Linear Sifting of Decision Diagrams*†

Christoph Meinel
FB IV – Informatik, Universität Trier
D – 54286 Trier, Germany

Fabio Somenzi
Dept. of Electrical and Computer Engineering
University of Colorado at Boulder

Thorsten Theobald
FB IV – Informatik, Universität Trier
D – 54286 Trier, Germany

Abstract

We propose a new algorithm, called *linear sifting*, for the optimization of decision diagrams that combines the efficiency of sifting and the power of linear transformations. We show that the new algorithm is applicable to large examples, and that in many cases it leads to substantially more compact diagrams when compared to simple variable reordering. We show in what sense linear transformations complement variable reordering, and we discuss applications of the new technique to synthesis and verification.

1 Introduction

Binary decision diagrams [6] are widely used in formal verification and synthesis algorithms because they provide compact representation of logic functions. Successful application to complex problems, however, almost invariably entails resorting to various techniques to reduce the size of the diagrams that the programs must manipulate. Some techniques are lossy, and rely on several forms of approximations [9, 14, 8, 20].

Other techniques try to find a more compact representation without any loss of information. Variable reordering [11, 21] is the most widely known and applied reversible transformation for BDDs. Recent work [1, 10] has addressed more general classes of reversible—hence lossless—transformations. The common premise for these more powerful transformations is that any automorphism on the set of the boolean functions of n variables is, by definition, reversible.

In practice it is necessary to identify one automorphism that maps the functions of interest onto functions with simple representations, and whose representation is also compact. Since the exhaustive search of all possible automorphisms is prohibitive even for small problems, efficient heuristic algorithms are sought for the identification of appropriate transformations.

In a recent paper [16], two of us have studied linear transformations, concluding that they form a class very well suited to the implementation of efficient algorithms. In this paper we present one such algorithm and report on its experimental evaluation. Our new algorithm combines linear transformations with variable reordering techniques.

Linear transformations replace one variable, x_i, with a linear combination of variables. A linear combination is obtained by taking the exclusive or of the arguments or its complement. In this paper we restrict ourselves to transformations of the form

$$x_i \mapsto x_i \equiv x_j. \tag{1}$$

where $\equiv$ indicates the equivalence function, that is, the complement of the exclusive or.

Before we examine the reasons for this choice and enter into the details of the algorithm, we briefly review two applications that demonstrate the usefulness of this kind of transformation. In [7] it is shown that the reachability analysis of the product of two very similar sequential machines can be made more efficient if a transformation essentially identical to (1) is applied to the state variables of one of the two machines. A program that can apply linear transformations, can therefore automatically apply the technique of [7] and, in addition, take advantage of similarities between subsets of the two machines or even between different parts of one machine. In other words, the technique we present in this paper can be used to automatically re-encode finite state systems to make reachability analysis more efficient. Many aspects of this application are discussed in [16] and are therefore no further considered here.

In [12] a technique based on spectral analysis is presented, whose purpose is to decompose a circuit in the cascade of two subcircuits: The first is a linear block, composed of exclusive or gates fed by the primary inputs of the overall circuit. Such a decomposition simplifies the synthesis task—sometimes dramatically.

*This work was supported in part by NSF grant MIP-94-22268 and SRC contract 95-DJ-560. Thorsten Theobald was supported by DFG-Graduiertenkolleg "Mathematische Optimierung".

DAC 97, Anaheim, California
© 1997 ACM 0-89791-920-3/97/06 ..$3.50

This first block of the cascade corresponds to the re-encoded primary inputs for a suitable combination of linear transformations. The second block implements the image of the original function according to the chosen transformation. Our algorithm thus provides an alternative to the one proposed in [12]; it is applicable to larger circuits and deals quite naturally with multiple-output circuits.

The rest of this paper is organized as follows: In Section 2 we recall definitions relevant to decision diagrams and linear transformations. In Section 3 we discuss the linear sifting algorithm. In Section 4 we report on our experiments. Section 5 concludes the paper.

2 Preliminaries

2.1 Decision Diagrams

Binary decision diagrams (BDDs) [6] are rooted directed acyclic graphs representing switching functions. Each BDD has two sink nodes which are labeled 1 and 0. Each internal node is labeled by an input variable x_i and has two outgoing arcs, labeled 1 and 0. A total order π is placed on the input variables. The variable occurrences on each path of the BDD have to be consistent with this order.

A BDD computes a switching function $f : B^n \mapsto B$ in a natural manner: Each assignment to the input variables x_i defines a unique path through the graph from the root to a sink. The label of this sink defines the value of the function on that input. A BDD is *reduced* if it contains no isomorphic subgraphs, and every vertex has distinct children. Reduced, ordered BDDs are canonical representations of boolean functions: for a fixed variable order, two functions are equivalent if and only if their BDDs are identical.

In most current implementations, several techniques are used to increase the compactness of the representation and the efficiency of the algorithms [2]. For instance, several functions can be represented by a multi-rooted graph. This allows one to check for equivalence in constant time. Another device commonly employed, with which we shall concern ourselves, is the *complement arc*. The complement arc is an arc in the BDD with an attribute set to indicate that the function pointed by the arc should be complemented to yield the desired function. To preserve canonicity, the arcs labeled 1 cannot carry the complement attribute.

The size of BDDs depends, sometimes critically, on the variable order π. Various algorithms have been proposed in recent years to determine good variable orders. The one that is most important to our treatment is Rudell's *sifting* algorithm [21]. Sifting is a local search algorithm that iteratively improves the variable order by a series of swaps of adjacent variables.

Each variable is considered in turn and is moved up and down in the order so as to take all positions successively. The variable is then returned to one position where the minimum size of the BDD was recorded. The process then continues with another variable.

The effectiveness of sifting stems from its ability to move a variable to any position in the order in a short time. Its time efficiency relies on the ability to quickly swap adjacent variables. It is indeed possible to perform such a swap by accessing only the nodes labeled by the two variables being exchanged.

2.2 Linear Transformations

Let $B = \{0, 1\}$ and let $\mathbb{B}_n$ denote the boolean algebra of the functions over B^n. An automorphism τ on B^n (a bijective mapping from B^n onto itself) induces an automorphism on $\mathbb{B}_n$, ϕ, with $\phi(f)(a) = f(\tau(a))$. This function automorphism is completely characterized by the images of the projection functions, $x_1, \ldots, x_n$. We define linear transformations as those automorphisms in which the images of the projection functions are obtained by composing a sequence of elementary transformations of the form $x_i \mapsto x_i \equiv x_j$.

It is possible—and convenient in our context—to regard variable reordering as an automorphism. Quite naturally, the exchange of variables x_i and x_j in the order corresponds to the application of the transformation $x_i \mapsto x_j, x_j \mapsto x_i$.

The number of linear transformations is shown in [16] to be

$$\prod_{i=0}^{n-1} (2^n - 2^i).$$

For large n, the number of linear transformations is much smaller than the number of all possible automorphisms ($2^n!$), but much larger than the number of transformations corresponding to variable permutations ($n!$).

On the other hand, the more limited is a set of transformations, the easier it is to represent it. Indeed, variable permutations require $O(n)$ space; linear transformations take $O(n^2)$ space; while general automorphisms require exponential space in the worst case. (In practice, the space requirements of linear transformations are quite far from the worst case quadratic bound.)

Linear transformations therefore considerably enlarge the search space for the BDD optimization problem, when compared to variable reordering; still they enjoy efficient manipulation. Moreover, in the context of the applications mentioned in Section 1, linear transformations are even more highly desired because they form a key step in the solution of the problem.

3 Linear Sifting

Sifting (see Section 2.1) proceeds by a sequence of pairwise exchanges of variables. Suppose that variables x_i and x_j are to be swapped, and that x_i immediately precedes x_j before the swap. Then the effect of the exchange on each node labeled x_i can be easily seen by applying Boole's expansion theorem w.r.t. both x_i and x_j. Assuming f is the function of a node labeled x_i, we have:

$$f = x_i x_j f_{11} + x_i x_j' f_{10} + x_i' x_j f_{01} + x_i' x_j' f_{00}. \qquad (2)$$

Exchanging x_i and x_j and rearranging terms yields:

$$\phi(f) = x_i x_j f_{11} + x_i x_j' f_{01} + x_i' x_j f_{10} + x_i' x_j' f_{00}. \qquad (3)$$

$$\phi(f) = x_i x_j f_{11} + x_i x_j' f_{00} + x_i' x_j f_{01} + x_i' x_j' f_{10}. \qquad (4)$$

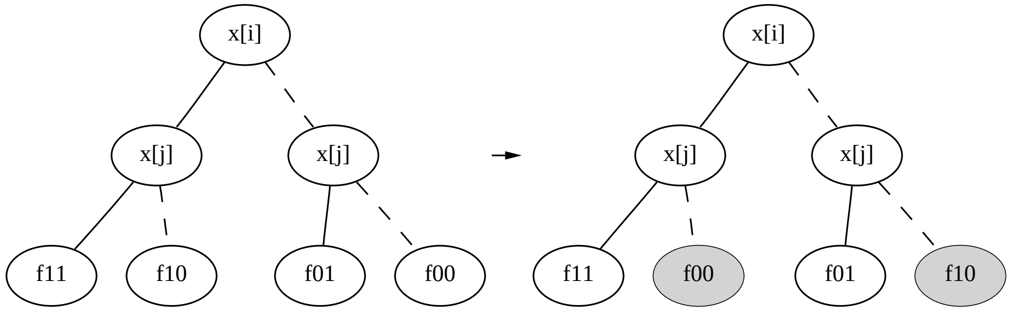

Figure 1: Linear transformations via local interchange.

In words, the effect of a swap is to interchange f_{10} and f_{01}. If we repeat the same process for the application of an elementary linear transformation (Equation 1), we obtain: A comparison of Equations 3-4 shows that the only difference is that f_{00} is involved in the interchange with f_{10}, instead of f_{01}. (See Figure 1.) The fundamental techniques applied to the efficient swapping of two variables can be used also for their linear combination.

Moreover, examination of Equations 2-4 shows that the application of both swapping and linear combination produces, if the initial state is included, all three ways of partitioning $\{f_{11}, f_{10}, f_{01}, f_{00}\}$ in two subsets of equal size. Therefore, by combining swapping and linear transformations it is possible to reduce the size of the BDD in more cases than by swapping only. This observation forms the basis for our linear sifting algorithm, which proceeds as follows.

Each variable is considered in turn, and, as in sifting, it is moved up and down in the order. Let x_i be the chosen variable, and let x_j be the variable immediately following it in the order. One basic step of linear sifting consists of the following three phases:

1. Variables x_i and x_j are swapped; let the size of the BDD after the swap be s_1.
2. The linear transformation $x_j \mapsto x_i \equiv x_j$ is applied; let the resulting size of the BDD be s_2.
3. If $s_1 \leq s_2$ then the linear transformation is undone. This is obtained by simply applying the transformation again, since it is its own inverse.

The net effect of the three-phase procedure is that x_i is moved one position onward in the order, and possibly linearly combined with x_j. Conversely, if x_j is the variable being moved, it is first swapped and then x_i is combined with it. ($x_j \mapsto x_i \equiv x_j$.)

The usual formulation of linear transformations is in terms of exclusive or, rather than its complement. We use the equivalence function ($\equiv$) because our BDDs use the complement arcs; hence, we cannot swap f_{11} with one of $\{f_{10}, f_{01}, f_{00}\}$, all of which could be complemented. Since the 1-arcs cannot be complemented, we would have to change the complementation of arcs into the node labeled (initially) x_i. This would make the operation non-local, which is highly undesirable. Nothing is lost by using the equivalence function instead of the exclusive or. It is shown in [16] that the combination of variable exchange and the transformation of Equation 1 subsumes all invertible transformations that affect only two variables.

An algorithm that reorders the variables of a BDD normally keeps track of the permutation produced. Likewise, our algorithm for linear sifting has to maintain the images of the projection functions (the transformed $x_1, \ldots, x_n$). The most expedient way to do that is to maintain the projection functions in the BDD package. Application of the transformations to the BDDs of the projection functions yield the BDDs of their transforms, which can be used to build functions directly in the transformed space. This minimizes the impact of linear sifting on the applications that use it. As an example, two combinational circuits may be tested for equivalence by building BDDs for both. By using the transformed BDDs of the primary inputs one guarantees that the two circuits are equivalent if and only if they have the same BDD. Functions like *Apply* [6] or *ITE* [2] are oblivious of linear sifting.

An important device to guarantee high speed in sifting is the *interaction matrix*. In standard sifting, this matrix tells the swapping procedure whether two variables appear together in the support of some function; in other words, whether there exists a root of the BDD from which one can reach nodes labeled by both variables. If two variables do not interact, the swap can be performed in constant time. The interaction matrix is initialized before reordering starts, and, when swapping is the only allowed transformation, it remains invariant throughout the execution of the sifting algorithm.

The interaction matrix is effective also for linear sifting. On the one hand, it allows cheap swaps for non-interacting variables. On the other hand, it is used to avoid linear combinations of variables that do not interact. In contrast to swapping operations, however, linearly combining variables may cause variables that did not interact to become interacting. Consider the case of two functions of three variables:

$$\begin{aligned} f &= x \oplus y \\ g &= y \equiv z. \end{aligned}$$

Variables x and z do not interact, because neither f nor g has both of them in its support. However, the transformation $y \mapsto x \equiv y$ yields

$$\begin{aligned} \phi(f) &= y' \\ \phi(g) &= x \oplus y \oplus z, \end{aligned}$$

where all variables interact. Therefore, when two variables are combined, the linear sifting algorithm updates the interaction matrix accordingly.

Linear sifting is slower than normal sifting in most cases, because the cost of a linear transformation is comparable to the cost of a variable swap. If we assume that the graphs manipulated by the two algorithms are approximately of the same size, then linear sifting will be approximately three times slower than standard sifting. This analysis is approximate and does not take into account, for instance, the effects of the linear transformations on the interaction matrix. However, it is confirmed by our experiments in most cases.

We close this section by recalling that several variants of decision diagrams exist. We have discussed linear sifting for BDDs, but the algorithm applies with null or minor modifications to all the diagrams that use Boole expansion (as opposed to Reed-Muller [13] or moment [5] expansions). In the case of Zero-Suppressed BDDs [18], the transformation $x_i \mapsto x_i \equiv x_j$ is not very effective in changing the size of the diagrams: A node is suppressed if its 1-arc points to the 0 sink node. This condition is not affected by swapping f_{10} and f_{00}. The graph size may still change as a result of increased or decreased sharing of nodes, but it is advisable to use $x_i \mapsto x_i \oplus x_j$ instead. The exclusive or transformation is also preferable for Edge-Valued BDDs [15]: In the typical case when the weight of the 0-arc is constrained to be 0, using $x_i \mapsto x_i \equiv x_j$ would lead to a non-local transformation.

4 Experiments

We have implemented linear sifting in the CUDD package [23] and applied it to the BDDs of a large collection of circuits. The collection includes well-known benchmark circuits [4, 3, 24], as well as several large industrial examples. Tables 1 and 2 (Table 2 continues Table 1) summarize our experiments, in which we compare the best results we have obtained by applying variable reordering (specifically, different variants of sifting [19]) to the best results obtained by combining normal sifting and linear sifting.

Two main strategies were used for linear sifting: One was to build the BDDs with a known good order and then applying linear sifting. The second strategy consisted of applying linear sifting also while building the BDDs, whenever their sizes doubled. The former strategy gave better results in more cases, but the latter produced the most striking results.

The reduction in size goes from 0% in 19 cases out of 77, to 98% for C499 and C1355. The cases of 0% improvement are mostly for small circuits. We left these small circuits in our test suite in spite of their limited significance, because we insisted on using the entire validation test suite of CUDD, so that no bias in favor or against linear sifting were introduced by the selection process.

The average improvement, measured by the geometric mean of the individual improvements, is 22%. The total number of nodes is decreased by 13% by the application of linear sifting. The CPU times should be considered as merely indicative and are shown for the sole purpose of supporting the claim that linear sifting is time efficient in practice. However, since different circuits give their best results when subjected to different methods, it is impossible to compare individual CPU times. More significant is the fact that for most cases the number of linear transformations applied is comparable to the number of swaps performed by the algorithm. When sifting and linear sifting produce graphs of almost equal sizes, sifting is about twice as fast. Improvements to the implementation of linear sifting should close this gap somewhat.

Though not all circuits equally benefit from the application of linear sifting (e.g., multipliers), in some cases like C499 or C1908 the reduction is such as to make a dramatic difference in the subsequent processing of the circuit. Indeed, it is not possible with tools like SIS [22] or even symbolic synthesis algorithms [17] to flatten the two aforementioned circuits. It is not even possible to synthesize effectively from a network of multiplexers representing the BDDs directly. However, if linear sifting is applied, flattening and synthesis from the BDDs become not only possible, but can be completed in seconds.

5 Conclusions

The optimization of decision diagrams has applications in both verification and synthesis. Whenever the cost of creating and optimizing the diagram is a minor fraction of the total elaboration cost; or whenever the optimality of the diagram may have a direct impact on the quality of the solution, the extra expense of time required by a more sophisticated algorithm may well be justified. We have presented in this paper linear sifting, which extends Rudell's variable reordering algorithm by combining it with linear transformations. We have shown that linear sifting reduces the combined size of the BDD and the transformation in most cases, with an average improvement of 22% in the number of nodes over a large set of experiments. Several applications can benefit from a more compact representation that preserves the ease of use and canonical properties of plain decision diagrams. We have outlined two of them: Reencoding for reachability analysis and logic decomposition for synthesis. We are currently working on the implementation of these techniques, and the initial results are quite encouraging.

name	in	ordering	linear sifting		
		size	size	ratio	CPU
9symml	9	25	25	1.00	0.1
alu4	14	350	350	1.00	0.1
t481	16	21	21	1.00	0.3
vda	17	485	472	0.97	0.2
cordic	23	42	29	0.69	0.1
s499	23	351	319	0.91	0.4
s344	24	104	83	0.80	0.1
ttt2	24	109	109	1.00	0.1
s1196	32	599	599	1.00	0.2
C1908	33	5708	1822	0.32	4.2
adder16	33	82	49	0.60	0.1
s635	34	128	128	1.00	0.1
C432	36	1209	1209	1.00	0.3
mm9b	38	1527	1476	0.97	1.0
too_large	38	319	319	1.00	0.5
mm9a	39	1112	972	0.87	0.3
C1355	41	25866	484	0.02	5.1
C499	41	25866	520	0.02	3.0
k2	45	1246	1235	0.99	1.2
s967	45	389	380	0.98	0.2
C3540	50	23828	23823	1.00	31.0
s1269	55	1810	1792	0.99	0.6
C880	60	4083	4080	1.00	3.8
comp32	64	97	97	1.00	0.1
adder32	65	162	97	0.60	0.4
adsb32	65	345	191	0.55	0.4
s938	66	161	161	1.00	0.3
sbc	68	917	917	1.00	0.5
alu32	73	478	388	0.81	0.4
alu32r	73	478	388	0.81	0.4
dalu	75	689	594	0.86	3.0
s991	84	707	707	1.00	0.6
s1512	86	572	476	0.83	1.4
xi30	90	91	91	1.00	0.3
s1423	91	1796	1477	0.82	3.1
dpath32	93	485	396	0.82	0.5
ellen_c	93	155432	154756	1.00	1203.4
ellen_t	109	442	442	1.00	0.3
M1	118	252153	153006	0.61	774.2

Table 1: Experimental results (first part). The column labeled *in* gives the number of inputs of the circuit. The column labeled *ordering size* gives the best results we have achieved by variable ordering only. Totals are shown at the bottom of Table 2.

name	in	ordering	linear sifting		
		size	size	ratio	CPU
mm30a	123	11065	9065	0.82	18.6
i3	132	133	133	1.00	0.2
i8	133	1276	1276	1.00	1.0
apex6	135	498	498	1.00	0.4
s3271	142	832	671	0.81	1.7
frg2	143	963	918	0.95	3.9
ind1sn	148	1796	1730	0.96	8.9
s4863	153	64088	61701	0.96	404.4
ind2sn	172	2989	2818	0.94	13.9
ind3sn	174	3201	3071	0.96	19.6
ellen_s	176	135007	134990	1.00	607.3
C5315	178	2104	1808	0.86	9.5
s5378	199	1932	1664	0.86	9.1
C7552	207	3730	1921	0.52	71.3
ellen_l	216	176022	170419	0.97	1010.1
s3384	226	710	597	0.84	0.8
C2670	233	2008	998	0.50	37.2
s9234.1	247	3045	2731	0.90	16.8
comp128	256	385	385	1.00	2.1
des	256	2945	2085	0.71	8.6
add128	257	642	385	0.60	10.4
adsb128	257	1401	767	0.55	9.9
i10	257	20660	19295	0.93	172.9
alu128	265	1918	1540	0.80	6.5
s6669	322	18043	17904	0.99	23.1
cps1364	328	20522	20510	1.00	109.7
OW	333	8842	5470	0.62	42.0
OW2	333	6684	4021	0.60	30.7
clma	415	738	715	0.97	3.4
clmb	415	690	618	0.90	5.2
dsip	452	3033	2594	0.86	14.0
bigkey	486	1595	1406	0.88	2.2
s15850.1	611	10111	8990	0.89	112.5
s13207.1	700	2941	2859	0.97	54.2
trainf	925	127987	124470	0.97	1103.6
s38584.1	1464	13303	12934	0.97	255.9
s38417	1664	266772	266404	1.00	2156.5
s35932	1763	5206	5168	0.99	162.5
Totals	16979	1436081	1250009	0.87	8562.9

Table 2: Experimental results (continued). See Table 1 for the legend.

References

[1] J. Bern, C. Meinel, and A. Slobodová. Efficient OBDD-based boolean manipulation in CAD beyond current limits. In *Proceedings of the Design Automation Conference*, pages 408–413, San Francisco, CA, June 1995.

[2] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th Design Automation Conference*, pages 40–45, Orlando, FL, June 1990.

[3] F. Brglez, D. Bryan, and K. Koźmiński. Combinational profiles of sequential benchmark circuits. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 1929–1934, Portland, OR, May 1989.

[4] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator in Fortran. In *Proceedings of the International Symposium on Circuits and Systems*, June 1985.

[5] R. Bryant and Y.-A. Chen. Verification of arithmetic circuits with binary moment diagrams. In *Proceedings of the Design Automation Conference*, pages 535–541, San Francisco, CA, June 1995.

[6] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[7] G. Cabodi, P. Camurati, F. Corno, S. Gai, P. Prinetto, and M. Sonza Reorda. A new model for improving symbolic product machine traversal. In *Proceedings of the Design Automation Conference*, pages 614–619, Anaheim, CA, June 1992.

[8] H. Cho, G. D. Hachtel, E. Macii, B. Plessier, and F. Somenzi. Algorithms for approximate FSM traversal based on state space decomposition. In *Proceedings of the Design Automation Conference*, pages 25–30, Dallas, TX, June 1993.

[9] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages*, January 1992.

[10] M. Fujita, Y. Kukimoto, and R. K. Brayton. BDD optimization by truth table permutations. Presented at IWLS95, Lake Tahoe, CA., May 1995.

[11] M. Fujita, Y. Matsunaga, and T. Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *Proceedings of the European Conference on Design Automation*, pages 50–54, Amsterdam, February 1991.

[12] J. P. Hansen and M. Sekine. Synthesis by spectral translation using boolean decision diagrams. In *Proceedings of the Design Automation Conference*, pages 248–253, Las Vegas, NV, June 1996.

[13] U. Kebschull, E. Schubert, and W. Rosenstiel. Multilevel logic synthesis based on functional decision diagrams. In *Proceedings of the European Conference on Design Automation*, pages 43–47, Brussels, March 1992.

[14] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton University Press, Princeton, NJ, 1994.

[15] Y.-T. Lai and S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Proceedings of the Design Automation Conference*, pages 608–613, Anaheim, CA, June 1992.

[16] C. Meinel and T. Theobald. Local encoding transformations for optimizing OBDD-representations of finite state machines. In *Proceedings of FMCAD*, pages 404–418, Palo Alto, CA, November 1996.

[17] S.-I. Minato. Fast weak-division method for implicit cube representation. In *SASIMI '93*, pages 423–432, Nara, Japan, October 1993.

[18] S.-I. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the Design Automation Conference*, pages 272–277, Dallas, TX, June 1993.

[19] S. Panda and F. Somenzi. Who are the variables in your neighborhood. In *Proceedings of the International Conference on Computer-Aided Design*, pages 74–77, San Jose, CA, November 1995.

[20] K. Ravi and F. Somenzi. High-density reachability analysis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 154–158, San Jose, CA, November 1995.

[21] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 42–47, Santa Clara, CA, November 1993.

[22] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. In *Proceedings of the International Conference on Computer Design*, pages 328–333, Cambridge, MA, October 1992.

[23] F. Somenzi. *CUDD: CU Decision Diagram Package.* ftp://vlsi.colorado.edu/pub/.

[24] S. Yang. Logic synthesis and optimization benchmarks user guide version 3.0. Technical report, Microelectronics Center of North Carolina, Research Triangle Park, NC, January 1991.