Minimization of Average Path Length in BDDs by Variable Reordering

Shinobu NAGAYAMA[1] Alan MISHCHENKO[2] Tsutomu SASAO[1,3] Jon T. BUTLER[4]

[1] Department of CSE, Kyushu Institute of Technology, Iizuka 820-8502, Japan
[2] Department of EECS, UC Berkeley, Berkeley CA 94720, U.S.A.
[3] Center for Microelectronic Systems, Kyushu Institute of Technology, Iizuka 820-8502, Japan
[4] Department of ECE, Naval Postgraduate School, Monterey, CA 93943-5121, U.S.A.

Abstract

Minimizing the Average Path Length (APL) in a BDD reduces the time needed to evaluate the Boolean function represented by the BDD. This paper describes an efficient APL minimization heuristic based on BDD variable reordering. The reordering algorithm is similar to classical variable sifting with the cost function equal to the APL rather than the number of BDD nodes. The main contribution of this paper is a fast way of updating the APL during the swap of two adjacent variables. Experimental results show that the proposed heuristic effectively minimizes the APL of large MCNC benchmark functions, achieving reductions of up to 47%. For some benchmarks, minimizing APL also reduces the BDD node count.

1. Introduction

Binary Decision Diagrams (BDDs) [1] are used to represent logic functions in various applications encountered in logic synthesis, logic simulation, and formal verification. In those applications that use a BDD to evaluate logic functions, the evaluation time is proportional to the Average Path Length (APL) in the BDD. Therefore, minimization of the APL leads to faster evaluation of the logic function. Particularly, in logic simulation using decision diagrams [6], minimization of the APL reduces the simulation time substantially because a logic function is evaluated again and again with different test vectors.

The minimization of the APL can also be applied in logic synthesis. Some methods for functional decomposition [14] use BDDs to detect Boolean divisors. The quality of a divisor is measured by the amount of don't-cares it provides for the minimization of the quotient. The don't-cares are generated by the paths in the BDD that lead to the terminal nodes. The shorter the paths, the more don't-care minterms they contain. Therefore, minimizing the APL in BDDs representing logic functions can improve the quality of decomposition.

The only known method to minimize the APL of a BDD [8] first minimizes the BDD for the number of nodes [5, 7] followed by applying incremental transformations to reduce the APL. Experimental results show that, for most benchmark functions, the ordering that results in the minimal APL differs from the ordering that results in the minimal number of BDD nodes. For some benchmarks, such as *cordic*, the minimal-APL ordering leads to a BDD that has twice the nodes of the minimal-node BDD. This observation suggests that the BDD reordered to minimize the nodes may not be a good starting point for the APL minimization.

In this paper, we develop a variable reordering algorithm, that minimizes the APL rather than the number of nodes. The proposed algorithm is similar to variable sifting [9]. It performs a series of swaps among pairs of adjacent variables, trying to minimize the cost function defined as the APL of the BDD. A part of the algorithm that has a significant effect on computation time is updating the APL after swapping each pair of the adjacent variables. The APL minimization algorithm of [8] does not provide an efficient solution to this problem, because the APL is computed by performing a traversal of the BDD. The traversal is required after the swap of each variable pair, which significantly slows the process of variable ordering.

The main contribution of this paper is a fast method to update the APL of the BDD after two adjacent variables have been swapped. This method is integrated into the swapping algorithm in such a way that there is no need to perform additional traversals of the BDD. The APL is updated ”on the fly”, as the BDDs nodes are being swapped. This explains why the proposed variable reordering algorithm is fast.

The rest of the paper is organized as follow. Section 2 contains the necessary terminology and definitions. Section 3 introduces a method to obtain statically the initial ordering of the variables. In Section 4, we propose the method to calculate the APL after the swap of two variables. Section 5 explains the implementation of the method described in Section 4. In Section 6, we propose a pruning technique to speed up the sifting algorithm. And, in Section 7, we show the efficiency of our method using benchmark functions.

2 Preliminaries

We assume that the reader is familiar with the basic terminology of Binary Decision Diagrams (BDDs) [1]. In particular, we consider the Reduced Ordered Binary Decision Diagram (ROBDD) derived from decision trees by removing redundant nodes and merging isomorphic subgraphs.

Definition 2.1 *The* **average path length** *or* **APL** *of a BDD is the sum of path lengths over all assignments of values to the variables divided by the number of assignments,* 2^n*.*

We are concerned with a traverse of the BDD, beginning at the root node and ending of a terminal node.

Definition 2.2 *The* **node traversing probability***, denoted by* $P(v_i)$*, is the fraction of all* 2^n *assignments of values to the variables whose path includes node* v_i*.*

Definition 2.3 *The* **edge traversing probability***, denoted by* $P(e_{i_0})$ *(or* $P(e_{i_1})$*), is the fraction of all* 2^n *assignments of values to the variables whose path includes* e_{i_0} *(or* e_{i_1}*), where* e_{i_0} *(or* e_{i_1}*) denotes the 0-edge (or the 1-edge) directed from away node* v_i*.*

Since all paths include the root node, this node is traversed with probability 1.0. Since all assignments to values of variables are equally likely, we have the following relation:

$$\frac{P(v_i)}{2} = P(e_{i_0}) = P(e_{i_1}).$$

Lemma 2.1 *[11] The node traversing probability on node* v_i *is the sum of the edge traversing probabilities of all incoming edges to* v_i*.*

Theorem 2.1 *[11] The APL is equal to the sum of the node traversing probabilities of the non-terminal nodes.*

That is,

$$APL = \sum_{i=0}^{N-1} P(v_i),$$

where N denotes the number of non-terminal nodes.

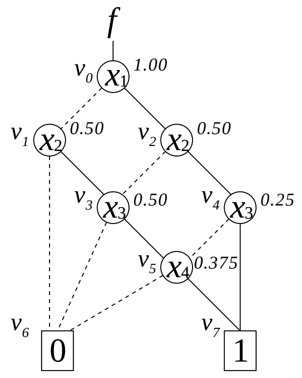


Figure 2.1. Example of node traversing probability in a BDD.

Example 2.1 *Consider the BDD in Fig. 2.1. First, we have* $P(v_0) = 1.00$*, and then we calculate* $P(v_1) = P(e_{0_0}) = 0.50$ *and* $P(v_2) = P(e_{0_1}) = 0.50$*. Similarly, we calculate*

$$\begin{aligned} P(v_3) &= P(e_{1_1}) + P(e_{2_0}) = 0.50, \\ P(v_4) &= P(e_{2_1}) = 0.25, \\ P(v_5) &= P(e_{3_1}) + P(e_{4_0}) = 0.375. \end{aligned}$$

Thus,

$$APL = \sum_{i=0}^{5} P(v_i) = 3.125.$$

(End of Example)

3 Initial Ordering of the Variables

The efficiency of APL minimization depends on finding an initial ordering that yields a reasonably small APL. An analysis of orderings that produces the minimal APL for known function classes [10] leads to a heuristic to find a good initial variable order. The value of a first-order Walsh spectral coefficient expresses the correlation between the variable value with the function value. Given a function $f(x)$ and a variable $x_i \in X$, the first order coefficient can be computed as follows:

$$R_i = \frac{(|\bar{x}_i \oplus f| - |x_i \oplus f|)}{2^n}.$$

Here, $|g(X)|$ means the number of assignments of values to the variables X such that $g(X) = 1$. R_i is just the number of agreements between the value of x_i and the value of $f(x)$ less the number of disagreements divided by the total number of assignments of values to variables. R_i is known as a first order spectral coefficient of $f(X)$ [3] and is related to the Chow parameters of the function [2]. For example, if

$f(X) = x_i$, then $R_i = 1$, corresponding to complete correlation and if $f(X) = \bar{x}_i$, then $R_i = -1$, corresponding to complete anti-correlation.

All spectral coefficients of a completely specified Boolean function can be computed by scanning the nodes beginning at the root node and ending on the terminal nodes using a fast algorithm [12]. The first-order coefficients can be computed by a simplified version of the general algorithm.

When the spectral coefficients R_i are known, an initial variable order is found by placing the variables in the decreasing order of the absolute values of the corresponding first-order spectral coefficients. For variables with identical absolute values, we arbitrarily choose the given ordering.

4 Change of APL During Variable Swap

Fig. 4.1 illustrates the BDD when two adjacent variables are interchanged. There are six cases, all shown in Fig. 4.1. In each case, the figure on the left occurs before the interchange, while the figure on the right occurs as a result of the interchange.

Note that these partial BDDs apply only to arcs incident to the root node. For example, for case a) prior to the interchange, there are also arcs incident to the two daughter nodes, each labeled x_{i+1}. For each node, case f) applies, creating for each a single node at the higher of the two levels. Case a) is the most general. Other cases occur when the functions at the nodes are independent of some variables.

It can be observed from Fig. 4.1 that only cases b) and c) change the APL of the BDD. This change is caused by merging some disjoint paths going through F_3 in case b), or splitting some disjoint paths going through F_3 in case c).

In Fig. 4.1 b), we say that one of the two nodes labeled x_{i+1} **disappears** as a result of the variable exchange. Similarly, in Fig. 4.1 c), we say that an additional node labeled x_i **appears** as a result of the variable exchange.

Suppose P_d is the sum of node traversing probabilities of nodes that disappear during the variable swap of two adjacent variables. That is, P_d is the sum of node traversing probabilities of nodes that are reduced in case b). And, suppose P_a is the sum of node traversing probabilities of nodes that appear during the variable swap. That is, P_a is the sum of node traversing probabilities of nodes that are inserted in case c). Then, the reduction in the APL for the BDD by interchanging x_i and x_{i+1} is equal to $P_d - P_a$. Note that $P_d - P_a$ can be negative, in which case, the interchange of x_i and x_{i+1} increases the APL. Thus, we can calculate the change in the APL during the variable swap by considering only the BDD nodes involved in the swap.

Note that, in the above discussion and in Fig. 4.1, we consider only the node traversing probability of a node on the lower level due to the incoming edges from nodes on the upper level. In general, a node on the lower level is incident to arcs from several upper level nodes. During the variable ordering, we consider all the subgraphs, one by one, and therefore the computed difference P_d and P_a accounts for all possible paths going through each node on the lower level. Also, there may be case b) on the same level as case c). They may cancel each other depending on the node traversing probabilities.

5 Implementation Issues

The change in the APL by swapping two adjacent variables can be computed by summing the node traversing probabilities of the lower-level nodes that appear and disappear in the BDD during the swap.

To compute the reduction in the APL, two double-precision floating point registers P_d and P_a are used. The registers are set to 0.0 before the swap. During the swap, when situations b) and c) occur, the registers are increased by the node traversing probability of the corresponding lower-level nodes that have disappeared and appeared, respectively. This node traversing probability is equal to the edge traversing probability of the edge from the upper-level node pointing to the lower-level node.

After the swap, the values of the two registers P_d and P_a are used to form $P_d - P_a$ and to update the APL of the BDD. The APL represents the cost function during the modified sifting procedure, similar to how the total node count is used during the classical sifting algorithm, which minimizes the number of BDD nodes. During sifting, all positions of the given variable are examined, and finally the variable is moved to the position corresponding to the minimum value of the APL.

Example 5.1 *Consider the sifting of BDD in Fig. 5.1(a). First, we calculate the APL of this BDD, which is* 2.25*. We seek the best position for* x_1*, and we begin by swapping* x_3 *and* x_1*. During this swap, case d) and case f) in Fig. 4.1 occur (see Fig. 5.1(b)). Hence, the APL of BDD does not change. And, we have the BDD in Fig. 5.1(c).*

Next, we swap x_2 *and* x_1*. During this swap, since case b) takes place, we increase register* P_d*:* $\Delta P_d = P(e_{v_0})$*, where* $P(e_{v_0})$ *is the edge traversing probability of 0-edge from the root node for the BDD, that is* 0.5*. Then, the APL of BDD is updated by using the registers* P_d *and* P_a*:* $APL_{new} = APL_{old} - (P_d - P_a)$*. Therefore, we have the BDD shown in Fig. 5.1(d) where* $APL = 1.75$*, and then we accept* x_1 *as the root node because this position minimizes the cost function.* *(End of Example)*

The algorithm executes in two rounds. In both rounds, each variable is sifted across all possible positions, except that certain extreme positions can be eliminated because they do not yield a minimum APL (discussed later). Within

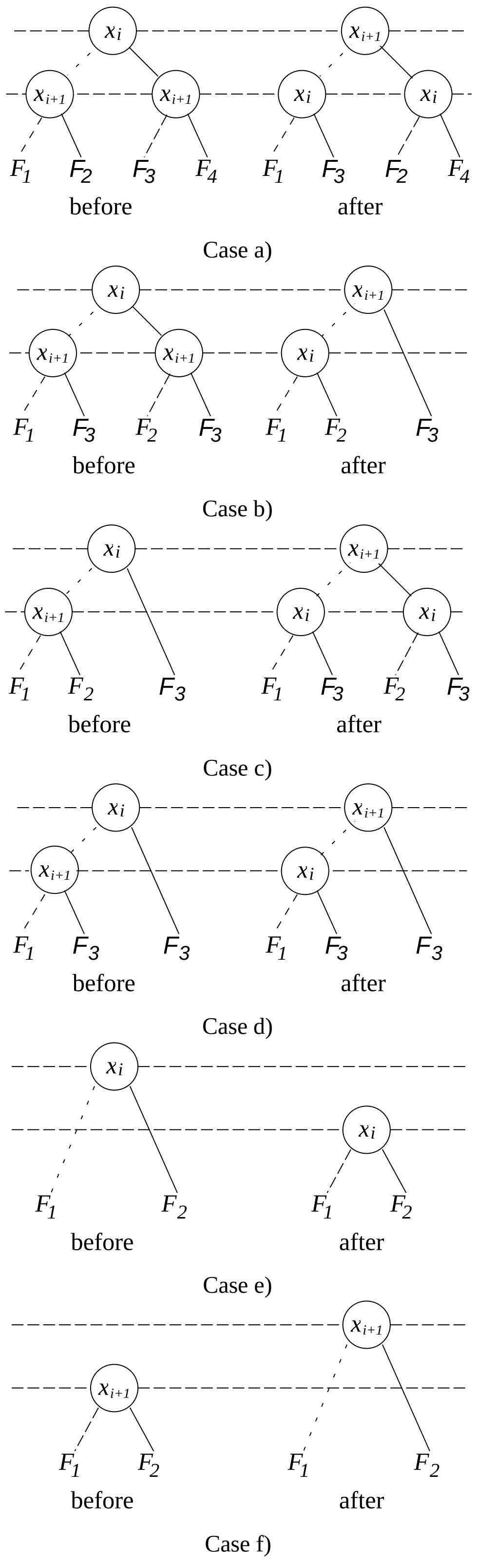
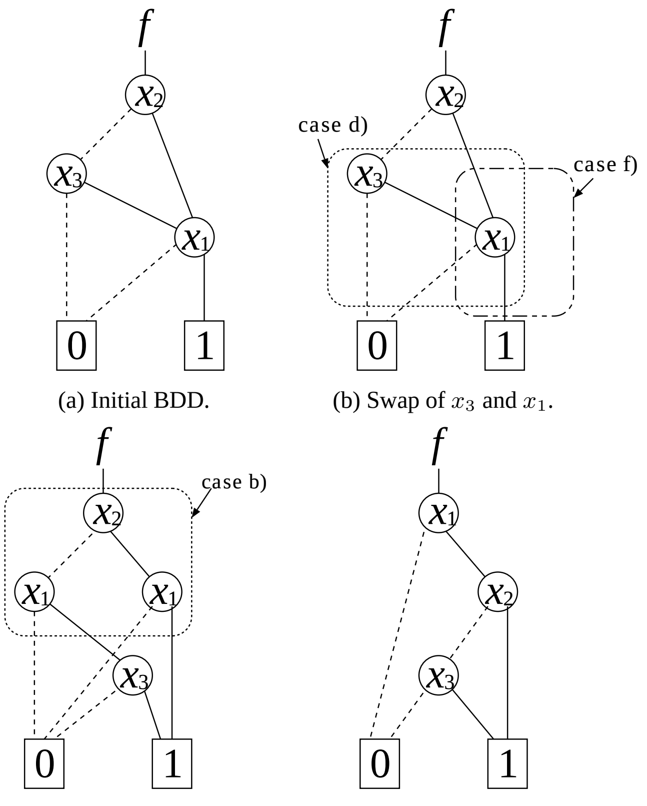


Figure 4.1. Exchanging two adjacent variables during BDD variable reordering.



(a) Initial BDD. (b) Swap of x_3 and x_1.

(c) Swap of x_2 and x_1. (d) Final BDD.

Figure 5.1. Example of sifting on BDD.

each round, the next variable to be sifted is chosen as the one labeling the maximum number of nodes. Two rounds are chosen because a third round rarely improved the APL.

Because the algorithm interchanges two adjacent variables, in order to insert the variable in all positions, it is necessary to first scan in one direction to an extreme, then scan in the opposite direction, returning to the starting position, and to continue scanning to the other extreme. The algorithm chooses first to scan toward the closer of the two extremes.

6 Lower Bound on APL During Sifting

This section describes an efficient lower bound that can be used to prevent useless swaps. The lower bound for the APL is similar to the one introduced for the number of BDD nodes during the classical sifting [4].

Theorem 6.1 *Fix the position of all variables except x. Let S be the smallest value of the APL achieved during sifting of variable x. Let variable x be currently on level i and the total sum of node traversing probabilities of all nodes in the BDD above the level i be U_i. Then, if $U_i \geq S$, sifting of* *variable x further down to levels $i+1$, $i+2$, etc., cannot reduce the cost function below S.*

(Proof) Let the total of the node traversing probabilities of all nodes in the BDD be T (i.e., APL of the BDD). Then, the following relation clearly holds;

$$T \geq U_i.$$

Sifting variable x further down to level $i+1$, $i+2$, etc., does not change U_i i.e., $P(v)$ remains unchange for any node v above level i. Therefore, we have the theorem. (Q.E.D)

A similar theorem can be formulated for the case when variable x is moved up.

The following discussion provides the rationale for the lower bound theorem. In sifting, the position of all variables in the BDD is fixed, except for variable x. Suppose x is moved down toward the bottom of the BDD, and the APL decreases to S and then increases. At some point, the sum of the node traversing probabilities of all nodes above x becomes equal to U, such that U is larger than S, the minimum APL achieved earlier. In this case, we can stop the sifting, since the APL of the BDD cannot be reduced below the limit set by the best position found so far, even if variable x is moved all the way down. It is also possible to show that a similar lower bound works for the case when the variable is moved up in the BDD variable order.

Using the lower bound theorems introduced above, it is possible to limit the range of sifting for variable x. Experiments show that the lower bound typically speeds up the computation by 30-50%.

7 Experimental Results

An experiment using MCNC benchmarks was conducted in the following environment:

- CPU: Pentium 4 Xeon 2.8GHz
- L1 Cache: 20KB
- L2 Cache: 512KB
- Main Memory: 4GB
- Operating System: Redhat (Linux 7.3)
- C-Compiler: gcc -O2

Table 7.1 compares the proposed algorithm for APL minimization with a previously published algorithm [8]. Benchmark functions are selected to be compatible with [8] except for incompletely specified functions. Each output of the multi-output benchmark functions is reordered independently, and the value reported is the sum of the APL over all outputs.

Table 7.2 shows the results for larger MCNC benchmarks. In this case, reordering was applied to the shared BDDs. Note that, in all experiments, the BDDs have complemented edges. Two rounds of sifting are performed in all experiments.

In the tables, *Name* denotes the benchmark function name, *In* and *Out* denote the number of inputs and outputs, respectively. In Table 7.1, the column "Results from [8]" shows the results reported in [8], and the column "Our results" shows the results obtained by the proposed sifting algorithm that minimizes the APL. Note that, in our results, initial variable orderings for BDDs are obtained by the static ordering algorithm described in Section 3. In Table 7.2, the column "Min_node BDD" shows the number of nodes and the APL for BDDs obtained by the sifting algorithm, which minimizes the number of BDD nodes. The column "Without static ordering" shows the results of the proposed sifting algorithm, which minimizes the APL, where initial variable order is the variable order of BDD in "Min_node BDD". And, the column "With static ordering" shows the results of the proposed sifting algorithm, where initial variable ordering is obtained by static ordering in Section 3. The column "Coef. Time" denotes the CPU time needed to calculate the coefficients R_i in Section 3. Unfortunately, for *C2670*, *C5315*, and *C7552*, BDDs with the initial variable ordering could not be constructed due to memory overflow. *Time* in the table denotes the CPU time needed to perform the corresponding reordering. This time does not include the time for reading the original benchmark from file. In Table 7.1, the row labeled *Average of ratios* represents a standardized average of the *Nodes* and *APL*, with the values of [8] standardized to 1.00. The "Our Results" column is a value relative to the results of [8]. In Table 7.2, the row labeled *Average* represents a standardized average of the *Nodes* and *APL*, with the values of "Min_node BDD" standardized to 1.00. The "Without static ordering" column and the "With static ordering" column are a value relative to the results for "Min_node BDD".

Table 7.1 shows that our proposed sifting algorithm improves the APL in 11 of the 17 benchmark functions considered in [8], yields the same APL for 5 of the remaining functions and yields a larger APL for one function. Further, it improves the number of nodes in 16 of 17 functions and yields the same number for one function. Especially, for *cordic*, both the number of nodes and the APL of our results are much smaller than [8]. Note especially that the computation time is small.

Table 7.2 shows that our algorithm performs well on large benchmark functions. For some of them, for example, *C1908*, *frg2*, and *rot*, the APL is reduced drastically. For *C7552*, the number of nodes is reduced as a byproduct of the APL minimization. However, for most functions, the number of nodes is increased by the sifting for

Table 7.1. Minimization of APL for individual BDDs

Name	In	Out	Results from [8]		Our results		
			Nodes	APL	Nodes	APL	Time, s
5xp1	7	10	91	31.31	79	31.28	0.01
alu4	14	8	899	47.54	516	39.97	0.01
b12	15	9	81	22.22	71	21.88	0.01
con1	7	2	16	6.06	16	5.94	0.01
cordic	23	2	259	11.82	88	9.47	0.01
sao2	10	4	128	10.71	121	10.59	0.01
vg2	25	8	230	30.37	204	30.16	0.01
misex1	8	7	68	22.16	64	21.97	0.01
cm150a	21	1	33	3.50	32	3.50	0.01
cm151a	12	2	36	6.50	32	6.00	0.01
cm162a	14	5	59	11.70	48	11.71	0.01
cm163a	16	5	42	11.70	36	11.70	0.01
cm85a	11	3	47	8.28	38	7.72	0.01
mux	21	1	33	3.50	32	3.50	0.01
z4ml	7	4	32	17.13	28	16.38	0.01
f51m	8	8	76	27.45	64	27.45	0.01
pcle	19	9	89	22.50	79	22.50	0.01
Average of ratios			1.00	1.00	0.84	0.96	–

the APL minimization. This shows that the minimization of the APL tends to be independent of the minimization of BDD nodes. The comparison of "Without static ordering" and "With static ordering" shows that the minimization of APL depends on the initial variable order. For 8 benchmark functions (*C432, C880, C1908, apex3, des, frg2, k2,* and *rot*), the APLs obtained by using static ordering in Section 3 are smaller than ones in "Without static ordering" column. However, for most functions, the computation time of sifting with static ordering is significantly longer than that of sifting without static ordering. The reason for this is the large size of initial BDDs. Swapping one pair of adjacent variables takes longer time because the time needed for the swap is roughly proportional to the number of nodes located in the BDD on the given levels.

8 Conclusions

This paper shows a fast way of updating the Average Path Length (APL) in the BDD during the swap of two variables adjacent in the variable order. Fast updating of the APL is used to create a specialized variable reordering algorithm for the heuristic minimization of the APL. The proposed algorithm is similar to the classical BDD variable sifting, except that the cost function used is the APL, instead of the number of BDD nodes. The proposed sifting algorithm processes the largest multi-output MCNC benchmark functions in reasonable time, while achieving a substantial reduction (up to 47%) in the APL. Our experiments also show that, for some benchmark functions, the number of nodes is reduced as a byproduct of the APL minimization.

Acknowledgments

This research is partly supported by the Grant in Aid for Scientific Research of The Japan Society for the Promotion of Science (JSPS), and the Takeda Foundation.

References

[1] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comp.*, Vol. C-35, No. 8, pp. 677–691, Aug. 1986.

[2] C. K. Chow, "On the characterization of threshold functions," in *Proc. IEEE Symp. Switching Theory and Logic Design*, pp. 34–38, 1961.

[3] M. L. Dertouzos, "*Threshold Logic: A Synthesis Approach*," Mass. Inst. Tech., Cambridge, Res. Monograph 32. Cambridge, Mass.: M. I. Press, 1965.

[4] R. Drechsler, W. Günther, and F. Somenzi, "Using lower bounds during dynamic BDD minimization," *IEEE Trans. CAD*, Vol. 20 (1), pp. 51–57, Jan. 2001.

[5] M. Fujita, Y. Matsunaga, and T. Kakuda, "On variable ordering of binary decision diagrams for the application of multi-level logic synthesis,“ *EDAC*, pp. 50–54, Mar. 1991.

[6] Y. Iguchi, T. Sasao, M. Matsuura, and A. Iseno "A hardware simulation engine based on decision diagrams," *Asia and South Pacific Design Automation Conference (ASP-DAC'2000)*, Yokohama, Japan, Jan. 26-28, 2000.

Table 7.2. Minimization of APL for shared BDDs for larger functions

Name	In	Out	Min_node BDD		Coef.	Without static ordering			With static ordering		
			Nodes	APL	Time, s	Nodes	APL	Time, s	Nodes	APL	Time, s
C432	36	7	1063	86.58	0.01	1081	86.24	0.15	1899	82.09	0.83
C499	41	32	25873	782.66	0.02	32105	641.16	7.12	32105	641.16	7.11
C880	60	26	4122	140.42	0.01	41701	123.85	4.48	91767	122.22	52.12
C1908	33	25	5532	254.65	0.01	16634	179.20	0.96	13868	171.96	2.73
C2670	233	140	1882	303.34	0.05	2755	278.17	1.30	–	–	–
C3540	50	22	24231	209.15	0.10	25162	208.44	7.44	56898	212.73	75.21
C5315	178	123	1728	460.78	0.05	1820	446.26	0.26	–	–	–
C7552	207	108	2212	485.03	0.05	2207	471.54	0.87	–	–	–
apex3	54	50	931	188.58	0.01	900	158.82	0.04	905	158.73	0.03
apex7	49	37	242	113.88	0.01	277	82.44	0.01	280	82.45	0.02
b9	41	21	108	61.16	0.01	131	55.25	0.01	129	55.39	0.01
dalu	75	16	688	102.67	0.01	990	78.81	0.08	1069	78.81	35.31
des	256	245	3297	1209.50	0.18	3343	1081.13	0.47	3886	1077.63	2.15
duke2	22	29	360	87.89	0.01	386	77.52	0.01	392	77.52	0.02
e64	65	65	128	128.00	0.01	128	128.00	0.01	573	128.00	0.05
ex4	128	28	497	51.38	0.01	629	47.26	0.02	630	47.26	0.03
frg2	143	139	1379	607.00	0.04	1580	322.89	0.15	2189	321.75	0.23
k2	45	45	1257	181.80	0.01	1426	177.52	0.07	1418	177.50	0.10
rot	135	107	7891	446.47	0.05	16164	312.08	5.61	18503	308.68	30.34
Average			1.00	1.00	–	1.87	0.85	1.53	3.0l	0.84	12.89

[7] N. Ishiura, H. Sawada, and S. Yajima, “Minimization of binary decision diagrams based on exchanges of variables,“ *ICCAD*, pp. 472–475, Nov. 1991.

[8] Y. Y. Liu, K. H. Wang, T. T. Hwang, C. L. Liu, “Binary decision diagrams with minimum expected path length,“ *Proc. DATE 01*, pp. 708–712, Mar. 13-16, 2001.

[9] R. Rudell, “Dynamic variable ordering for ordered binary decision diagrams,” *ICCAD ’93*, pp. 42–47.

[10] T. Sasao, J. T. Butler, and M. Matsuura, “Average path length as a paradigm for the fast evaluation of functions represented by binary decision diagrams,” *International Symposium on New Paradigm VLSI Computing*, Sendai, Japan, Dec. 12-14, 2002.

[11] T. Sasao, Y. Iguchi, and M. Matsuura, “Comparison of decision diagrams for multiple-output logic functions,” *International Workshop on Logic and Synthesis*, New Orleans, Louisiana, June 4-7, 2002, pp.379-384.

[12] M. Thornton, D. M. Miller, and R. Drechsler, “Transformations amongst the Walsh, Haar, arithmetic and Reed-Muller spectral domains,” *Proc. Intl. Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, August 10-11, 2001, pp. 215-225.

[13] S. Yang, *Logic synthesis and optimization benchmark user guide version 3.0*, MCNC, Jan. 1991.

[14] C. Yang and M. Ciesielski, “BDS: A BDD-based logic optimization system,” *IEEE Trans. CAD*, Vol. 21 (7), July 2002, pp. 866-876.