] and for solving problems such as classification of functions [6] and Boolean matching [3]. The autocorrelation coefficients provide a measure of the function's similarity to itself, shifted by a given amount. This is also called the cross-correlation, or convolution function. The autocorrelation coefficients have been used to perform optimization and synthesis of combinational logic [13], and to order variables when building ROBDDs [9]; however, their use has been limited, likely due to the fact that the commonly used methods of computation are exponential in the number of inputs to the function(s).

The information inherent in the autocorrelation coefficients has the potential to be very useful, particularly in areas such as identifying complexity measures of functions [12] and symmetries of functions. However, it is necessary to first find efficient methods to compute the values. We discuss various methods for computation of the autocorrelation coefficients, and weigh the advantages and disadvantages. It should be noted that all methods have been applied to multiple-output functions but are currently limited to calculation of the first order autocorrelation coefficients only.

2 Background

2.1 Autocorrelation Coefficients

Boolean functions can be translated to other domains, such as the *spectral domain*. In this paper we consider the calculation of the function's *autocorrelation coefficients*.

The general cross-correlation (convolution) function between two given functions f and g at a distance u is defined as

$$\sum_{v=0}^{2^n-1} f(v) \cdot g(v \oplus u) \tag{1}$$

for two functions $f(X)$ and $g(X)$ where $X = x_n x_{n-1} \ldots x_1$. The symbol $\oplus$ represents the bitwise exclusive-or function and the symbol $\cdot$ represents the arithmetic multiplication function. When $f = g$, the resulting equation gives the cross-correlation of a function with itself, translated by u. The resulting coefficients are referred to as the autocorrelation coefficients of the function.

The autocorrelation function is defined as follows:

$$B(u) = \sum_{v=0}^{2^n-1} f(v) \cdot f(v \oplus u) \tag{2}$$

where the function is $f(X)$, $X = x_n x_{n-1} \ldots x_1$, $u = \sum_{i=1}^{n} u_i 2^{i-1}$, $v = \sum_{i=1}^{n} v_i 2^{i-1}$, and n is the number of inputs [7].

For multiple output functions a second step must be performed, to combine the autocorrelation function for each of the individual functions into the *total autocorrelation function*. The total autocorrelation function is defined as

$$B(u) = \sum_{i=0}^{m-1} B_i(u) = \sum_{i=0}^{m-1} \sum_{v=0}^{2^n-1} f_i(v) \cdot f_i(v \oplus u) \tag{3}$$

where m is the number of outputs and the multiple-output function F consists of $f_0 f_1 \ldots f_{m-1}$.

Using Equation 3 to calculate the autocorrelation coefficients for a function requires $O(m2^{2n})$ operations. However, there is a significantly faster method for calculating the autocorrelation coefficients. Using the spectrum $\mathbf{R}_i$ of the function f_i, as calculated using the Rademacher-Walsh or another appropriate transform $\mathbf{T}^n$ where $\mathbf{R}_i = \mathbf{T}^n \mathbf{f}_i$, then the autocorrelation function can be computed as

$$\mathbf{B}_i = \frac{1}{2^n} \mathbf{T}^n \mathbf{R}_i^2 \tag{4}$$

where ${\mathbf{R}_i}^2$ is defined as the spectral vector $\mathbf{R}$ for the ith function output with each term squared. Each corresponding element of $\mathbf{B}_i$ is then summed over the number of outputs to calculate the total autocorrelation coefficients, $\mathbf{B}$. Using this method requires $O(mn2^n)$ operations, assuming a fast method is used to calculate $\mathbf{R}_i$[7].

In this paper we are calculating only the $n+1$ *first order* autocorrelation coefficients; that is, the coefficients for $u = 0$ *to* $2^n - 1$ where the weight of u is 0 or 1.

The simple use of Equation 4 requires on the order of 2^n operations simply to calculate $\mathbf{R}$, followed by squaring each element, and again applying the matrix multiplication. A faster method can be used for the matrix multiplication, but still, all 2^n elements of the vector $\mathbf{R}^2$ must be calculated and stored for determination of each autocorrelation coefficient. Thus this simple implementation of Equation 4 was not implemented, due to its infeasibility.

Implementation of Equation 3 also requires an exponential number of calculations for each coefficient if it is applied to each of the 2^n outputs of the function. However, if it is instead, applied only to the true minterms of the function, it is conceivable that large savings can be made. This is still a very simple algorithm, with little in the way of optimization; however, it will clearly perform far better than the straightforward application to each of the 2^n outputs (in most cases; the worst case situation would be if all but one of the 2^n inputs were true minterms). This method has been implemented as a comparison for the more sophisticated methods discussed further on.

2.2 ROBDDs

Reduced Ordered Binary Decision Diagrams (ROBDDs) are an efficient data structure for the representation and manipulation of boolean functions. The original representation, Binary Decision Diagrams (BDDs) were presented by Akers in 1978 [1], and are simply a graphical implementation of the Shannon decomposition of the function,

$$f(X) = \overline{x} f_0 + x f_1$$

where f_0 is the function f where $x = 0$ and f_1 is the function f where $x = 1$.

Restrictions to create ROBDDs were added by Bryant [2]. These restrictions are that all redundant nodes are removed, duplicate nodes are shared, and an ordering is imposed on the variables.

The BDD and ROBDD representation for the function $f = AB + AC$ are shown in Figure 1.

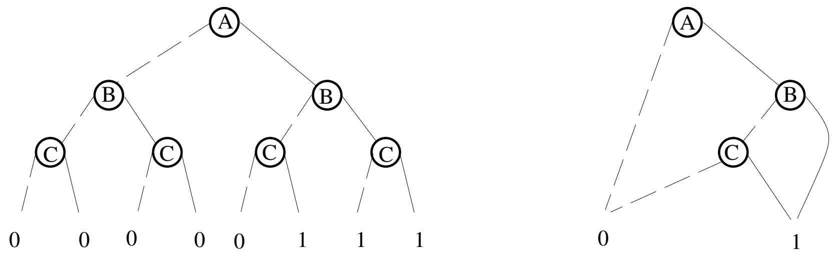

Figure 1: The BDD and ROBDD representation of $f = AB + AC$

3 Techniques for Calculation of the Autocorrelation Coefficients

More sophisticated methods of calculating the autocorrelation coefficients make use of various representations that do not require all of the possible input combinations to be described.

3.1 Disjoint Cubes Method

The method we refer to as the *disjoint cubes method* uses a list of cubes as the representation of the function(s). This method is an improvement over the method described above because it takes into account two things: first of all, that not all of the 2^n input combinations are in the on-set, and secondly, that a single cube may represent more than one input combination through the use of *don't care* values, meaning that the value of that variable has no effect on the output for the given combination of the other input variables.

Our version of the disjoint cubes method is based on a technique for calculating the Rademacher-Walsh coefficients described in [5]. A technique previously published by Varma and Trachtenburg [12, 4] is very similar to this method.

Briefly, this technique requires a disjoint set of cubes. The disjoint cube list may be found for each distinct output (as we have implemented), or over all the outputs (see [4]).

The algorithm then requires summing the weights, or contributions that each cube makes to the autocorrelation coefficient being computed. See [4] for further details.

3.2 BDD Methods

Two methods of calculating the autocorrelation coefficients were implemented using ROBDDs as a data structure. Both of the methods described below were implemented using the CUDD decision diagram package [10].

3.2.1 Comparing two ROBDDs

This technique is a very straightforward calculation method based on the initial representation of the function as a ROBDD. To calculate the 0 order coefficient is simply a matter of counting the minterms.

To calculate the first order coefficients, the following is performed:

- For each variable, create an ROBDD by taking the original function and replacing that variable with its negative. Call this function f_2, and the original f_1.
- Next, peform the AND operation $f_1 \cdot f_2$ and call the result f_3.
- Count the minterms in f_3.

The first step in this method has the effect of shifting the function by the variable of interest. The second step compares the two functions and results in only minterms that are common in the two functions. The final step counts these minterms, to give us the autocorrelation coefficient.

When calculating the autocorrelation coefficients for multiple output functions, the coefficients are summed over all the functions.

3.2.2 A Recursive Method

The recursive technique requires that only one ROBDD be built. For each coefficient required, a recursive comparison is done of the appropriate outputs of the function, as represented by the ROBDD. For instance, for the value $u = 100$ where the variables for a function f are ordered $x_0x_1x_2$, the algorithm will compare the left subtree of x_0 with the right subtree of x_0. Where any 1's match in position in the left and right subtrees the value 2 is added to the coefficient being calculated. In this way the reduced properties of the ROBDD are used. Again, for multiple output functions, the values of each individual function are found and then summed for each coefficient.

4 Experimental Results

The methods implemented were the disjoint cubes method described in subsection 3.1, the simple BDD-based method described in subsection 3.2.1, the recursive BDD-based method described in subsection 3.2.2, and a brute force method as described at the end of subsection . This was implemented mainly for comparison and verification purposes. The disjoint cubes and brute force methods were implemented in C++, while the BDD-based methods were implemented in C and made use of the CUDD package [10].

All methods were checked on approximately 130 benchmarks, and timings as generated on an Ultra 10 SPARC Server running Solaris 2.7 were compared. It should be noted that the memory usage for the BDD package was limited to 50 MB, and that timing statistics include building the BDD and performing initial sifting. For the disjoint cubes method the timing statistics include determing the disjoint cube set from the given input.

Memory usage statistics are not reported at this time, as the brute force and disjoint cubes method store only the on-sets of the functions, and the value of one coefficient at a time; thus their memory usage is self-limiting. The BDD package was limited to use of 50 MB of memory. All input files were used in the ESPRESSO-MV (or pla) format [8].

Table 1 summarizes the performance of the four techniques used for calculating the autocorrelation coefficients. The first column lists the method that was used for generating the statistics on that row. The second column lists for each method the number of benchmarks for which the method did not complete, out of the 130 tested. Reasons for not completing included running out of memory, taking too long (*e.g.* over 10 hours of cpu time), or some internal failure of the algorithm. The third column lists the percentage of successful benchmarks for that method that completed in the lowest time of all the methods, or within 5 seconds of that time. The final column lists the average time for computing the coefficients over all the successful benchmarks, for each method.

There is only one benchmark for which all methods fail (o64). Also, although the average time over all benchmarks for the disjoint cubes method is high, this is due to a small number of

method	**number of benchmarks that failed**	**percentage within 5 sec of lowest time**	**avg time over all benchmarks**
disjoint cubes	4	87 %	43.33 sec
BDD-simple	3	100 %	0.72 sec
BDD-recursive	10	68 %	24.52 sec
brute force	64	60 %	98.90 sec

Table 1: Some statistics for comparison of calculation methods for the first order autocorrelation coefficients.

files having high timing results. In fact, only 13 benchmarks take over 10 seconds to complete (10 % of the successful completions) when using the disjoint cubes method. The simple BDD method has only 1 benchmark requiring more than 10 seconds to complete ($< 1\%$), while the recursive BDD method has 26 benchmarks requiring over 10 seconds to complete (22 %).

For comparison, the brute force method failed (ran out of memory, in most cases) on 64 of the 130 files, and had an average run time of 98.9 seconds. 30 % of the completed benchmarks required over 10 seconds to complete.

Complete results are given in the Appendix.

5 Discussion

The disjoint cubes method requires $m * (n * \frac{p*(p+1)}{2})$ operations where again n is the number of variables, m is the number of outputs, and p is the number of disjoint cubes. Initial overhead will be incurred for building the disjoint cube set.

The simple BDD method requires overhead for building the initial function, and minimization is performed to reduce the size if possible (sifting). After this the algorithm must perform a COMPOSE operation and an AND operation $m * n$ times.

The recursive BDD method will compare each of the 2^n outputs for each of the m functions; however, redundancy in the BDD is inherently taken into account. Thus if a particular output is independent of a variable, no comparisons are done in computing the coefficient for that output. If a particular part of the BDD is independent of the variable, this will also save in comparisons. At this time, the method reorders the BDD for each computation. The ordering time is included in the overall timing values reporeted, and additionally, the reordering may increase the size of the ROBDD, and thus the time required for computation. The algorithm may be modified in order to remove this need for reordering; this could have a marked effect on the overall times required for computation of each coefficient.

It is a clear indication of the efficiency of the CUDD package that the simple BDD method performs so well, even though a new BDD must be built for each coefficient.

6 Conclusions

We have implemented and compared four methods for computing a function's first-order autocorrelation coefficients. The brute force method clearly is not feasible for functions of any reasonable size. The disjoint cubes method and both BDD methods complete on most benchmarks in a reasonable amount of time, and can be modified to compute a single coefficient if required.

Each have their own disadvantages. The disjoint cubes method requires a disjoint cube set, which in some cases cannot be computed in a reasonable amount of time. The BDD methods both require that a BDD for the function be built; if a poor initial ordering is used this may require more memory than is available. The simple BDD method requires building a new BDD for each coefficient computed, while the recursive BDD method, while requiring only one BDD, (currently) reorders the BDD for each coefficient computed.

All three of the methods run in a reasonable amount of time on a majority of the benchmarks tested, and provide a feasible method of computing the first order autocorrelation coefficients.

7 Future Work

Much work in finding an efficient method of computation is required in order for the autocorrelation coefficients to be a useful tool in logic synthesis. The methods presented are efficient in most cases; however, each have various benchmarks for which the method fails. Future work may include heuristics on the best method to use.

All of the methods require work to optimize in terms of memory and time. The recursive BDD method, in particular, can be improved by removing the necessity for reordering the ROBDD for each calculation (as discussed above). All of the methods may be extended to multiple order coefficients, and each of the methods may also be modified to compute the coefficients on all of the outputs of a function at once, instead of individually and then summing the result.

Other techniques under investigation include use of QDDs [11] for fast determination of the autocorrelation coefficients.

A Appendix

The following table gives the complete list of experimental results. All times are given in seconds. Blank entries indicate that the method failed to complete for that benchmark.

file	num inputs	num products	num outputs	BDD recursive	BDD simple	disjoint cubes	brute force
5xp1	7	75	10	1.50	0.70	0.00	0.7
9symml	9	87	1	0.30	0.20	0.00	2.2
C17	5	7	2	0.30	0.20	0.00	0.0
C5315	49	517	37	16.90	0.40	3.70	
C7552	5	16	16	2.50	0.20	0.00	0.1
add6	12	355	7	1.80	0.20	0.80	104.7
adr4	8	75	5	0.90	0.10	0.00	1.3
alu1	12	19	8	1.00	0.10	0.00	21.8
alu2	10	261	6	1.30	0.20	0.00	16.1
alu3	10	66	8	1.90	0.20	0.00	7.8
alu4	14	1799	8		0.50	3.10	2180.9
apex1	45	206	45	419.10	5.70	52.00	
apex2	39	1035	3	427.30	2.80		
apex3	54	280	50			37.60	
apex4	9	438	19	6.00	0.30	0.40	38.4
apex5	117	1227	88	76.00	1.50	84.80	
apex6	143	4377	139		4.60	1378.80	
apex7	49	517	37	16.80	0.40	3.70	
apla	10	26	12	3.30	0.10	0.00	8.5
b1	45	1224	45	385.70	2.10	7.70	
b9	41	138	21	5.20	0.20	0.20	
bc0	26	179	11	6.80	0.30	1.10	
bca	26	180	46	27.60	0.40	0.20	
bcb	26	155	39	22.90	0.40	0.10	
bcc	26	137	45	25.80		0.10	
bcd	26	117	38		0.30	0.00	
bw	5	87	28	4.40	0.20	0.00	0.3
cc	21	52	20	2.40	0.10	0.00	
chkn	29	140	7	5.10	0.20	16.20	
cht	47	120	36	5.10	0.20	0.00	
clip	9	167	5	1.50	0.20	0.00	5.2
cm150a	21	17	1	21.80	1.80	0.00	
cm151a	19	45	9	2.50	0.20	0.00	
cm152a	11	8	1	0.40	0.10	0.00	2.3
cm162a	27	61	17	4.30	0.20	0.00	
cm163a	16	37	13	2.00	0.20	0.00	0
cm42a	4	40	10	1.20	0.10	0.00	0.0
cm82a	5	23	3	0.40	0.10	0.00	0.0
cm85a	11	48	3	0.90	0.20	0.00	9.7
cmb	16	54	4	1.50	0.10	0.00	573.7
co14	14	47	1	0.50	0.10	0.00	26.8
con1	7	9	2	0.40	0.20	0.00	0.1
count	35	184	16	6.70	0.20	0.10	
cps	24	162	109	64.50	0.40	0.60	
cu	14	22	11	2.10	0.10	0.00	128.9
dc1	4	9	7	0.90	0.20	0.00	0
dc2	8	40	7	1.40	0.10	0.00	1.2
decod	5	16	16	2.50	0.10	0.00	0.1
dist	8	121	5	1.30	0.20	0.00	2.1
dk17	10	18	11	2.60	0.20	0.00	6.5

file	num inputs	num products	num outputs	BDD recursive	BDD simple	disjoint cubes	brute force
dk27	9	10	9	1.00	0.10	0.00	2.0
dk48	15	22	17	2.80	0.20	0.00	423.5
duke2	22	87	29		0.30	0.00	
e64	65	65	65	93.70	0.60	0.00	
ex1	5	16	1	0.20	0.10	0.00	0
ex2	5	7	1	0.20	0.10	0.00	0.0
ex3	5	4	1	0.20	0.10	0.00	0.0
example2	10	261	6	1.30	0.20	0.0	
exep	30	109	63	30.00	0.30	0.10	
f2	4	12	4	0.50	0.10	0.00	0.0
f51m	14	1799	8	4.60	0.60	3.10	2177.4
frg1	28	119	3	1.20	0.20	6.30	
gary	15	107	11	5.10	0.20	0.10	834
in0	15	107	11	5.10	0.20	0.10	833.8
in1	16	104	17	10.10	0.40	1.00	6780.9
in2	19	137	10	6.20	0.20	0.70	
in3	35	74	29	13.60	0.30		
in4	32	212	20	14.90	0.30	14.80	
in5	24	62	14	6.90	0.20	0.10	
in6	33	54	23	8.30	0.20	0.10	
in7	26	54	10	4.60	0.20	0.10	
jbp	36	122	57	17.90	0.30	0.00	
k2	45	1224	45	385.90	2.10	7.70	
lal	21	52	20	2.40	0.20	0.00	
ldd	47	120	36	5.10	0.20	0.00	
life	9	512	1	0.40	0.20	0.00	2.8
log8mod	8	47	5	1.00	0.10	0.00	1.5
majority	5	5	1	0.20	0.10	0.00	
max46	9	46	1	0.30	0.10	0.00	1.2
misex1	8	32	7	1.20	0.20	0.00	1.0
misex2	25	29	18	4.60	0.10	0.00	
misex3	14	1848	14	10.70	0.60	2.80	2253.5
misex3c	14	305	14	4.90	0.20	11.40	486.7
misg	56	69	23	2.30	0.20	0.20	
mish	94	82	43	4.40	0.30	0.10	
mlp4	8	128	8	1.70	0.20	0.00	2.5
mux	21	36	1	20.40	2.70	0.00	
o64	130	65	1				
opa	17	78	69	21.20	0.20	0.00	
parity	16	32768	1	72.60	30.50	3689.50	
pcle	18		9	1.70	0.10	0.00	
pcler8	16	45	5	1.10	0.10	0.00	545.1
pm1	4	40	10	1.30	0.10	0.00	0.0
radd	8	75	5	0.90	0.10	0.00	1.3
rckl	32	32	7	7.50	0.20	0.00	
rd53	5	32	3	0.50	0.20	0.00	0.0
rd73	7	141	3	0.70	0.20	0.00	0.6
rd84	8	256	4	1.10	0.10	0.10	3.1
risc	8	28	31	4.60	0.20	0.00	3.1
root	8	57	5	1.30	0.10	0.00	1.2
ryy6	16	112	1	0.50	0.10	0.10	629.2
sao2	10	58	4	1.30	0.10	0.00	6.3
sct	19	74	15	3.10	0.20	0.00	
seq	41	1459	35		9.20	26.90	
sqn	7	38	3	0.70	0.10	0.00	0.3
sqr6	6	50	12	1.50	0.20	0.00	
sym10	10	837	1	0.50	0.30	1.40	25.8
t481	16	481	1	0.70	0.20	19.70	2527.3
tcon	17	32	16	1.00	0.20	0.00	
term1	34	257	10	5.30	0.30	24.60	
ti	47	213	72	61.30	0.70	7.10	
tial	14	586	8	3.80	0.30	3.00	855.6
too_large	38	1075	3	424.80	1.20		
ttt2	24	149	21	5.20	0.20	0.00	
unreg	36	48	16	3.10	0.20	0.00	
vda	17	793	39		0.30	0.30	
vg2	25	110	8	4.30	0.20	4.70	
wim	4	9	7	0.70	0.10	0.00	0.0
x1	51	324	35	13.80	0.40	6.40	
x1dn	27	110	6	3.70	0.20	2.90	
x2	10	35	7	1.30	0.20	0.00	5.4
x2dn	82	105	56	6.10	0.30	0.10	
x3	135	739	99		1.10	14.50	
x4	94	535	71		0.60	2.10	
x6dn	39	82	5	5.50	0.20	0.60	
x7dn	66	539	15	12.60	0.50	11.90	
x9dn	27	120	7	4.60	0.20	4.20	
xor5	5	16	1	0.20	0.10	0.00	0.0
z4	7	59	4	0.70	0.10	0.00	0.4
z4ml	7	59	4	0.80	0.2	0.00	0.4
AVG:				24.52	0.72	43.33	98.90

References

[1] S. Akers. Binary decision diagrams. *IEEE Transaction on Computers*, vol. C-27(no. 6):509–516, June 1978.

[2] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, vol. C-35(no. 8):677–691, August 1986.

[3] 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 *Proceedings of the 30th ACM/IEEE Design Automation Conference*, 1993.

[4] S. Devadas, D. Varma, and E. A. Trachtenberg. Design automation tools for efficient implementation of logic functions by decomposition. *IEEE Transactions on Computer Aided Design*, vol. 8(no. 8), August 1989.

[5] Bogdan Falkowski, Ingo Schaefer, and Marek A. Perkowski. Calculation of the rademacher-walsh spectrum from a reduced representation of boolean functions. In *Proceedings of the IEEE/ACM European Design Automation Conference*, 1992.

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

[7] M. Karpovsky. *Finite Orthogonal Series in the Design of Digital Devices*. John Wiley and Son, 1976.

[8] Robert Lisanke. Logic synthesis and optimization benchmarks user guide version 2.0, 1988. a report to provide documentation for benchmark examples used in conjunction with the 1989 MCNC International Workshop on Logic Synthesis.

[9] J. E. Rice, J. C. Muzio, and M. Serra. The use of autocorrelation coefficients for variable ordering for robdds. In *Proceedings of the 4th International Workshop on Applications of the Reed-Müller Expansion in Circuit Design*, 1999.

[10] F. Somenzi. Cudd: CU decision diagram package. C implementation of decision diagrams, version 2.3.0.

[11] R. Stanković, T. Sasao, and C. Moraga. Spectral transforms decision diagrams. In *Proceedings of the 2nd International Workshop on Applications of the Reed-Müller Expansion in Circuit Design*, 1995.

[12] ed. T. Sasao. *Logic Synthesis and Optimization*. Kluwer Academics, 1993. Chapter 10, Efficient Spectral Techniques for Logic Synthesis by D. Varma and E. A. Trachtenberg.

[13] Randal Tomczuk. Autocorrelation and decomposition methods in combinational logic design, 1996. PhD dissertation.