Global Rebuilding of OBDDs Avoiding Memory Requirement Maxima*

Jochen Bern, Christoph Meinel, Anna Slobodová**

University of Trier, D–54286 Trier, Germany

Abstract. It is well-known that the size of an ordered binary decision diagram (OBDD) may depend crucially on the order in which the variables occur. In the paper, we describe an implementation of an output–efficient algorithm that transforms an OBDD P representing a Boolean function f with respect to one variable ordering π into an OBDD Q that represents f with respect to another variable ordering σ. The algorithm runs in average time $O(|P||Q|)$ and requires $O(|P| + n\,|Q|)$ space.
The importance of the algorithm is demonstrated by means of experimental results on basically two different applications. In one of them, the algorithm is used merely once. Such transformations are needed to test equivalence or to perform synthesis on OBDDs in which variables appear in different orders. The other application shows a way how to decrease the size of intermediate representations in the course of the construction of OBDDs from a given circuit. Here the algorithm is used dynamically, whenever the size of the manipulated OBDDs becomes too large.

1 Introduction

In [Bry86], Bryant introduced *ordered binary decision diagrams (OBDDs)* as data structure for Boolean function and circuit manipulation. Let π be a permutation of the variables $X = \{x_1, \ldots, x_n\}$. A πOBDD P over X (sometimes shortly called OBDD) is an acyclic directed graph where the nodes are labeled with variables from X. The sinks are labeled with Boolean constants. Each non-sink node has two outgoing edges, one labeled with 0 and the other with 1. Edges starting in a node labeled with $\pi(i)$ end in a node labeled with $\pi(j), i < j$, or in a sink. Each OBDD node v represents a Boolean function f_v. Starting in v, the value $f_v(a)$ for an input $a \in \{0,1\}^n$ is the constant assigned to the sink that is reached if from every x_j–node the edge that is labeled with a_j is followed. An OBDD is called reduced if there is no node whose 0–successor and 1–successor coincide and if it does not contain any isomorphic labeled subgraphs. From [Bry86], it is well–known that, with respect to a given permutation π, reduced OBDDs provide a universal and canonical representation scheme for Boolean functions and circuits that allows efficient composition as well as equivalence test. The

* Proc. CAV'95, LNCS, 1995

** Granted by DFG Me 1077/2-1

many nice properties of OBDDs have made OBDDs the favorite data structure used nowadays in many applications in CAD.

OBDDs provide an efficient graph–based data structure for Boolean functions and circuits if good variable orderings are known. Unfortunately, the problem of computing an optimal variable ordering even from an OBDD – i.e., a variable ordering in accordance to which the corresponding OBDD–representation is of minimal size – was shown to be NP–complete [THY93, BW94]. The assymptotically best known deterministic algorithm for computing an optimal variable ordering is due to Friedman and Supowit [FS90] and needs exponential (with respect to the number of variables) time and space. Therefore, variable orderings are computed by heuristics. Due to the great importance of finding good orderings, a lot of heuristic algorithms have been proposed in the past (see, e.g., [MWBS88, FFK88, MIY90, BRKM91]). Since none of them works well for all circuits of interest, the idea arose to improve orderings computed by a heuristic by some local search algorithm (e.g. [FMK91, Rud93, MKR92]). In particular, Rudells "sifting technique" has found great attention and many applications. The idea is to start with an OBDD–representation corresponding to an ordering π (chosen randomly or found by any heuristics), then, by means of a sequence of pairwise exchanges of the neighboring variables, to find a more concise representation. Unfortunately, in the meantime, the first enthusiasm for this approach was dampened by experimental work. It spelt out that in many cases it is not possible, because of a shortage in space and/or time, to come from a weak local minimum to a neighboring better (local) minimum in this way since it might be impossible to overcome the local maxima located between these minima.

The question arises whether it is possible to directly transform a given OBDD of a function (of a circuit) with respect to one ordering π into an equivalent OBDD with respect to another ordering σ, without the necessity to perform this transformation via a (long) sequence of local changes. A first positive answer to this question is given in [MS94][3]. by describing an algorithm that transforms a πOBDD P into a functionally equivalent σOBDD Q in average time less than $O([Q][P]\log[P])$, where $[Q]$ ($[P]$) denotes the space for encoding a BDD Q (resp. P). Since the output of such a transformation can be exponential with respect to the size of the input, an output–efficient algorithm like this one is the best one can hope for. Its performance depends polynomially on the size of the input OBDD and the size of the output OBDD. Neither time nor space depend from unknown and unpredictable sizes of intermediate OBDDs, unlike the approach of using sequences of local changes for this transformation. That is the substantial difference between the direct global rebuilding and the rebuilding by local changes.

After describing the rebuilding algorithm *REBUILD* and discussing some details of its implementation and its complexity in Section 2, in Section 3, we report our experimental work with ISCAS circuits, done with *REBUILD* in the frame of the OBDD package of CMU [BRB90] and of the BDD–software environment of Trier $\mathrm{T\overline{RUS}T}$ [BMS94]. We show that

[3] Simultaneously and independently, a rebuilding algorithm with a similar performance was designed in [TI94] and later on in [SW94].

- it is possible to decrease the total memory resources needed to compute a σOBDD–representation Q directly from a circuit by computing a suited πOBDD P and applying *REBUILD* to transform P into Q,
- it is sometimes faster to transform a given πOBDD–representation of a circuit C into an equivalent σOBDD–representation by means of *REBUILD* than to derive the σOBDD–representation from C by symbolic simulation,
- it is possible to decrease the total memory requirements of a symbolic simulation of a circuit as well as the size of its final OBDD-representation by dynamic global reordering of the variables and the corresponding rebuilding of the OBDDs in the course of the simulation.

Other possible applications of our rebuilding algorithm are

- the extension of the idea of dynamical reordering [Rud93] by allowing a simultaneous exchange of blocks of variables instead of single variables,
- combination of the known heuristics based on the local improvements with the rebuilding algorithm as a way of "jumping" from the local minima,
- combination of the sifting with global reordering.

Experiments in that area are still ongoing.

2 The Rebuilding Algorithm and its Implementation

The idea of the algorithm that allows to transform any given (reduced) πOBDD P of a function f into a (reduced) σOBDD of f, where π and σ are arbitrary permutations of the variables of f, was described first in [MS94]. Working on this idea, we implemented the algorithm (called *REBUILD* in the following) as a part of the CMU-pacakge [BRB90]. In this section, we discuss some implementation details and the complexity behavior of the algorithm. Its pseudo-code is given below.

REBUILD is based on the recursive procedure *transform–OBDD* that is built along the divide-and-conquer principle, and that uses a computed table and dependency check in order to avoid the creation of reducible nodes (and respective exponential behavior).

The initialization, i.e., the creation of the sinks of Q, and the call to *transform–OBDD* on the highest level are done in the main program.

Algorithm *REBUILD*.

Input: A reduced OBDD P, an order σ of all variables appearing in P.

Output: The reduced σOBDD Q that is computationally equivalent to P.

Rebuild

begin

read(P, σ);

initialize Q to consist of two sinks - ZERO and ONE;

write(*transform–OBDD(P,* $\sigma, \emptyset$*)*)

end

transform–OBDD(R, τ, a)

begin

(1) **if** *constant(R)*
then return the respective sink of Q;
(2) **if** *computed*(R) **then return** result;
(3) $\sigma := \sigma\langle top\_var(\tau, R)\rangle^1$, where
$top\_var(\tau, R) = min\{i \mid R$ depends on $\tau[i]\}$;
(4) $R0$:=*restrict_and_reduce*$(R, \tau[1] = 0)$;
(5) $v0$:=*transform–OBDD*$(R0, \tau\langle 2\rangle,\ a \cup \{\tau[1] = 0\})$;
(6) $R1$:=*restrict_and_reduce(R,* $\tau[1] = 1$*)*;
(7) $v1$:=*transform–OBDD*$(R1, \tau\langle 2\rangle,\ a \cup \{\tau[1] = 1\})$;
(8) create a new node v of Q, labeled with $\tau[1]$,
with then-son $v1$ and else-son $v0$;
(9) *store_in_computed_table*(*signature*$(R), a, v)$;
(10) **return** v

end

computed(S)

begin

(2.1) sgn :=*signature*(S);
(2.2) for each item $I = (sgn, \tilde{a}, \tilde{v})$ in the computed table
(2.2.1) $\tilde{S}$ =*restrict_and_reduce*$(P, \tilde{a})$;
(2.2.2) **if** $\tilde{S} \equiv S$ **return** $\tilde{v}$;

end

The procedure *transform–OBDD* has three parameters: R – the OBDD we want to rebuild, τ – the desired order of the resulting OBDD (a suborder of σ), and a – the partial assignment that specifies R as a restriction of P. It starts with the test of the trivial cases. If the function represented by R is a constant, the sink of Q for this constant is returned. Otherwise, it continues with a dependency check which chooses the first variable x in σ the function depends on. Since the function is not a constant, such a variable always exists.

[1] By $\tau\langle i\rangle$ we denote the suborder $(\tau[i] < \tau[i+1] < ... < \tau[n])$ of τ.

The next step is a call to the function *computed* that performs a look-up into the computed table. For any reduced πOBDD S *computed* returns TRUE if the procedure *transform_OBDD* has been previously called with parameter $\tilde{S}$ such that $\tilde{S} \equiv S$. If the look-up is successful, the result is simply loaded from the computed table. Otherwise *computed* returns FALSE. In that case we are sure that the node created by this call will not be reduced in the future.

This is a tricky part that uses the following facts:

- Each boolean function has a canonical OBDD–representation with respect to a fixed order. Hence, the function represented by R rather than R itself is the parameter of the computed function.
- Each of the functions considered in the previous calls to *transform–OBDD* (i.e., each function represented by some node of S) is a restriction of P.
- The restriction of P according any partial assignment can be computed in linear time with respect to the size of P.
- All restrictions of P are consistent with the same order, and the equivalence test among them can be performed in constant time (in average) if they are represented as shared OBDDs.

Since P is permanently in the memory and with respect to the facts above, the items in the computed table can be of the following form:

signature	a function represented by a partial assignment to variables	result represented by the respective node in Q

A *signature* is an element from a finite field that can be associated with each boolean function as described in [BCW80].

In order to get the estimation of the behavior of the algorithm as tight as possible, we will analyze it more precisely than in [MS94] and with respect to all implementation details. Before doing this, let us make some remarks to the difference between the theoretically estimated and real-world time/space.

Every precise asymptotic analysis of the algorithm has to start from the fact that a BDD G of size $|G|$ (number of nodes) is encoded in space $[G] = |G| \log |G|$. This follows from the observation that each node is accessed via its address that has length $\log |G|$. Hence, each algorithm with a linear number of constant cost operations has performance time $|G| \cdot \log |G|$. However, such an estimation can be far from the real–world run–time of the algorithm, since a specific implementation a priori uses restricted resources. Nevertheless, we can always use the theoretical upper bound as a starting point for our estimation.

Theorem 1. *The algorithm REBUILD*(P, σ) *transforms a given OBDD P over n variables into a functionally equivalent σOBDD Q. If k-tuple signatures are used, then it runs in average time $\mathcal{O}(k\,|Q|\,|P| \log^2 |P|)$ and requires space $\mathcal{O}(|P| + (k+n)|Q|)$.*

Proof. Due to commands 1 - 3, no redundant node is created. Since by each call to the procedure *transform* one edge is created and since the number of edges is two times the number of internal nodes in the OBDD, $2|Q|$ calls to *transform* are generated.

The most expensive command in the procedure is the second one – the look-up into the computed table, which is performed by *computed*. Let us estimate the time behavior of this function. An important observation is that $|S| \leq |P|$ holds for all values of the parameter S (S is always a restriction of P).

If k-tuple signatures of [BCW80] are used, step 2.1 costs $\mathcal{O}(k\,|P|\,log^2|P|)$. The probability that two different functions have the same signature is less than $p = (\frac{1}{2})^k$. If the signature is used as the key for the hash function of the computed table, access to items with the same signature can be done in constant time.

Step 2.2 consists of two commands that can be repeatedly performed until either an item is found whose function is equivalent to that represented by S or all items with the signature sgn (i.e., at most $|Q|-1$) are processed. Since 2.2.1 and 2.2.2 cost $\mathcal{O}(|P|\,log|P|)$ and constant time, respectively, the body of 2.2 costs $\mathcal{O}(|P|\,log|P|)$. With probability more than $(1-p)$ just the first equivalence test in 2.2.2 will be positive and *computed* terminates. With probability $p(1-p)$ the second test will be positive, and generally, with probability $p^i(1-p)$ the equivalence test will be positive after exactly $i+1$ performances of the body of 2.2, i.e. $i+1$ items will be checked. Hence, the average time of *computed* can be estimated by

$$T_{comp}(p,q,r1,r2) = \mathcal{O}\left(\left(\left(\sum_{i=0}^{q} p^{i-1} \cdot i\right)(1-p) + p^q q\right) r1 + r2\right)$$

where $p = (\frac{1}{2})^k$, $q = |Q|-1$, $r1 = |P|\,log|P|$, and $r2 = k\,|P|\,log^2|P|$.
Since

$$\sum_{i=0}^{q} p^{i-1} \cdot i = \frac{1-(q+1)p^q + qp^{q+1}}{(1-p)^2} < \frac{1}{(1-p)^2}$$

and

$$p^q q = \frac{q}{2^{kq}} < 1 \text{ for any } k > 0$$

we have

$$\begin{aligned} T_{comp}(p,q,r1,r2) &= \mathcal{O}\left(|P|\log|P| \cdot \left(\tfrac{1}{1-p}+1\right) + k\,|P|\,log^2|P|\right) \\ &= \mathcal{O}\left(|P|\log|P| \cdot \left(\tfrac{2^k}{2^k-1} + k\,|P|\,log^2|P|\right)\right) \\ &= \mathcal{O}\left(k\,|P|\,log^2|P|\right) \end{aligned}$$

and the time bound is proved.

The space bound follows from the fact that at any time we store only P, two restrictions of P, and Q. The additional information is stored in the computed table that has at most $|Q|$ items and consists of signatures and assignments to variables. □

Let us estimate the complexity of our implementation on the basis of the CMU-package. Without loss of generality we can make the following two assumptions:

- Addressing of a BDD node costs constant time.
  The restriction is given by the constant length of the adresses. Since the length is long enough to store any BDD of managable size, this assumption is not a real restriction.
- A signature is stored in constant space (4 bytes) and is manipulated (over the field) in constant time.
  We use the simple signatures ($k = 1$) over the field $\mathbb{Z}_m$, where $m = 2^{31}-1$. Signatures are stored as integers and the implemented arithmetic costs constant time and space. The selection of $\mathbb{Z}_m$ means no restriction as long as we work with circuits with up to 2^{30} inputs.

Under these assumptions, the average time of the algorithm REBUILD dropps as stated in the following corollary.

Corollary 2. *The implementation of the algorithm REBUILD based on the CMU-package [BRB90] runs in average time $\mathcal{O}(|P|\,|Q|)$ and requires $c(|P|+|Q|+\frac{n}{2c}\,|Q|)$ bytes space, where $c = 26$ is the number of bytes used per BDD-node by the package.*

In order to test the usability of the algorithm in several possible applications, we made a series of experiments.

3 Experimental Work

3.1 Experiment Setup

For our experiments, we worked with six order heuristics for determining variable orderings that lead to succinct OBDD–representations. These heuristics are based on the following three well-known heuristics: The fanin heuristic of Malik et. al. (`d0`)[MWBS88], the closely related heuristic implemented in SIS (`d0s`), and the weight propagation heuristic of Minato et. al. (`W0`) [MIY90]. The heurstics `d0r`, `d0sr` and `W0r` are derived from these by simply reversing the respective orders. First, we implemented these heuristics into our application software $\mathrm{T\overline{RUS}T}$ [BMS94] and the rebuilding algorithm into the OBDD package of Brace, Bryant and Rudell [BRB90]. A rebuilding phase can be started by calling *REBUILD*, both during the symbolic simulation (triggered by a size limit, and using a given heuristic to determine a suited variable ordering based on the circuit representation of the currently active functions) and in a final step to transform the resulting OBDDs into other ones with respect to a given order.

We use the following notations in the tables: $\pi(i)$ denotes the order obtained by the heuristic i, $M(i)$, denotes the maximum (shared) size of the $\pi(i)$OBDDs during the derivation, and $m(i)$ denotes their final size.

All experiments were performed on SUN ELC and SPARCstation 10 with the memory bound 60 MB. The time is always given in seconds.

3.2 One–time rebuilding

Many different situations are imaginable in which a rebuilding of an OBDD according to a new variable ordering is desired. Let us mention some of them.

One such situation appears when the equivalence test of two OBDDs with respect to different orders is required. For instance, by verifying a circuit against its specification which is in the form of some πOBDD. We have two possibilities to do it. First – to derive the πOBDD directly from the circuit; second – to derive a σOBDD from the circuit, and then either to transform the specification according to σ, or to transform the σOBDD representation of the circuit according to π.

The first way can be time and space consuming, or even unpracticable, if there are some gates in the circuit that are not well representable according to π. On the other hand, these gates could be well–representable according to some other order σ. Then the second way is viable.

Another situation appears if we want to synthesize two OBDDs which have different variable ordering. The rebuilding to a common variable order is a solution.

The experiments that were made for the behavior of the algorithm in such situations are evaluated in Tab.1 and Tab.2. Table 1 shows examples where the derivation of the $\pi(i)$OBDD from the circuit followed by rebuilding into an equivalent $\pi(j)$OBDD is *less space consuming* than the derivation of a $\pi(j)$OBDD directly from the circuit. The total memory requirements in both runs were considered. In the first case it was $M(j)$, in the second the maximum of $M(i)$ and $m(i) + m(j)$. More precisely, the approach with final rebuilding makes sense if for the given circuit $max\{M(i), m(i) + m(j)\} < M(j)$. Taking d0 and W0 for the order heuristics i and j, respectively, we found several circuits exhibiting this behavior.

Table 2 summarizes examples where obtaining OBDDs of given functions and orders by rebuilding is *faster* than the direct symbolic simulation.

3.3 Dynamic global rebuilding

Verification of a combinational circuit consists of comparing a representation of the (outputs of the) circuit with the representation of its specification. The usual way to obtain the representation of the circuit is the symbolic simulation – starting with the representation of the inputs, the representation of the output of each gate is constructed gate by gate. The process terminates having constructed the representations of the outputs of the circuit. Symbolic simulation requires a fast performance of the boolean operations on the representation. This is fulfilled by OBDDs.

The functions represented by the gates can be quite different. The situation where no variable ordering is "good" for all of these functions is not rare. Even if such "good" variable orderings exist, there is no known heuristic how to find them. In this context, the idea of changing the variable ordering in the course of the symbolic simulation seems to be very natural. The "sifting" [Rud93] is an implementation of this idea. Starting with any variable ordering (random or computed by some heuristic), the process of the symbolic simulation proceeds until the OBDD–representation becomes too large. Then, by the exchange of neighbouring variables, a new, possibly better, order is looked for. The search space is very restricted due to the restricted local changes of the considered

Table 1. Reductions of memory requirements by final rebuilding. Bold-faced figures indicate a comparison of the maximal memory requirements. All experiments run on SPARCStation 10.

Circuit C	$C \to \pi(W0)$OBDD			$C \to \pi(d0)$OBDD$\to$ $\pi(W0)$OBDD			Gain
Name	$M(W0)$	$m(W0)$	Time	$M(d0)$	$m(d0)$+$m(W0)$	Time	%
c1355	**156,836**	82,417	93	79,739	**123,094**	3,455	27.4
c3540	**607,245**	338,972	452	187,504	**455,790**	2,156	33.2
s9234.1	**507,096**	271,595	360	48,559	**292,813**	1,688	73.2
c2670 .1208gat	**89,421**	51,101	13	2,366	**52,668**	112	69.8
c6288							
.3552gat	**2,185**	673	1	**1,752**	1,166	4	24.7
.3895gat	**6,204**	1,625	2	**4,080**	2,757	12	52.1
.4241gat	**17,961**	3,872	7	**9,754**	6,553	46	84.1
.4591gat	**50,342**	9,812	22	**24,378**	16,552	168	106.5
.4946gat	**146,653**	22,032	67	**62,320**	39,111	688	135.3
.5308gat	**407,042**	58,263	215	**154,782**	100,768	2,517	163.0
.5672gat	**1,184,677**	127,595	891	**389,474**	234,558	10,306	204.2
.lower9	**8,694**	3,364	2	5,141	**5,530**	14	57.2
.lower10	**23,898**	8,152	8	12,646	**13,307**	50	79.6
.lower11	**64,873**	19,282	25	31,832	**32,249**	181	101.2
.lower12	**178,442**	44,664	74	**79,276**	76,922	616	125.1
.lower13	**485,474**	102,459	218	**202,250**	184,281	2,199	140.0
.lower14	**1,347,382**	235,984	698	**509,028**	441,746	7,758	164.7
.partial	**541,793**	71,424	344	**168,324**	119,317	2,387	221.9

variable ordering. Although this saves time, it makes it often impossible to leave a local minimum and to overcome the hills on the way to the better order.

The global rebuilding opens a new way in dynamic reordering. Our experiments run according to the following scheme. We start by symbolically simulating a given circuit to obtain OBDDs of a given initial order. In addition, we have set a limit for the total size of all current OBDDs. When we reach this limit, we try to reduce the size with a rebuilding step as follows: First, we collect the functions, respectively circuit gates, we need to continue the symbolic simulation (i.e., gates whose fanouts are still to be used as inputs into other gates and the already computed outputs) – the so-called *frontier*. Next, we take the ten gates of the frontier with the biggest OBDD–representations. Then we use the same heuristic that provided the initial order to determine a new order for this modified circuit, and attempt to rebuild all OBDDs to this new order. If it turns out that the total size gets smaller, we continue to work with the new order until we reach the limit again; If the size could not be reduced, we stay with the old order (and its OBDDs) and increment the limit by multiplying it with 1.5. The

Table 2. Speedups obtained by constructing OBDDs from other OBDDs by rebuilding rather than from circuits by symbolic simulation. All experiments run on SUN ELC. The time is given in seconds.

	Symbolic Simulation		Rebuilding		
Circuit Name	Target Heuristic 1	Time 1	initial Heuristic 2	Time 2	Gain %
c6288.2877gat	W0	0.9	d0r	0.5	80.0
			d0sr	0.5	80.0
	d0	1.5	W0	0.8	87.5
			W0r	0.3	400.0
			d0s	0.3	400.0
	d0s	1.1	W0r	0.3	266.7
			d0	0.3	266.7
c6288.3211gat	d0r	1.6	W0	0.8	100.0
			d0sr	0.7	128.6
	d0sr	1.6	W0	0.9	77.8
			d0r	0.6	166.7
c6288.4946gat	W0r	174.6	d0	93.0	87.7
			d0s	70.6	147.3
	d0	216.8	W0r	69.3	212.8
			d0s	62.6	246.3
	d0r	109.0	d0sr	62.9	73.3
	d0s	211.9	W0r	69.3	205.8
			d0	63.9	231.6
s386	d0	1.4	W0	0.7	100.0
	W0	1.4	d0	0.7	100.0
s510	d0	1.6	W0	0.4	300.0
	W0	1.6	d0	0.4	300.0
s1494	d0	6.2	W0	2.8	121.4
	W0	5.9	d0	2.0	195.0
s5378	W0	78.0	d0	63.0	23.8
s35932	d0	1706.9	W0	217.7	684.0

results of various such runs are listed in Tab.3. The initial limit in these runs was set as indicated by the factor γ – the percentage with respect to the $M(i)$ observed in the respective runs without rebuilding. Tests with other settings of the initial limit are underway. The obtained results show a *decrease of the total memory requirements* in the course of the symbolic simulation.

The size of the current OBDD does not necessarily increase all the time. For a lot of circuits, the space requirements will reach a maximum during the symbolic simulation and then drop to a final size which can be considerably lower. With the method described above, the final OBDDs are computed according to an order which was created with respect to another set of circuit gates as primary outputs. This fact explains the differences between the final sizes of OBDDs obtained by the symbolic simulation with a fixed and with dynamically changed order. Surprisingly, we found that in all but one run of the simulation with the dynamic rebuilding the *final size has decreased* (Tab.3).

Due to the limited resources available, we could not obtain more than a very basic first scan of the possibilities of this approach. We currently develop software which, in the case of an unsuccessful rebuilding attempt, will use the next from a *list* of heuristics to try another order. The desired effect should be significantly increased by this method. A further improvement could be a reasonable well-behaving mechanism which will allow automatic determination of all possible parameters.

References

[BCW80] M. Blum, A. K. Chandra, M. N. Wegman: Equivalence of Free Boolean Graphs Can Be Decided Probabilistically in Polynomial Time, Inf. Process. Lett. 10, 2 (Mar.), 80–82, 1980.

[BMS94] J. Bern, Ch. Meinel, A. Slobodová: $\mathrm{T\overline{RUS}T}$ – a Programming Environment for the FBDD–package, in print, Univ. Trier, 1994.

[BRB90] K. S. Brace, R. L. Rudell, R. E. Bryant: Efficient Implementation of a BDD Package, Proc. of 27th ACM/IEEE Design Automation Conference (Orlando, June), 40–45, 1990.

[BRKM91] K. M. Butler, D. E. Ross, R. Kapur, M. R. Mercer: Heuristics to Compute Variable Orderings for Efficient Manipulation of Ordered Binary Decision Diagrams. Proc. 28th ACM/IEEE DAC, 417–420, 1991.

[Bry86] R. E. Bryant: Graph-Based Algorithms for Boolean Function Manipulation, IEEE Trans. Comput. C-35, 6 (Aug.), 677–691, 1986.

[BW94] B. Bollig, I. Wegener: Improving the Variable Ordering of OBDDs is NP-Complete, Forschungsbericht Nr. 542, 1994.

[FFK88] M. Fujita, H. Fujisawa, N. Kawato: Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams. Proc. of IEEE ICCAD, 2–5, 1988.

[FMK91] M. Fujita, Y. Matsunaga, T. Kakuda: On Variable Ordering of Binary Decision Diagrams for the Application of Multi-Valued Logic Synthesis. Proc. of EDAC, 50–53, 1991.

[FS90] S. J. Friedman, K. J. Supowit: Finding the Optimal Variable Ordering for binary Decision Diagrams. IEEE Trans. on Computers 39, 710–713, 1990.

[MIY90] S. Minato, N. Ishiura, S. Yajima: Shared Binary Decision Diagrams with Attributed edges for Efficient Boolean Function Manipulation. Proc. of the 27th ACM/IEEE Design Automation Conference, 52–57, 1990.

[MKR92] M. R. Mercer, R. Kapur, D. E. Ross: Functional Approaches to Generating Orderings for Efficient Symbolic Representation. Proc. of 29th ACM/IEEE DAC, 614–619, 1992.

[MWBS88] S. Malik, A. Wang, R. K. Brayton, A. Sangiovanni–Vincentelli: Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment, Proc. of the IEEE International Conference on Computer-Aided Design (Santa Clara, Calif., Nov.), 6–9, 1988.

[MS94] Ch. Meinel, A. Slobodová: On the Complexity of Constructing optimal Ordered Binary Decision Diagrams, Proc. of MFCS'94, LNCS 841, 515–524, 1994.

[Rud93] R. Rudell: Dynamic Variable Ordering for Ordered Binary Decision Diagrams. Proc. of IEEE ICCAD, 42–47, 1993.

[SW94] P. Savický, I. Wegener: Efficient Algorithms for the Transformation Between Different Types of Binary Decision Diagrams. Proc. FST&TCS, 1994.

Table 3. Reduction of the final and intermediate sizes by intermittent rebuilding. The experiments run on SPARCstation 10. The initial limit is set to γ% of the $M(i)$ without rebuilding. The bold-faced figures indicate improved sizes.

Circuit Name	i	without Rebuilding $M(i)$	without Rebuilding $m(i)$	without Rebuilding Time	γ	intermittent Rebuilding $M(i)$	intermittent Rebuilding $m(i)$	intermittent Rebuilding Time	Gain (%) M	Gain (%) m
c1908	W0	41,560	25,384	30	90	**39,784**	**16,926**	886	44	499
c5315	d0	32,906	32,842	33	80	**27,876**	**23,900**	85	180	374
s1196	W0	2,171	1,956	1	50	**1,446**	**1,381**	4	501	416
s1238	d0	2,284	2,020	1	50	**1,534**	**1,234**	5	488	636
s1238	W0	2,375	1,819	1	50	**1,661**	**1,351**	6	429	346
s344	d0	164	158	0	50	**142**	**129**	2	154	224
s382	d0	245	215	0	50	**189**	**181**	2	296	187
s382	W0	272	266	0	70	**198**	**191**	2	373	392
s420.1	d0	713	320	0	30	**480**	320	2	485	0
s420.1	W0	672	596	0	50	**355**	**195**	3	892	2056
s444	d0	241	200	0	70	**190**	**178**	2	268	123
s444	W0	251	235	0	90	**228**	**168**	2	100	398
s526	d0	380	360	0	70	**299**	**281**	1	270	281
s526n	d0	377	353	0	70	**301**	**279**	1	252	265
s713	W0	2,093	2,053	0	30	**960**	**780**	6	1180	1632
s9234.1	d0	48,559	20,935	42	70	**47,086**	**19,192**	135	31	90
s9234.1	W0	507,096	271,337	379	30	**157,425**	**75,365**	1,862	2221	2600
s9234.1	W0	507,096	271,337	379	70	**490,386**	**31,076**	3,521	34	7731
c2670 .1208gat	d0	2,366	1500	0	50	**1,454**	**900**	7	627	666
c880 .877gat	d0	1,013	794	0	50	**841**	**721**	2	204	101
c6288										
.lower10	d0s	27,085	12,018	12	10	**19,685**	**6,481**	53	37.6	85.4
.lower11	d0s	54,942	25,819	32	10	**40,081**	**14,225**	392	37.1	81.5
.lower12	d0s	141,340	66,271	78	10	**86,323**	**36,288**	495	63.7	82.6
.lower13	d0s	601,358	181,068	352	10	**195,276**	**85,217**	865	208.0	112.5
.lower14	d0s	1,318,013	348,382	853	10	**488,908**	**211,262**	4,588	169.7	64.9
.partial	W0	541,793	71,424	224	10	**213,582**	**63,390**	1,274	153.7	12.7

[THY93] S. Tani, K. Hamaguchi, S. Yajima: The Complexity of the Optimal Variable Ordering of a Shared Binary Decision Diagram. Proc. 4th ISAAC, LNCS 762, 389–398, 1993.

[TI94] S. Tani, H. Imai: A Reordering for an Ordered Binary Decision Diagram and an Extended Framework for Combinatorics of Graphs. Proc. of 5th ISAAC, LNCS, 1994.