published in *Proceedings of the IEEE/ACM Int. Workshop of Logic and Synthesis (IWLS2001)*, Lake Tahoe, CA, USA, June 2001, pp. 304-309.

Heuristics for ⊕-OBDD Minimization

Christoph Meinel, Harald Sack
FB IV - Informatik, Universität Trier
D-54286 Trier, Germany
{meinel,sack}@uni-trier.de

Abstract

Ordered Binary Decision Diagrams (OBDDs) have already proved useful in the process of electronic design automation. Due to limitations of the descriptive power of OBDDs more general models of Binary Decision Diagrams have been studied. In this paper, ⊕-OBDDs as a true extension of the OBDD data structure are addressed. One important factor for the representation size of ⊕-OBDDs is determined by the number and the position of the introduced ⊕-nodes. Based on a greedy strategy a heuristic for improving the representation size of ⊕-OBDDs is developed and its efficiency is proven by symbolic simulation of a standard benchmark set.

1. Introduction

A major problem in the computer aided design of digital circuits is the choice of a suitable representation of the circuit functionality for the computer's internal use. A concise representation, which simultaneously provides the possibility of fast manipulation is very important for all problems given in terms of switching functions. During the last decade, Ordered Binary Decision Diagrams (OBDDs) have proved to be well qualified for this purpose (for an overview see [MT98]).

But, the descriptive power of OBDDs is limited, due to their property of being a canonical representation for Boolean functions. This important quality is responsible for the nice algorithmic properties of OBDDs. But, on the other hand, the OBDD representation for most Boolean functions must be of exponential size w.r.t. the number of input variables, and not every Boolean function of practical importance can be represented efficiently. E.g., the OBDD-representations of the *multiplication* or the *hidden weighted bit function* are always of exponential OBDD-size [Bry91]. Therefore, generalizations of the BDD data structure have been studied.

In this paper we address ⊕-OBDDs (also known as Mod2-OBDDs), a true extension of OBDDs [GM96]. ⊕-OBDDs are more, sometimes even exponentially more, space-efficient than OBDDs are. They preserve the algorithmic properties of OBDDs: important operations as *apply, quantification*, and *composition* have the same complexity as in the case of OBDDs. Even better, the Boolean functions *exclusive or* (EXOR) and *logical equivalence* (EQU) can be performed in constant time.

However, ⊕-OBDDs do not provide a canonical representation of Boolean functions and therefore, equivalence can only be tested fast probabilistically [GM96]. The fastest known deterministic equivalence test requires time $O(|P|^3)$ [Waa97], with $|P|$ denoting the number of nodes of the ⊕-OBDD P, and thus, it is too slow for any application in practice.

The representation size of ⊕-OBDDs does not only depend on the chosen variable order as in the case of OBDDs. To make use of their full potential, also the number and the position of introduced ⊕-nodes is rather important [MS00].

In this paper we introduce a heuristic based on dynamic adjustment of ⊕-node positions for reducing ⊕-OBDD size, called the *jiggle*-heuristic. Similar as in the case of exchanging adjacent variables in OBDDs, ⊕-nodes can be exchanged with adjacent branching nodes [HDB97]. But, for performing this task efficiently with ⊕-OBDDs, it is rather important to consider some specific implementational details [MS99]. The jiggle-heuristic can be applied after the synthesis of the ⊕-OBDD is finished, but also during synthesis time, for avoiding peak memory sizes that otherwise, would exceed given resource limitations. For giving proof about the efficiency of the heuristic, it is applied to symbolic simulation of a set of standard benchmarks [LGS].

The paper is structured as follows: In Section 2, we recall basic definitions concerning ⊕-OBDDs. Section 3 covers the exchange operation of branching nodes and ⊕-nodes and discusses the requirements for an efficient implementation. Section 4 introduces the jiggle-heuristic for ⊕-OBDD minimization. Section 5 concludes with a discussion of achieved experimental results and gives an outlook on future work.

2. ⊕-OBDDs - an Overview

Definition of the Data Structure

A ⊕-OBDD P over a set $X_n = \{x_1, \ldots, x_n\}$ of Boolean variables is a directed acyclic connected graph $P = (V, E)$. V is the set of nodes, consisting of non-terminal nodes with out-degree 2, and of terminal nodes with out-degree 0. There is a distinguished non-terminal node, the *root*, which, as only node, has the in-degree 0. The two terminal nodes with no outgoing arcs are labeled with the Boolean constants 0 and 1. The remaining nodes are either labeled with Boolean variables $x_i \in X_n$ (*branching nodes*), or with the binary Boolean operator EXOR (⊕-*nodes*). On each path, every variable must occur at most once. In the following, let $l(v)$ denote the label of the node $v \in V$ and $|P|$ the number of non terminal nodes of P.

$E \subseteq V \times V$ denotes the set of edges. The two edges starting in a branching node v are labeled with 0 and 1. The *0(1)-successor* of node v is denoted by $v_0(v_1)$. There is a permutation π, which defines an order $x_{\pi(1)} < x_{\pi(2)} < \ldots < x_{\pi(n)}$ on the set of input variables. If w is a successor of v in P with $l(v), l(w) \in X_n$, then $l(v) < l(w)$ according to π must hold.

Note that since the $\oplus$-operation is symmetric, the outgoing edges of $\oplus$-nodes do not have to be labeled separately. The function f_P associated with the $\oplus$-OBDD P is determined in the following way: For a given input assignment $a = (a_1, \ldots, a_n) \in \{0,1\}^n$, the Boolean values assigned to the leave nodes are extended to all other nodes of P as follows:

- Let v_0 and v_1 be the successors of v, carrying the Boolean values $\delta_0, \delta_1 \in \{0,1\}$.
- If v is a branching node, $l(v) = x_i \in X_n$, then v is associated with δ_{a_i}.
- If v is a $\oplus$-node, then v is associated with $\oplus(\delta_0, \delta_1) = (\delta_0 + \delta_1) \bmod 2$.

The function $f_P(a)$ computes to the value associated with the source of P.

To achieve a more compact representation, we may furthermore consider the use of complemented edges [MB88].$\oplus$-OBDDs do not provide a canonical representation of Boolean functions, i.e. there might be several different representations $P_f^1, \ldots P_f^k$, $k \in \mathbb{N}$ for the same Boolean function f.

Probabilistic Equivalence Test

Since a deterministic equivalence test for $\oplus$-OBDDs requires runtime $O(|P|^3)$ [Waa97], for practical applications we have to choose a faster method. A probabilistic equivalence test for $\oplus$-OBDDs proposed in [GM93] requires only linear many arithmetic operations in the number of variables. It is based on a probabilistic equivalence test for *read-once branching programs* (BP1), originally introduced in [BCW80] and further refined in [JAB+92]. Equivalence of two $\oplus$-OBDD s is determined probabilistically, after an algebraic transformation of the $\oplus$-OBDDs in terms of polynomials over a finite field. For a more detailed description see [MS00].

Reduction and Synthesis of $\oplus$-OBDDs

The reduction rules that, if exhaustively applied, guarantee a canonical representation for OBDDs are also extended for $\oplus$-OBDDs. Here, these rules serve only for a reduction in size and do not provide any canonicity. Additionally to OBDD reduction rules, reductions for $\oplus$-nodes have to be considered [MS00].

For $\oplus$-OBDD-synthesis the already known *ite*-algorithm [BRB90] used for OBDDs can be easily extended. There, for computing $f \oplus g$ or $f \equiv g$, a new $\oplus$-node will be created and connected to f and g, where $f \equiv g = \overline{f \oplus g}$. In all other cases, the regular *ite*-algorithm is applied with an adapted cofactor creation algorithm for $\oplus$-OBDDs, where, for the computation of the cofactor f_{x_i} of a function f associated with a $\oplus$-node v_f according to a variable x_i, in some cases, the allocation of a new $\oplus$-node $v_{f_{x_i}}$ is required that is connected to the cofactors of the left and right successor of v_f [MS00].

But, for symbolic simulation, if the circuit under consideration does not contain any EXOR(EQU) gate, the extended *ite*-algorithm that is based on the recursive application of the Boole-/Shannon decomposition for Boolean functions would only create an OBDD instead of a $\oplus$-OBDD. To benefit from the potential of $\oplus$-OBDDs, somehow $\oplus$-nodes have to be introduced into the data structure. This can be done by using alternative functional decompositions including the application of XOR, as e.g. the positive or negative Davio expansion (pDE/nDE), also referred to as Reed-Muller expansion [Mul54, Ree54]. There, each EXOR operator can directly be translated to a $\oplus$-node.

$$\begin{array}{lcl} \text{pDE: } f & = & f|_{x=0} \oplus x(f|_{x=1} \oplus f|_{x=0}) \\ \text{nDE: } f & = & f|_{x=1} \oplus \overline{x}(f|_{x=1} \oplus f|_{x=0}). \end{array}$$

3. Dynamic Relocation of $\oplus$-Nodes

As shown in [MS00] not only the chosen variable order, but also the number and the position of the introduced $\oplus$-nodes determines heavily the size of $\oplus$-OBDDs. This motivates the development of methods for changing the position of already existing $\oplus$-nodes and thus, the development of heuristics for $\oplus$-OBDD minimization by applying these relocation techniques.

As shown in [HDB97], the exchange of arbitrary binary operator nodes with branching nodes remains a local operation and therefore, is efficiently computable. For $\oplus$-OBDDs, we have to adapt this algorithm, because often it might be the case that $\oplus$-nodes are not only adjacent to branching nodes. But, on a path between any two branching nodes labeled with adjacent variables there might be an arbitrary number of $\oplus$-nodes. It was shown in [MS99] that for reasons of efficiency, adjacent $\oplus$-nodes can be merged together to so called meta-$\oplus$-nodes. Meta-$\oplus$-nodes are nodes representing an EXOR operation on an arbitrary number of successors. To ensure the efficiency of meta-$\oplus$-nodes the following rule has to be maintained: On a path between any two branching nodes of adjacent variables there must be at most one meta-$\oplus$-node.

If this rule is maintained, additional reduction rules can be applied and it enables the application of dynamic variable exchange techniques. But, it also speeds up the computation of cofactors and keeps algorithms for dy-

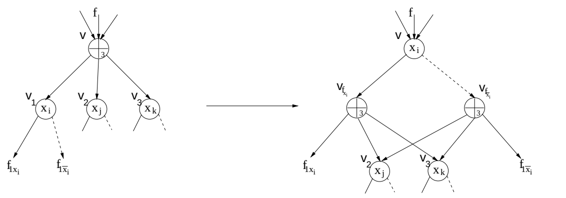

Figure 1: Meta-⊕-Node Swap Down Operation.

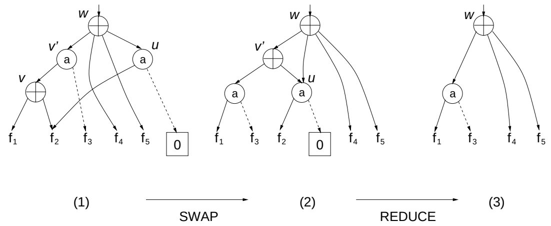

Figure 2: Non Reversible SWAP Operation.

namic restructuring of ⊕-OBDDs much simpler.

For exchanging a meta-⊕-node with an adjacent branching node, we have to proceed in the following way: Given a ⊕-OBDD P representing the Boolean function f_P, with the meta-⊕-node v as the root node. For the variables x_i, x_j, x_k of the support set of P it holds that $x_i < x_j < x_k$. W.l.o.g. let v be a meta-⊕-node with $n = 3$ successors. The three successors v_1, v_2, and v_3 of v are branching nodes labeled with x_i, x_j, and x_k, respectively. Now, we exchange the meta-⊕-node v with its successor x_i (*swap_down*), which comes first w.r.t. the variable order. v itself is only relabeled and transformed into a branching node labeled with x_i and connected to its two new successors $v_{f_{x_i}}$ and $v_{f_{\overline{x_i}}}$, both being meta-⊕-nodes with $n = 3$ successors each, at least if no reduction rule can be applied. $v_{f_{x_i}}$ is connected with the 1-successor of v_1, or, in the arbitrary case, with every 1-successor of a former successor of v labeled with x_i, and also with v_2, v_3, i.e. every other former successor of v not labeled with x_i. $v_{f_{\overline{x_i}}}$ then is connected with the 0-successors of all former successor nodes of v labeled with x_i and with all other former successors of v not labeled with x_i (see Fig. 1).

But, what, if a successor node of the branching nodes v_i is a meta-⊕-node? By performing a regular exchange, two meta-⊕-nodes would be adjacent afterwards. Thus, the exchange procedure has to be extended to joining meta-⊕-nodes automatically that occur as being adjacent after an exchange operation. The upwards exchange of a meta-⊕-node with a branching node (*swap_up*) works in a similar way, besides now, the case has to be considered that the meta-⊕-node might have meta-⊕-node predecessors at its new position that also have to be merged.

4. A Heuristic for ⊕-OBDD Minimization

On the basis of the exchange operation for meta-⊕-nodes and branching nodes a simple heuristic for ⊕-OBDD minimization can be developed. The heuristic uses local effects achieved by considering a single ⊕-node swap. If this swap results in a reduction of ⊕-OBDD size, then, this swap operation is confirmed or reversed, otherwise. But, is it really necessary to perform the expensive swap operation in both directions, if possibly no benefits will be achieved? By keeping the fixed rule that between two adjacent branching nodes, there must be at most one meta-⊕-node, possible merge operations between two adjacent meta-⊕-nodes after performing a swap operation might occur. Note that this merge operation is mandatory for achieving all possible reductions. But, if, after a merge operation no reduction can be achieved, the split of two already merged ⊕-nodes will be difficult, because information is already lost. Thus, we have to consider that the exchange-operation is not always symmetric, i.e. there are some cases, where swapping a meta-⊕-node back and forth might result in a different ⊕-OBDD while representing the same Boolean function. See Fig. 2 for an example: After a swap-up operation for meta-⊕-node v (1) the resulting meta-⊕-node v' (2) has to be merged with the predecessor meta-⊕-node w above (3). For considering any changes in size, reductions are taking place and thus, information on the sub-⊕-OBDD rooted by node u is lost.

A solution of that problem is the simulation of the swap operation, without really performing it's effects on the involved nodes. Thus, the ⊕-OBDD size after a possible swap operation can be computed. Only, if there is a decrease in size the real swap operation is performed. In that way, we have already designed a simple greedy heuristic for achieving a local minimum in ⊕-OBDD size by changing the position of already created ⊕-nodes.

The efficiency of the algorithm can be adapted by using a threshold parameter α that decides, whether a swap operation will be carried out or not. Let $|P_{new}|$ denote the size of the ⊕-OBDD after a possible swap operation and let $|P_{old}|$ denote the original size, respectively. Then, the proposition that has to be fulfilled for performing a swap operation is

$$\text{do swap, if } |P_{new}| \leq \alpha \cdot |P_{old}|$$

If $\alpha = 1$, then the original designed greedy algorithm is carried out. If $\alpha < 1$, then a swap will only be performed, if the achieved reduction in size is more relevant. Otherwise, if $\alpha > 1$, the algorithm is no longer greedy anymore and also swaps that don't directly lead to a reduction in size will be carried out. This gives way for designing an algorithm that is also able to leave an already achieved local minimum again, but, on the other hand, the $\oplus$-OBDD might also grow in size.

For each meta-$\oplus$-node of the $\oplus$-OBDD we try to find a better position by jiggling it up and down. The resulting algorithm will be referred to as the *jiggle*-algorithm. The jiggle-algorithm receives the threshold factor α as an input parameter and steps through each level i of the $\oplus$-OBDD starting from the root level. For each meta-$\oplus$-node v in level i a swap_up operation is simulated and if the computed resultant size $|P_{new}|$ is smaller than the original size $|P_{old}|$ multiplied by the threshold factor α, the swap_up operation really will be carried out. Otherwise, a swap_down operation is simulated and will only be carried out, if a positive effect can be expected. See Fig.3 for an outline of the jiggle-algorithm in pseudo code.

```
Input: ⊕-OBDD P_f with top node v, threshold factor α
Output:⊕-OBDD P'_f

jiggle(P_f, α ) {
  for all levels i do {
   for all nodes v in level i do {
    if (v is meta-⊕-node) {
     new=|P_f| after simulated swap_up(v);
     if(new < α·actual size of P_f) {
      swap_up(v);
     } else {
      new=|P_f| after simulated swap_down(v);
      if(new < α·actual size of P_f) {
       swap_down(v);
      }
     }
    }
   }
  }
  return(P_f);
}
```

Figure 3: Sketch of the Jiggle-Algorithm for $\oplus$-OBDD Minimization.

Furthermore, the concept of the jiggle algorithm can also be utilized during $\oplus$-OBDD synthesis. Its purpose simply is to enable the construction of $\oplus$-OBDDs that can not be computed within the given resource limitations otherwise. By reducing the size of the already constructed part of the $\oplus$-OBDD under consideration, we are able to continue synthesis beyond the previously stated resource limits.

In principle the concept of the *dynamic jiggle algorithm* can be defined in the following way: Let P be a $\oplus$-OBDD and let $\alpha \in \mathbb{N}$, $\alpha > 0$ be a fixed threshold value.

(1) Start regular $\oplus$-OBDD Synthesis of $|P|$.

(2) If $|P| > \alpha$, interrupt the synthesis and start the jiggle algorithm.

(3) Continue the synthesis for $|P|$, chose a new threshold value $a' > a$, and goto step (2).

5. Experimental Results and Conclusion

For showing the efficiency of the jiggle-algorithm, we have chosen the symbolic simulation of a subset of the LGSynth'93 [LGS] benchmarks. All experiments were computed on an Intel Pentium III 500 MHz based Linux system. Memory size is limited to 200 MB and computation time to 2 cpu hours. Circuits that are resulting in OBDDs less than 100 nodes or that are exceeding the given resource limitations are excluded. The variable order for all circuits was kept fixed for showing the effect of dynamic $\oplus$-node placement and reflects the order of the inputs given with the circuit description. It would have been sufficient to limit the number of signatures used for identifying $\oplus$-OBDDs to $n = 2$, but for reasons of security $n = 3$ was chosen. After successful symbolic synthesis, the jiggle algorithm was applied to the $\oplus$-OBDD until no further improvement could be achieved.

In Table 1 the results for the regular jiggle-algorithm are listed. Note that due to space limitations not all results have been put into the table, but the given overall size refers to the complete set. The first column gives the circuit's name, while in the second column the OBDD size is given for a reference. Note that all sizes are given in bytes, since the plain number of nodes is not significant anymore for $\oplus$-OBDDs, because they include nodes with more than two successors. Column 3 to 6 give the $\oplus$-OBDD sizes, while column 3 gives the size for the exclusive application of the nDE-based synthesis serving as a starting point for the heuristic. Column 4 to 6 give the $\oplus$-OBDD sizes for the ongoing application of the jiggle algorithm. The last row summarizes all achieved sizes per column and gives the percentage related to the starting point of the heuristic as a reference. A dash inside a cell denotes that for this round the jiggle heuristic did not achieve any further improvement.

Note that the computation time for $\oplus$-OBDDs is much longer compared to the computation time of plain OBDDs. This is due to the fact that for each new node 3 signatures have to be computed. For $\oplus$-OBDDs this operation is more complicated compared to OBDDs, because the computation has to be carried out in a polynomial ring over a Galois field and not only in an arbitrary finite field [GM96]. Also, the computation of the cofactor for $\oplus$-OBDDs is more time consuming. For this reason, the computation time is not given explicitely in the table, but, overall it requires up to 5 times the time required for OBDDs.

Due to the given resource limitations for the following six circuits the jiggle heuristic could not finish successfully: *C880, comp, rot, mm9a, mm9b, mult16a*. In the average the first application of the jiggle heuristic results in a 9.8% gain in size. In the next round additional 6.0% are achieved, while in the last application is only less than 0.1% are gained. Thus, the overall improvement in size comes to 13.8% in the average. While for single circuits as $s499$ the improvement of the first application of jiggle is up to about 50%, for most of the other circuits the improvement is much less.

Due to the strictly greedy nature of the algorithm the search space for positioning the $\oplus$-nodes is rather limited. Although the improvement obtained by the jiggle algorithm is not rather significant, it can be really useful, if it is applied during synthesis. In that way, exceedingly high peak memory sizes can be avoided.

In Table 2 the results for the dynamic application of the Jiggle Algorithm are given. The threshold parameter α_0 was set to 100000 Bytes and it was doubled, each time it was exceeded. The six circuits that could not be fully computed with the regular jiggle-algorithm, now, can be computed within the given limitations. This confirms the obvious assumption that applying the optimization routine during construction of the $\oplus$-OBDD decreases the required peak memory sizes. Because the optimization already did take place during construction of the $\oplus$-OBDD the achieved sizes for the dynamic jiggle-algorithm are related to the OBDD size and here, the overall reduction in sizes ranges from 13.4 % up to 19.1 % in the average for dynamic jiggle with further optimization by regular jiggle.

Although the improvement gained by the application of the jiggle algorithm comes only to at most 20 % of the final size of the regular OBDD, it is still useful for avoiding memory peaks, if it is applied dynamically during synthesis. Thus, giving a proof of efficiency of the $\oplus$-OBDD concept not only in theory.

References

[BCW80] M. Blum, A. K. Chandra, M N. Wegman, Equivalence of Free Boolean Graphs can be decided Probabilistically in Polynomial Time, *in Information Processing letters* **10**, No. 2, 1980, 80-82.

[BRB90] K. S. Brace, R. L. Rudell, R. E. Bryant, Efficient Implementation of a BDD Package, *in Proc. of the 27th ACM/IEEE Design Automation Conf.*, 1990, 40-45.

[Bry91] R. E. Bryant, On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Applications to Integer Multiplication, *in IEEE Trans. on Computers* **40** Vol. 2, 1991, 205-213.

[GM93] J. Gergov, Ch. Meinel, Frontiers of Feasible and Probabilistic Feasible Boolean Manipulation with Branching Programs, *in Proc. 10th Annual Symp. on Theoretical Aspects of Computer Science*, **665** of LNCS, Springer, 1993, 576-585.

[GM96] J. Gergov, C. Meinel, Mod2-OBDDs: A Data Structure that generalizes EXOR-sum-of-products and Ordered Binary Decision Diagrams, *in Formal Methods in System Design* **8**, Kluwer, 1996, 273-282.

[HDB97] A. Hett, R. Drechsler, B. Becker, Reordering Based Synthesis, *in Proc. of the 3rd Int. Workshop on Applications of the Reed-Muller Expansion in Circuit Design (RM'97)*, Oxford, UK, 1997, 13-22.

[JAB+92] J. Jain, M. Abadir, J. Bitner, D. S. Fussell, J. A. Abraham, IBDDs: An Efficient Functional Representation for Digital Designs, *in Proc of the European Conference on Design Automation* (1992), 440-446.

[LGS] LGSynth93 Benchmarks:
`http://www.cbl.ncsu.edu/CBL_Docs/lgs91.html`.

[MB88] J.-C. Madre, J.-P. Billon, Proving Circuit Correctness using Formal Comparison between Expected and Extracted Behavior, *in Proc. 25th ACM/IEEE Design Automation Conference (Anaheim, CA)*, 1988, 205-210.

[MS99] C. Meinel, H. Sack, Algorithmic Considerations of $\oplus$-OBDD Reordering, *in Proc. of the 4th International Workshop on Applications of the Reed-Muller Expansion in Circuit Design (Victoria, BC, Canada)*, 1999, 197-184.

[MS00] C. Meinel, H. Sack, Mod2OBDDs - a BDD Structure for Probabilistic Verification, *in Electronic Notes in Theoretical Computer Science*, vol.**22**, 2000.

[MT98] Ch. Meinel, T. Theobald, Algorithms and Data Structures in VLSI Design: OBDD - Foundations and Applications, *Springer*, Heidelberg, 1998.

[Mul54] D. E. Muller, Application of Boolean Algebra to Switching Circuit Design and Error Detection, *in IRE Trans. on Electronic Computing* **EC-3**, 1954, 6-12.

[Ree54] L. S. Reed, A Class of Multiple Error-Correcting Codes and their Decoding Scheme, *in IRE Trans. on Information Theory* **4**, 1954, 38-42.

[Waa97] S. Waack, On the Descriptive and Algorithmic Power of Parity Ordered Binary Decision Diagrams, *in Proc. 14th Symp. on Theoretical Aspects of Computer Science*, **1200** of LNCS, Springer, 1997.

Circuit		⊕-OBDD size [Bytes]			
	OBDD	**nDE-meta**	**Jiggle (1)**	**Jiggle (2)**	**Jiggle (3)**
sbc	133740	135892	117584	117220	-
s967	62352	27464	23612	23564	-
s820	95436	16572	13856	13460	-
s713	48672	115788	98272	98128	-
s641	48672	115720	98244	98100	-
s635	23616	64692	42688	42688	-
s510	686736	15484	12388	11872	-
s499	12096	24128	12964	-	-
s420	9440172	19932	16436	16412	-
s386	10116	9900	8552	-	-
s3271	121140	189548	165048	164676	164596
s208	37188	5092	4448	4436	-
s1512	680256	331236	300732	300732	-
s1494	36576	36196	31068	30636	-
s1488	36576	34492	30092	29584	29360
s1423	3544344	3938476	3360020	3341752	-
s1269	1734336	1101800	965772	964792	963868
s1196	82584	118968	102108	101800	101776
dsip	501156	292420	254988	238376	-
bigkey	222120	203712	184552	-	-
8085	4242276	2614032	2255804	2253144	2253004
x3	99360	76232	64520	-	-
vg2	37584	66124	52468	51368	-
vda	156420	43416	35096	34752	34632
too_large	255456	423524	371800	369712	369580
term1	20880	34232	24864	-	-
pair	2436660	3411992	3100748	3100316	-
my_adder	11796372	18873336	16252724	13369124	-
mux	4718556	2240	1896	-	-
k2	1020096	161748	150268	149432	-
i9	82008	236480	212804	-	-
i8	157176	404896	283212	283140	283104
frg2	232956	142604	119036	-	-
cm150a	4718556	2280	1876	-	-
booth8x8	229896	445108	444060	437500	437464
baugh-wooley6x6	29880	96592	87752	86652	-
apex7	59760	29080	23476	-	-
apex1	1020096	197656	187108	185165	-
alu4	42552	53772	40064	38108	-
alu32r	6813576	460920	334264	-	-
alu32	438984	31628	26384	-	-
adder16	11801232	21629848	21238784	-	-
C499	1653192	670188	671680	670072	-
C432	62388	184080	160700	160396	-
C1908	1296252	1163384	1129004	1109496	1108708
C1355	1653192	670024	671516	669908	-
Σ	**72.843.732**	**59.273.572**	**54.084.137**	**51.118.210**	**51.115.706**
	122.9%	100%	91.2 %	86.2%	86.2%

Table 1: Jiggle Heuristic for ⊕-node Placement.

Circuit		⊕-OBDD size [Bytes]			
	OBDD	**nDE-DynJiggle**	**Jiggle (1)**	**Jiggle (2)**	**Jiggle (3)**
C880	12456	10571900	10291912	10170540	10085944
comp	16513128	19746056	19663748	18741360	18741216
rot	36166674	7705076	7112688	6918432	-
mm9a	26487648	21406700	21405004	-	-
mm9b	30540926	24485432	24482980	-	-
mult16a	12975912	23822712	19286296	-	-
Σ	**194.146.880**	**168.593.280**	**156.974.776**	**155.555.840**	**155.461.524**
	100.0 %	86.6 %	80.9 %	80.1 %	80.1 %

Table 2: Dynamic Jiggle Heuristic for ⊕-node Placement.