publIshed In
Proceedings of the 1998 IEEE/ACM Int. Workshop on Logic Synthesis (IWLS98),
Lake Tahoe, CA, 1998, pp. 346-352.

Binary Decision Diagrams and the Multiple Variable Order Problem

Gianpiero Cabodi[†] Stefano Quer[†]

Christoph Meinel[‡] Harald Sack[‡] Anna Slobodová[‡] Christian Stangier[‡]

[†] Politecnico di Torino
Dip. di Automatica e Informatica
Torino, Italy

[‡] FB IV – Informatik
Universität Trier
Trier, Germany

Abstract

Ordered Binary Decision Diagrams (OBDDs) are the first choice in manipulating and representing Boolean functions in CAD. Since the size of an OBDD heavily depends on the chosen variable order, much effort is spent in finding good and improving existing variable orders. If these optimizing techniques are used in OBDD applications, one has to cope with OBDDs of different variable orders very often (e.g., representing the transition relation and the reachable state set in sequential verification). For an efficient manipulation and represention of OBDDs as a multi rooted OBDD, a common variable order is desired.

In this paper we present approaches for the solution of the multiple variable order problem*, i.e., we present heuristics that compute a well suited intermediate variable order based on the original variable orders of the involved OBDDs.*

1 Introduction

Manipulation of Boolean functions is of the outermost importance in several areas of CAD for VLSI such as logic synthesis, verification, testing, etc. The efficiency of logic function manipulation depends on the data structure used for representing Boolean functions.

In the last few years Reduced Ordered Binary Decision Diagrams (ROBDDs or simply BDDs) have been intensively used for their efficiency. They constitute a canonical representation and they are compact for many functions encountered in practice. It is easy to implement operations on Boolean functions, relations, sets and graphs operating on BDDs.

A hard problem is to select some global variable ordering before generating BDDs. It has been proved that finding a suitable variable ordering is critical for efficiency. Poor orderings cause node explosion and exceeding the limits of physical memory. As an example the OBDD representation of the function $sum_n(a,b) = a + b$ computing the sum of two numbers a and b given by their binary representation is of exponential size if the natural order $(a_1, \ldots, a_n, b_1, \ldots, b_n)$ is chosen, but only of cubic size if the variables are grouped by their indices $(a_1, b_1, \ldots, a_n, b_n)$.

BDDs became popular with the development of heuristic methods for automatically generating an ordering based on information about the application. For example, given a gate level description of a circuit, heuristic methods can find a variable ordering for the primary inputs such that the primary outputs have compact BDDs. The ordering is *static*, in the sense that it is the same for all steps in the application. In practice, the ideal ordering may change as the application moves through different phases of computation. For example, the analysis of sequential circuits typically proceed through a series of phases and *dynamic* reordering allows BDDs to adapt to the changing functions as computation proceeds. As the application proceeds, a global, multi–rooted BDD is maintained representing all functions with a single variable ordering. Periodically the program attempts to reorder variables in this graph to reduce memory requirements. Dynamic reordering can slow down an application by a factor of 10 or more, but it often allows to complete an application that otherwise would have been aborted.

From one hand, dynamic reordering is mandatory, from the other one, applications have to cope with BDDs represented according to different variable ordering. For example, very often in reachability analysis the transition relation and reachable state sets have very different variable order requirements. During initial phases of the process, the size of the transition relation dominate the problem whereas reachable state set become predominant as their size grows larger. Moreover, partitioned BDDs in the field of reachability analysis and formal verification have been recently introduced [CCQ96, NJF+96, NIJ+97]. In these work the common framework is to partition, i.e., decompose, a BDD in sub–BDDs. In this way the application can deal with each BDD separately and optimize their size separately. Finally, it has been suggested [CCQ96] that BDD can be dumped on files and used as a communication mean among different tools (e.g., synthesis versus verification tools, or verification tools with different characteristics) and different working groups. In this situation it can be necessary to load a BDD generated in a different environment and with a different variable order. Memory and time requirements usually are lower but different variable order have to be coped with at the same moment.

Another related problem is testing the equivalence of two given OBDDs which do not share the same variable order. In [FHS78], a polynomial time algorithm for deciding this problem is given, but this algorithm does not provide the difference/equivalence of the OBDDs, i.e., the set of the input assignments on which the functions differ/agree that is desired in some applications.

In this paper we concentrate on the *multiple variable order* problem. In particular, we present methods for the generation of common variable orders for OBDDs to be represented simultaneously. We discuss the quality of the different approaches and show first experimental results. Unfortunately the most motivating experiments concerning transition relation and reach set are not finished yet. The paper is structured as follows. In Section 2 we give some basic definitions about OBDDs. Section 3 presents our new intermediate order strategies and Section 4 concludes the paper with experimental results.

2 Preliminaries

An *Ordered Binary Decision Diagram* (OBDD) P for a Boolean function $f : \mathcal{B}^n \to \mathcal{B}$ over the variables $X_n = \{x_1, \ldots, x_n\}$ is a directed acyclic graph consisting of inner nodes labeled by Boolean variables and sinks labeled by the Boolean constants 1 and 0. Each inner node has two outgoing edges: the 1-edge and the 0-edge. The OBDD has a starting node called root. The computation of $f(a_1, \ldots, a_n)$ with $a_i \in \{0,1\}$ follows a path from the root to a sink, where on a node labeled by x_i the input bit a_i is tested. If $a_i = 1$, the path follows the 1-edge, else the 0-edge. The value of the reached sink determines the value of $f(a_1, \ldots, a_n)$. On a path from the root to the sink, each variable occurs at most once. The variables on a path respect a given order, which is a permutation π on the variable indices $x_{\pi(1)} < x_{\pi(2)} < \ldots < x_{\pi(n)}$. For an edge leading from a node labeled by x_{π_i} to a node labeled by x_{π_j} it follows that $i < j$.

To deal with Boolean functions $f : \mathcal{B}^n \to \mathcal{B}^m$, we consider *multi-rooted shared* OBDDs by introducing multiple roots into a single OBDD, each root representing a subfunction of $f = (f_1, \ldots, f_m), f_i \in \mathcal{B}^k \to \mathcal{B}, k \leq n$.

All important operations on Boolean functions like the ones usually performed in synthesis and verification have efficient algorithms based on OBDDs. Due to the fact, that there are 2^{2^n} functions on n variables, many functions have an OBDD representation of exponential size. But most functions of practical relevance have only small or moderate OBDD size.

The size of an OBDD heavily depends on the variable order. Therefore, one may have an exponential gain in size by choosing a good variable order. Since it is NP–complete to find the optimal variable order for a given function (see [THY93]), much effort is spent on finding a good order or on improving a given one.

Based on swapping the positions of neighborhood variables in a given OBDD, which can be performed locally and thus, can be carried out efficiently, dynamic reordering techniques (see [Rud93]) are applied to find a smaller OBDD representation.

Another strategy to improve the variable order is based on global rebuilding of the OBDD [BMS95], but this algorithm performs even slower than sifting.

3 Problem Definition: The Multiple Variable Ordering Problem

Now, let us suppose to have two Boolean functions $f, g : \mathcal{B}^n \to \mathcal{B}^m$ over the Boolean variables $\{x_1, \ldots, x_n\}$. Further, let us suppose that they are represented with OBDDs and that f is represented using the variable order π_f, and g the variable order π_g (We only make a distinction between a function and its BDD when it is necessary to avoid ambiguity).

Let us call $\otimes$ a generic binary operator and h the result of the operation $f \otimes g$, i.e., $h = f \otimes g$, π_h being the variable ordering of h. One way to perform the operation is based on the following approach: Represent f and g with a common variable order, and then perform the operation $\otimes$.

Depending on the choice of the order π_h, we conseder two cases:

1. (Naive approach) Express f into ordering of g: $\pi_h = \pi_g$ or transform g into ordering of f: $\pi_h = \pi_f$.
2. Represent both OBDDs into an intermediate variable order: $\pi_h = \rho(\pi_f, \pi_g)$.

In the naive approach, it is more likely that the size of the resulting OBDD is exponentially larger than the original ones. Therefore, we focused on the second case of generating a well suited variable order for both OBDDs.

3.1 Greedy Method

The first heuristic uses a *greedy* method to achieve an intermediate variable order. The pseudo–code is reported in Figure 1. Given two OBDDs P and Q, with the same variable support $X_n = (x_1, \ldots, x_n)$ we simultaneously manipulate both OBDDs and compute for each level (i.e., position in the order) a common variable index step by step.

The index computation is done in the following way: The levels are processed in top-down manner. Let k be the current level. Let x_i (respectively, x_j) be the variable in OBDD P (respectively, Q) at level k, i.e., $\pi_P(k) = i$, and $\pi_Q(k) = j$, $(1 \leq i, j, k \leq n)$. Let $\pi_Q^{-1}(i)$ denote the level of x_i in π_Q. Similarly, let $\pi_P^{-1}(j)$ denote the level of x_j in π_P. If $\pi_P^{-1}(j) < \pi_Q^{-1}(i)$, move up x_j in P to the k-th level and shift down all variables between levels k and $\pi_P^{-1}(j)$ by one position. Otherwise, do the same with the variable x_i in Q. Now, P and Q share the same variable at level k.

At this point two ways of proceeding are possible:

1. (Gradual Method) Shuffle the chosen OBDD to the new order and perform a reordering of the part below the actual level for size reduction if necessary. Then proceed with the next level.
2. (At-once Method) Just compute the common variable for the next level without really affecting the

original OBDDs. After concluding the computation for all levels, shuffle both OBDDs to the new common variable order at once.

```
Adapt_step(level k, order π_P, order π_Q):
begin
        i = π_P(k);
        j = π_Q(k);
        if (i = j)
                continue;
        δ_Q = π_Q^{-1}(i) − k;
        δ_P = π_P^{-1}(j) − k;
        if (δ_P < δ_Q)
                for t = k to π_P^{-1}(j) do
                        π_{P_new}(t+1) = π_P(t);
                π_{P_new}(k) = j;
        else
                for t = k to π_Q^{-1}(i) do
                        π_{Q_new}(t+1) = π_Q(t);
                π_{Q_new}(k) = i;
end
```

Figure 1: Compute which variable to jump up to level k.

3.2 Global Weight Function

Next, we tried to generalize the greedy method described above by taking all of the remaining variables into account when deciding which variable to put at the actual level under consideration. In the greedy method, only the variables at the current level and in both orders were taken into account for a single step of the algorithm. One of the two OBDDs remained unchanged, whilest the other one was adapted. But sometimes, it might be less expensive to put other variables at the top position.

Therefore, we tried the following algorithm: Suppose we are currently proceeding with level k and are dealing with order π_P and π_Q.

1. For each variable x_i below the k-th level (note that there are the same variable set below the level in both orders), compute its *distance* to this level with respect to both orders: $\delta(k,i) = (\pi_P^{-1}(i) - k) + (\pi_Q^{-1}(i) - k)$
2. Put the variable with the lowest distances on level k.
3. Continue at step 1 with the next level.

The evaluation of the single distances for each order can be combined with a weight function $w(\delta) : \mathbb{Z} \to \mathbb{Z}$. An exponential weight function for example might closer reflect the influence of a change in the variable ordering. We experimented with the following weight functions:

- $w(\delta) = \delta$ (linear)
- $w(\delta) = \delta^2$ (quadratic) and
- $w(\delta) = 2^\delta$ (exponential).

3.3 Partial Order Method

Another possible method to compute intermediate variable orders is based on preserving the longest common partial order of both OBDDs. This technique is better suited every time the two considered variable orders π_f and π_g are close to each other. The distance of the orders is measured by number of inconsistencies which are defined as pairs (i,j) such that $(x_i <_{\pi_1} x_j) \wedge (x_j <_{\pi_2} x_i)$. The goal is to find an intermediate order π that is as close as possible to both π_1 and π_2.

We need some additional technical notions. An order σ_1 defined on some subset $A \subset X$ is called a *suborder* of an order σ_2 over X if for any $x_i, x_j \in A$ holds $x_i <_{\sigma_1} x_j$, if and only if $x_i <_{\sigma_2} x_j$, i.e., σ_1 is a restriction of σ_2 to a set A. σ_1 is then denoted by σ_2^A, and $|A|$ is called the length of σ_1. A *common suborder* σ of orders σ_1 and σ_2 is defined by $\sigma = \sigma_1^A = \sigma_2^A$, for any $A \subset X$.

Let P_1 and P_2 are considered OBDDs that are built with respect to orders π_1 and π_2, respectively. For the construction of a new order π not only π_1 and π_2 is taken into account, but also P_1 and P_2. π is constructed in top-down manner by processing π_1 and π_2 in the same way. In every step, we choose one from the top variables of π_1 and π_2 according to the values of the weight functions defined below, remove it from π_1 and π_2, and append to π. There is one exception when the weight functions are not applied. This happen if π_1 and π_2 have the same top variable that is chosen in a straightforward way.

We use three weight functions w_1, w_2, and w_3 as a selection criterion for the next variable in the created order. They are applied subsequently on the top variables of π_1 and π_2 (denoted by *top(π_1)* and *top(π_2)*, respectively), until their values differ. If the values of a weight function differ, we choose the variable with the larger weight. If the values of all weight functions on the variables agree, we chose the top variable from π_2 by default.

$w_1(x)$ is defined as the maximal length of the suborders of π_1 and π_2 with x on the top. If the values of w_1 on *top(π_1)* and *top(π_2)* agree, we compute w_2 which is defined as follows. Let i and j be the positions of x in π_1 and π_2, respectively. If $i < j$ ($j < i$), $w_2(x)$ is defined as the number of nodes on levels $i, i+1, \ldots, j$ ($j, j+1, \ldots, i$) in P_2 (respectively, in P_1). This weight function reflects an overhead for moving up x from its position in one order to its position in the second order, that change P_2 (P_1). w_3 is defined as the difference in the positions of x in π_1 and π_2. The pseudo-code in Figure 2 describes how to create π, which is assumed to be implemented as a list that is empty at the beginning. n is the number of variables.

4 Experimental Results

We performed all experiments on an Intel PentiumPro 200MHz Linux Workstation with datasize limited to 200 MByte and CPU time limited to 60 minutes. Our heuristics were implemented in CUDD package [Som] and experimentally evaluated on a subset of the LGSynth91 Benchmark circuits [LGS].

```
Create(π1, π2, π)
begin
        if (|π| = n)
                return π;
        if (top(π1) = top(π2))
                y = top(π1);
        else
                if (w1(top(π1)) = w1(top(π2)))
                        if (w2(top(π1)) = w2(top(π2)))
                                if (w3(top(π1)) > w3(top(π2)))
                                        top(π1);
                                else
                                        y = top(π2);
                        else
                                if ((w2(top(π1)) > w2(top(π2))))
                                        y = top(π1);
                                else
                                        y = top(π2);
                        else
                                if ((w1(top(π1)) > w1(top(π2))))
                                        y = top(π1);
                                else
                                        y = top(π2);
        Remove(y, π1);
        Remove(y, π2);
        Append(y, π);
        Create(π1, π2, π);
end
```

Figure 2: Computing the Partial Order Heuristic.

4.1 Splitted Circuits

We started with the following experimental setup: We have chosen only those circuits from LGSynth91 including output gates of fanin 2. Then, we have splitted the circuits at this binary gate into two parts as shown in Fig. 3.

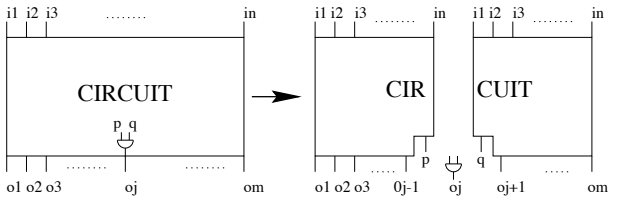

Figure 3: Splitting a Circuit in Two Parts

For the two sub-circuits OBDDs have been constructed and optimized separately via dynamic reordering. This results in most cases in OBDDs with different variable orders. By applying the output gate operation onto these two subcircuits we re-compute the original circuit. To do this, we have to compute a common variable order for the OBDDs of the sub-circuits.

The experimental results are shown in Figure 4 and Figure 5.

The first column denotes the circuit name. In the following columns we show for each circuit the maximum number of OBDD nodes created during computation (Peak) and the final size of the resulting OBDD (Final). The two columns titled by *Gradual* and *Atonce* show the greedy approach discriminated in gradual and at once computation. The three columns title by *Global Quadratic* and *Global Exponential* show the Global weight approach discriminated in quadratic and exponential weights. The columns titled by *Partial Order* show the size of the resulting OBDD and the length of the maximum common suborder (PO) in relation to the number of variables (Var). For comparison the columns titled by *Transform 1 to 0* and *Transform 0 to 1* show the results of transforming the order of the first OBDD to the order of the second OBDD and vice versa i.e. the naive approach. Empty fields in the table denote experiments that could not be performed due to memory/CPU limitats.

From our results we can draw the following conclusions: Comparing the *gradual* and the *at once* version of the greedy algorithm, the *gradual* method beats in most cases the *at once* version due to reordering while computing the intermediate order.

Partial Order is better suited in situations where we have a large common suborder. Therefore, we may switch from the greedy method to *Partial Order* if the length of the common suborder is greater than one quarter of the total number of variables.

Comparing intermediate orders to the static techniques in most cases the intermediate order method results in smaller OBDD sizes. In some cases intermediate orders are even better than orders computed by conventional dynamic reordering.

4.2 Pairs of Circuits

For the second group of experiments we have chosen those pairs of circuits from the LGSynth91 Benchmark set, that have similar numbers of variables and comparable OBDD sizes if they are constructed separately. The variable names of both circuits are changed to $x_1, \ldots, x_m$ where m is the size of the larger variable set, according to the order of their appearance in the circuit description files. Now, both circuits join exactly the same variables. After constructing and optimizing the circuits separately, the different heuristics for finding intermediate variable orders are applied.

For the experimental results see Figure 6 and Figure 7. Due to the fact that there are artifically introduced variables in the circuits, the heuristic "partial order" is not applied here.

In comparison to the naive approach of just transforming one OBDD to the variable order of the other OBDD, our heuristics are able to finish computation for much more circuits. On the other hand, regarding those circuits that admit to finish computation in the naive approach, our heuristics in most cases gain better results.

4.3 Sequential Circuits

In the third group of experiments we consider a problem of practical relevance i.e. the computation of a common variable order of the transition relation of a sequential circuit and the set of states reachable after a certain number of traversal steps. Often the transition relation and the reachable states set of a circuit have different variable order requirements. Hence, the OBDDs are optimized separately, especially the variable order for the reachable states set has to be chosen carefully. But, also a common representation is sometimes usefull.

Figure 8 and Figure 9 show the results of applying our heuristics to the transition relation of some circuits of the LGSynth91 benchmark circuits and their reachable state sets after various traversal steps. The OBDDs for the reachable states sets are optimized to a good order. For comparison the sum of OBDD-nodes of the concerning OBDDs before the computation, i.e. with different variable orders, is shown in the column "TR+RS Before".

Also in these experiments of practical evidence our heuristics compute good variable orders. Since the heuristics are not yet especially tuned for sequential circuits, this seems to be a very promising approach for this problem.

5 Some Correlated Problems

We can also target the opposite problem.

If the function h is not possible to represent with order o_h (as the BDD is large) how to decompose it in two sub–BDDs (for example using the decomposition strategy presented in [CCQ96]) and representing each sub–BDDs with a proper order? The technique has been already used, see for example [NIJ+97], but no theory and/or optimization seem to e reported there.

Another correlated problem is to evaluate the number of BDDs in which is necessary to decompose a single BDD to obtain a suitable variable ordering for each of them, i.e., minimizing the overall number of nodes.

Acknowledgement

We like to thank Arno Wagner for his help on our experimental environment.

References

[BMS95] J. Bern, C. Meinel, and A. Slobodová, Global Rebuilding of OBDDs – Avoiding Memory Requirement Maxima,in *Proc. Computer Aided Verification 95*, volume 939 of *Lecture Notes in Computer Science*, 4–15, 1995.

[Bry86] R. E. Bryant, Graph-Based Algorithms for Boolean Function Manipulation, *IEEE Transactions on Computers* , C-35:677–691, 1986.

[Bry91] R. E. Bryant, On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication, *IEEE Transactions on Computers*, 40:205–213,1991.

[FHS78] S. Fortune, J. Hopcroft and E. Schmidt, The Complexity of Equivalence and Containment for Free Single Variable Program Schemes, *Proc. of ICALP'78 Automata, Languages and Programming*, volume 62 of *Lecture Notes in Computer Science*, 227-240, 1978.

[CCQ94] G. Cabodi, P. Camurati and S. Quer, Auxiliary variables for extending symbolic traversal techniques to data paths, in *Proc. ACM/IEEE DAC'94*, 289–293, 1994.

[CCQ96] G. Cabodi, P. Camurati and S. Quer, Improved Reachability Analysis of Large Finite State Machine, in *Proc. IEEE/ACM ICCAD'96*, San Jose, CA, USA, 354–360, 1996.

[LGS] LGSynth91 Benchmarks: http://www.cbl.ncsu.edu/CBL_Docs/lgs91.html.

[NJF+96] A. Narajan, J. Jain, M. Fujita and A. Sangiovanni–Vincentelli, Partitioned ROBDDs – A Compact, Canonical and Efficient Manipulable Representation of Boolean Functions, in *Proc. IEEE/ACM ICCAD'96*, San Jose, CA, USA 547–554, 1996.

[NIJ+97] A. Narajan, A. Isles, J. Jain, R. K. Brayton and A. Sangiovanni–Vincentelli, Reachability Analysis Using Partitioned–ROBDDs, in *Proc. IEEE/ACM ICCAD'97*, San Jose, CA, USA, 547–554, 1997.

[QCC+96] S. Quer, G. Cabodi, P. Camurati, L. Lavagno, E. M. Sentovich, R. K. Brayton, Incremental Re–encoding for Symbolic Traversal of Product Machines, in *Proc. EURO–DAC'96*, Geneva, Switzerland, 158–163, 1996.

[Rud93] R. Rudell, Dynamic Variable Ordering for Ordered Binary Decision Diagrams, in *Proc. IEEE/ACM ICCAD'93*, 42–47, 1993.

[Som] F. Somenzi, CUDD CU Decision Diagram Package.

[THY93] S. Tani, K. Hamaguchi, and S. Yajima, The Complexity of the Optimal Variable Ordering Problem of Shared Binary Decision Diagrams, in *Proc. ISAAC'93*, volume 762 of *Lecture Notes in Computer Science*, 389–398, 1993.

	Gradual		Atonce		Global Quadratic		Global Exponential		Partial Order			Transform 1 to 0		Transform 0 to 1	
Circuit	Peak	Final	Peak	Final	Peak	Final	Peak	Final	(PO)Vars	Peak	Final	Peak	Final	Peak	Final
C3540	296380	30186	-	-	296380	30192	296380	**30183**	(26)51	296380	32138	296380	30559	296380	43598
C7552	70518	**17142**	-	-	-	-	-	-	-	-	-	-	-	-	-
adder16	11242	961	19418	11828	3066	333	9198	1083	(11)34	8176	948	7154	981	2044	**131**
alu4	1022	147	1022	147	1022	60	1022	**50**	-	-	-	1022	147	1022	147
dpath32	290248	30109	-	-	294336	28170	310688	32634	(23)94	40880	**496**	146146	1966	19418	702
dsip	19418	**4889**	19418	7175	19418	4896	19418	**4889**	(105)453	19418	4892	19418	4891	19418	17480
k2.blif	17374	1554	17374	2761	17374	1504	17374	**1492**	(18)46	17374	1826	17374	1737	17374	1755
mm30a	-	-	-	-	-	-	-	-	(24)124	4235168	587546	2645958	**83820**	4235168	587561
mm9a	8176	1533	88914	86581	18396	8231	8176	1545	(8)40	8176	2376	8176	1557	8176	**1054**
mm9ac	9198	6199	50078	44348	7154	5750	5110	**946**	(7)40	6132	5252	5110	957	16352	10718
mm9b	13286	2506	13286	10604	13286	6719	13286	2395	(12)39	13286	**2305**	13286	2397	13286	3137
mm9bc	13286	2425	13286	4423	17374	13848	13286	2251	(12)39	13286	2774	13286	2768	13286	**2020**
s1196	3066	1504	3066	2554	3066	2242	3066	**867**	(13)33	3066	1841	3066	922	3066	1403
s3384	8176	1869	839062	169051	8176	1827	8176	**1485**	(64)227	8176	2402	8176	1788	8176	2528
s344	1022	**204**	1022	409	1022	**204**	1022	**204**	(24)25	1022	409	1022	409	1022	409
s4863	756280	**128178**	756280	308055	-	-	-	-	(31)154	756280	177066	756280	150941	756280	177064
s499c	2044	375	2044	**340**	2044	404	2044	457	(23)24	2044	**340**	2044	**340**	2044	**340**
s967c	2044	**351**	2044	472	2044	373	2044	369	(10)46	2044	483	2044	420	2044	461
ttt2	1022	**146**	-	-	1022	201	1022	183	-	-	-	1022	232	-	-
vda	7154	600	7154	636	7154	732	7154	**575**	(7)18	7154	817	7154	604	7154	985

Figure 4: Experimental Results for "Splitted Circuits" OBDD-Sizes

	Gradual	Atonce	Global Quadratic	Global Exponential	Partial Order	Transform 1 to 0	Transform 0 to 1
C3540	279.22	-	336.62	345.39	185.26	186.47	185.27
C7552	451.56	-	-	-	-	-	-
adder16	2.25	0.61	2.88	3.22	0.46	0.42	0.36
alu4	0.44	0.06	0.61	0.65	-	0.06	0.07
dpath32	102.40	-	117.24	167.61	7.17	38.45	3.85
dsip	452.86	21.79	740.13	753.00	21.62	21.52	21.52
k2.blif	5.93	2.01	8.81	9.28	1.53	1.55	1.52
mm30a	-	-	-	-	2701.93	1490.80	1539.83
mm9a	6.31	3.43	14.79	9.20	14.59	2.57	2.60
mm9ac	5.32	1.73	7.80	6.11	7.47	1.27	1.46
mm9b	7.06	3.53	12.70	11.65	19.75	3.45	3.49
mm9bc	7.48	3.82	16.87	12.64	21.53	3.79	3.88
s1196	2.97	0.67	5.03	4.47	3.71	0.63	0.65
s3384	97.68	261.42	184.16	190.14	50.08	8.89	8.91
s344	0.84	0.10	1.36	1.33	0.50	0.10	0.09
s4863	3216.70	519.33	-	-	516.90	511.82	511.61
s499c	1.11	0.08	1.78	1.87	0.07	0.09	0.11
s967c	2.72	0.24	4.28	4.20	0.21	0.23	0.23
ttt2	0.83	-	1.31	1.29	-	0.06	-
vda	1.00	0.42	1.43	1.38	0.35	0.36	0.35

Figure 5: Experimental Results for "Splitted Circuits" CPU-Time (seconds)

	Gradual		Atonce		Global Linear		Global Quadratic		Global Exponential		Transform 0 to 1		Transform 1 to 0	
	Peak	Final	Peak	Final	Peak	Final	Peak	Final	Peak	Final	Peak	Final	Peak	Final
C1355–C3540	402668	278626	402668	187962	402668	368609	402668	**134532**	402668	233522	–	–	–	–
C499–C1355	320908	127661	320908	91454	320908	89891	320908	**78758**	320908	122730	–	–	–	–
C2670–s9234.1	–	–	181916	**171057**	–	–	–	–	–	–	–	–	–	–
adsb32–alu32	8176	5884	2044	**1078**	7154	5438	106288	86319	6132	4300	5110	4445	–	–
adsb32–dalu	70518	68759	160454	159526	15330	**2634**	–	–	15330	2995	15330	2891	25550	25075
i8–k2	18396	6991	18396	**3933**	18396	10766	18396	8547	18396	10339	–	–	–	–
mm9b–too–large	117530	72161	58254	30312	–	–	24528	13150	20440	8959	859502	4031	11242	**3929**
s1269–mm9b	97090	64849	28616	**6912**	91980	54402	26572	9115	70518	30660	–	–	–	–
s6669–s15850.1	–	–	309666	**65828**	–	–	–	–	–	–	–	–	–	–
sbc–alu32	10220	8683	2044	**5511**	27594	21012	21462	12227	5110	3741	20440	19196	–	–
sbcc–alu32r	1055726	937639	31682	1748	1297940	1296836	31682	2009	318864	313319	31682	**695**	31682	2416
too–large–vda	13286	4111	13286	2011	**13286**	4960	13286	1718	13286	3745	13286	6429	13286	2176
vda–alu4	6132	**1129**	6132	1212	6132	1294	6132	1218	6132	971	6132	1189	6132	1140

Figure 6: Experimental Results for "Pairs of Circuits" OBDD-Sizes

	Gradual	Atonce	Global Linear	Global Quadratic	Global Exponential	Transform 0 to 1	Transform 1 to 0
C1355–C3540	661.1	322.9	1321.3	1148.6	1175.3	–	–
C499–C1355	550.8	239.0	715.8	599.4	759.5	–	–
C2670–s9234.1	–	55.0	–	–	–	–	–
adsb32–alu32	27.0	5.5	38.3	103.0	39.6	5.6	–
adsb32–dalu	54.9	5.7	26.4	–	26.9	3.8	4.0
i8–k2	64.0	5.2	102.7	79.3	106.9	–	–
mm9b–too–large	71.9	6.1	–	22.1	27.6	233.0	5.0
s1269–mm9b	148.0	9.6	46.1	34.3	33.3	–	–
s6669–s15850.1	–	517.1	–	–	–	–	–
sbc–alu32	31.8	5.9	44.5	15.2	37.5	6.5	–
sbcc–alu32r	1457.5	2.1	1390.1	20.8	351.8	2.2	2.1
too–large–vda	5.6	1.3	7.6	6.3	7.5	1.4	1.3
vda–alu4	1.3	0.4	1.7	1.7	1.7	0.5	0.4

Figure 7: Experimental Results for "Pairs of Circuits" CPU-Time (seconds)

Circuit	Traversal Steps	Gradual		Global Linear		Global Quadratic		Global Exponential		TR+RS Before
		Peak	Final	Peak	Final	Peak	Final	Peak	Final	
s1512	50	290248	132968	150234	5708	680652	610200	150234	6416	132972
	80	430262	163249	184982	10079	–	–	184982	22593	163253
s1423	6	192136	81374	166586	49809	–	–	1070034	838660	50112
	8	475230	138489	425152	107247	–	–	–	–	68935
s1269	1	126728	57100	97090	14518	62342	7046	62342	5400	52379
	2	–	–	570276	139165	383250	91136	247324	51452	65381

Figure 8: Experimental Results for Sequential Circuits, OBDD-Sizes

Circuit	Traversal Steps	Gradual	Global Linear	Global Quadratic	Global Exponential
s1512	50	113.2	76.1	1604.1	97.6
	80	195.6	100.2	97.6	124.5
s1423	6	309.3	1882.5	–	3623.7
	8	2010.1	4106.5	–	–
s1269	1	48.0	73.2	52.3	53.5
	2	–	1241.8	278.7	251.2

Figure 9: Experimental Results for Sequential Circuits, CPU-Time (seconds)