Combination of Lower Bounds in Exact BDD Minimization

Rüdiger Ebendt
Department of Computer Science
University of Kaiserslautern
67663 Kaiserslautern, Germany
ebendt@informatik.uni-kl.de

Wolfgang Günther
CL DAT TDM VM
Infineon Technologies
81730 Munich, Germany
wolfgang.guenther@infineon.com

Rolf Drechsler
Institute of Computer Science
University of Bremen
28359 Bremen, Germany
drechsle@informatik.uni-bremen.de

Abstract

Ordered Binary Decision Diagrams (BDDs) are a data structure for efficient representation and manipulation of Boolean functions. They are frequently used in logic synthesis and formal verification. The size of BDDs depends on a chosen variable ordering, i.e. the size may vary from linear to exponential, and the problem of improving the variable ordering is known to be NP-complete.

In this paper we present a new exact branch&bound technique for determining an optimal variable order. In contrast to all previous approaches, that only considered one lower bound, our method makes use of a combination of three bounds and by this avoids unnecessary computations. The lower bounds are derived by generalization of a lower bound known from VLSI design. They allow to build the BDD either top down or bottom up. Experimental results are given to show the efficiency of our approach.

1. Introduction

There is a renewed interest in multiplexor based design styles, since often multiplexor nodes can be realized at very low cost (as e.g. Pass Transistor Logic (PTL)). In addition, these techniques allow to consider layout aspects during the synthesis step and by this guarantee high design quality [4, 8, 15, 14, 7, 18]. Furthermore, these techniques make use of ordered *Binary Decision Diagrams* (BDDs) as introduced in [2].

As is well known, the size of BDDs is often very sensitive to a chosen variable ordering. In [2] an example has been given, where the BDD size of a function varies from linear to exponential dependent on the ordering of the variables. Especially in applications like logic synthesis it is important to determine a good ordering, since a reduction in the number of BDD nodes directly transfers to a smaller chip area.

In general, determining an optimal variable ordering is a difficult problem, i.e. improving a given ordering has been proven to be an NP-complete problem [1]. For this, in the past many heuristic approaches have been proposed, that are based on structural information [10] or on dynamic reconstruction of BDDs [16]. But all these methods cannot guarantee an optimal result and experimental studies have shown that they are often up to a factor of two away from the best known solution. For the applications given above, this is a significant drawback.

Beside these heuristic approaches, exact algorithms have been studied. While the first method proposed in [9] was limited to few variables, more advanced techniques make use of an efficient representation of the information and use a branch&bound approach. Alternative lower bounds have been proposed in [12, 13, 6]. Furthermore, functional aspects, like symmetry, have been used to prune parts of the search space. Using these methods, larger functions can be handled, e.g. in [6] exact solutions for 32-bit adders have been computed. But all methods have in common that they completely rely on one lower bound technique and still runtime is a crucial factor.

In this paper we present a new exact algorithm for the computation of an optimal variable ordering. As all other exact algorithms presented so far, the core technique is based on [9] in combination with branch&bound. But in contrast to previous approaches, we make use of three lower bounds in parallel and by this can prune large parts of the search space at an early stage. The lower bounds are generalizations of bounds known from VLSI design and can be applied top down and bottom up. This gives more flexibility during the algorithm run. Experiments show that significant runtime reductions can be observed on benchmark functions, i.e. for the functions considered an improvement up to 49% has been obtained.

2. Preliminaries

Boolean variables (denoted by Latin letters) are bound to values in $\mathbf{B} := \{0,1\}$. It is well-known that a Boolean function $f\colon \mathbf{B}^n \to \mathbf{B}$ over the variable set X_n can be represented by a *Binary Decision Diagram* (BDD) [2], i.e. a directed

1530-1591/03 $17.00 © 2003 IEEE

acyclic graph where a Shannon decomposition

$$f = \overline{x}_i f_{x_i=0} + x_i f_{x_i=1} \quad (1 \le i \le n)$$

is carried out in each node. In the following, only reduced, ordered BDDs are considered and for briefness these graphs are called BDDs. Redundant nodes are assumed to be eliminated and variables are encountered at most once and in the same order (the "variable ordering") on every path from the root to a terminal node. For more details see [2].

The size of BDDs often critically depends on their variable ordering, i.e. it may vary from linear to exponential. Variables are denoted by permutations π: $X_n \to X_n$. For simplicity we write $x_i = \pi(k)$, if variable x_i is the k-th element of the variable ordering, i.e. x_i is in the k-th level of the BDD. For a set of variables $I \subseteq X_n$, let $\Pi(I)$ be the set of permutations π on X_n, whose first $|I|$ members constitute I. We extend π straightforwardly to also map subsets of X_n to subsets of X_n, i.e. I is a fixpoint for π. Further, we denote the minimal number of nodes in a BDD labeled by a variable in $I \subseteq X_n$ with min_cost_I.

BDDs are defined analogously for multi-output functions f: $\mathbf{B}^n \to \mathbf{B}^m$, using a graph for each of the m single-output functions for the *shared* BDD representation. In the following we assume shared BDDs with *Complement Edges* (CEs) without mentioning it further. (Note that all results reported here directly transfer to BDDs without CEs.) We write BDD(f,π) for the BDD with variable ordering π, which represents the Boolean function f. For a BDD F, let label(F,x_k) denote the number of nodes in F labeled by x_k. We extend this notation straightforwardly to also cover sets of variables, i.e. label$(F,I) = \sum_{x_i \in I}$ label(F,x_i).

3. Previous Work

To keep the paper self-contained, we briefly review previous approaches to exact minimization of BDDs. Since the problem of improving a variable ordering is NP-complete [1], the runtime complexity of all exact minimization algorithms presented so far has been significantly higher (i.e. exponential) than that of mere "rules of thumb" to find a "good" variable ordering, i.e. heuristics.

In [9] an approach working on truth tables has been presented, where the number of considered variable orderings has been reduced to 2^n, a significant improvement over the "naive" approach considering all $n!$ variable orderings. Next, in [12] two new ideas have been introduced: the use of BDDs instead of truth tables and the use of *upper and lower bounds* on BDD sizes. The idea is to skip examination of BDDs as soon as a lower bound on the sizes of BDDs achievable from that BDD already exceeds an upper bound for the minimal size. The problem now is seen as a *search problem* in a *state space*, where states are subsets of X_n. A branch&bound technique is used to skip large parts of the state space. This approach has been further enhanced in [13]. In [6], a lower bound known from VLSI design has been adapted for exact minimization. The approach is the fastest known so far with a speed-up factor up to 400 compared to [13].

Next we give the basic minimization algorithm following this top down approach. Suppose we minimize the BDD for a multi-output function f: $\mathbf{B}^n \to \mathbf{B}^m$. In brief, we compute the optimal variable ordering iteratively by computing for increasing k's min_cost_I for each k-element subset I of X_n, until $k = n$: then, the BDD has a variable ordering yielding a BDD size of $min\_cost_{X_n}$. This is an optimal variable ordering.

At step k of the algorithm, a state I with $|I| = k-1$ is retrieved from a hash table (which holds all states of the previous step $k-1$). The algorithm now builds transitions $I \xrightarrow{x_i} I \cup \{x_i\} =: I'$ for $x_i \in X_n \setminus I$[1]. The subject is to compute $min\_cost_{I'}$ for each successor I'. This is done by a gradual scheme of continuous minimum updates. Following [6], this scheme uses the interrelation

$$min\_cost_{I'} = \min_{x_k \in I'} \left(min\_cost_{I' \setminus \{x_k\}} + \text{label}(F_k, x_k) \right),$$

where F_k is a BDD representing f with variable ordering π_k such that $\pi_k(I' \setminus \{x_k\}) = I' \setminus \{x_k\}$ and $\pi_k(|I'|) = x_k$. This recurrent interrelation uses an argument of [9], informally: the number of nodes in a level is constant, if the corresponding variable is fixed in the variable ordering and no variables from the lower and upper part are exchanged. Note, that this holds independently of the ordering of the variables in the upper and lower part of the BDD. The terms $min\_cost_{I' \setminus \{x_k\}}$ have been saved to the hash table in the previous step $k-1$, since $|I' \setminus \{x_k\}| = k-1$. Their values are simply retrieved from the hash table. The only terms still left to compute are label(F_k, x_k) for each $x_k \in I'$.

For $I := I' \setminus \{x_k\}$, suppose F_I is a BDD representing f with a variable ordering π_I such that $\pi_I(I) = I$. Then F_k can be constructed from F_I by shifting variable x_k to the $(|I|+1)$-th level. That way, the minimized BDD is built top down, starting with the first level and, as k increases, repeatedly adding another level below the current level. In [6], the variable ordering $\pi_{I \cup \{x_k\}}$ resulting by this variable shifting of x_k is saved in a hash table for later steps. Since $I \cup \{x_k\}$ is again a fixpoint for $\pi_{I \cup \{x_k\}}$, this ordering can be used in the next step like π_I was used for I.

At the end of step k, all states of which the lower bound exceeds or equals the current upper bound, are excluded.

Remark 1 In fact the BDD can also be built bottom up following the same outline. In this case the interrelation $\Pi(I) = \Pi(X_n \setminus I)$ is used. Note, that the approaches in [12, 13] are bottom up *only*, whereas the approach in [6] is top down *only*.

4. Lower Bound Technique

In this section, we generalize a lower bound known from VLSI design such that it can be also used for bottom up construction of a minimized BDD. So the new approach is not restricted to a top down construction like the approach in [6]. Moreover, *combining* the two lower bounds yields a new lower bound, that is used to exclude states earlier than in previous approaches, resulting in a further speed-up.

[1] The transition is built only, if I' has not already been excluded (see Section 4.2).

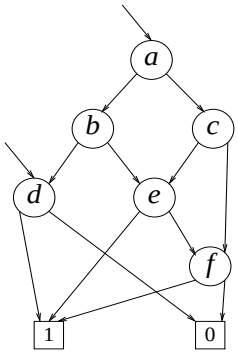

Figure 1. BDD for Example 4.1.

4.1. A Generalized Lower Bound

Before the generalized lower bound is presented, we give a brief review of the lower bound used in [6], which is an adaptation of a lower bound known from VLSI design [3].

For $k > 0$, let $\mathrm{ref}(F,k)$ denote the set of nodes in levels $k+1,\ldots,n$ of the BDD F referenced directly from the nodes in levels $1,\ldots,k$ of F. If a node has no direct, i.e. only external references, it is *not* contained in $\mathrm{ref}(F,k)$. Let $\mathrm{ref}(F,0)$ denote the set of externally referenced nodes, i.e. the set of nodes which represent user functions. The size of $\mathrm{ref}(F,0)$ is equal to the number of output nodes in F.

Example 4.1 Consider the BDD F given in Figure 1. The two outputs are represented by nodes a and d, thus $\mathrm{ref}(F,0) = \{a,d\}$. The other sets are given by
$\mathrm{ref}(F,1) = \{b,c\}$,
$\mathrm{ref}(F,2) = \{d,e,f\}$,
$\mathrm{ref}(F,3) = \{f\}$,
$\mathrm{ref}(F,4) = \emptyset$.

Lemma 4.1 Let $f\colon \mathbf{B}^n \to \mathbf{B}^m$ be a multi-output function. Let (L,R) be a partition of X_n, $\pi \in \Pi(L)$ and $F := \mathrm{BDD}(f,\pi)$. If $|\mathrm{ref}(F,|L|)| = c$, then each BDD with a variable ordering in $\Pi(L)$ representing f has at least c nodes in levels $|L|+1,\ldots,n$.

In exact minimization, this argument is used to obtain a lower bound based on $\mathrm{ref}(F,|I|)$ at each step, a state $I \subseteq X_n$ is considered. Suppose that the minimized BDD F is constructed top down. Further assume that, when computing the lower bound for a multi-output function $f\colon \mathbf{B}^n \to \mathbf{B}^m$, the minimal number min_cost_I of nodes in levels $1,\ldots,|I|$ is already known. Let $c_\alpha = |\mathrm{ref}(F,|I|)|$. Then by Lemma 4.1, the lower bound can be computed as

$$l\_b_\alpha = min\_cost_I + \max\{c_\alpha + r\_lower, n-|I|\} + 1,$$

In order not to count some output nodes twice, r_lower is the number of output nodes in levels $|I|+1,\ldots,n$ not already representing a node in $\mathrm{ref}(F,|I|)$ and $n-|I|$ is the number of variables with an index in $X_n \setminus I$, since there will be at least one node for each of these variables. The constant node is always needed.

The next result enables us to generalize this lower bound such that it can be also used for bottom up construction.

Lemma 4.2 Let $f\colon \mathbf{B}^n \to \mathbf{B}^m$ be a multi-output function. Let $I \subseteq X_n$, $\pi \in \Pi(I)$ and $F := \mathrm{BDD}(f,\pi)$. Then we have $|\mathrm{ref}(F,|I|)| - r\_upper \le min\_cost_I$, where r_upper is the number of output nodes in levels $1,\ldots,|I|$ of F.

The idea is to calculate the minimal number of nodes needed to connect the output nodes with the nodes in $\mathrm{ref}(F,|I|)$. Suppose now the minimized BDD is constructed bottom up. The BDD must respect a variable ordering in $\Pi(X_n \setminus I)$. Let $c_\omega = |\mathrm{ref}(F,n-|I|)|$. Then a lower bound can be computed as

$$l\_b_\omega = min\_cost_I + \max\{c_\omega - r\_upper, n-|I|\} + 1,$$

where r_upper is the number of output nodes in levels $1,\ldots,n-|I|$. This lower bound holds, since, by Lemma 4.2, $c_\omega - r\_upper$ is a lower bound for $min\_cost_{X_n \setminus I}$, which is the minimal size of the upper part of the BDD.

In the next section, the introduced lower bounds l_b_α and l_b_ω will be combined to a new lower bound, which is used to exclude states at an early stage of the algorithm.

4.2. Early Pruning

We describe two techniques to exclude states from further examination *as early as possible*. These techniques save a significant number of transitions from one state to another. As transitions involve variable shiftings at high computational cost, this is a significant gain for the new algorithm.

The situation, in which the techniques are applied, is briefly described. In step k, the algorithm expands all states I with $|I| = k-1$, that have not been excluded in the last step, to all possible successors. Successor states are being revisited frequently in the progress of step k, since many distinct states I, I' have successors in common. When repeatedly revisiting such a successor J, min_cost_J is gradually computed by continuously updating the previous smallest number of nodes labeled with a variable in J. This number reaches min_cost_J at the end of step k.

It is desirable to

1) avoid transitions to successors, which are already known *not* to contribute to the actualization of the smallest node number,

2) find a means, which allows to test every successor for a possible exclusion right after it was generated: in this way, unnecessary repeated movements to successor states can be avoided.

- 1): Consider a transition $I \xrightarrow{x_i} I \cup \{x_i\}$, where $I \cup \{x_i\}$ is a state, that has already been visited before.

Let F_I be the BDD for I. Since state I was processed in the previous step, we already have $\mathrm{label}(F_I, I) = min\_cost_I$. The potential successor state would be built by e.g. shifting variable x_i to level $|I|+1$, resulting in a new BDD F'. A lower bound on $\mathrm{label}(F', I \cup \{x_i\})$ is

$$l\_b_{cost} = min\_cost_I + 1,$$

since there will be at least one node labeled by the variable x_i. Note, that this lower bound can be computed in constant time.

If this lower bound is not smaller than the previous smallest number of nodes labeled by a variable in $I \cup \{x_i\}$, neither is the exact value label$(F', I \cup \{x_i\})$. In this case this transition is skipped, since revisiting this state does not contribute to the actualization of its previous minimum.
- 2): At an early stage of step k min_cost_I cannot be used to compute a lower bound, since the exact value of the minimum is not known until step k finishes. Therefore min_cost_I must be estimated.

This can be done both efficiently and effective by combining l_b_α and l_b_ω. The idea is to estimate the (yet) unknown exact part of the lower bound, i.e. min_cost_I, with the according estimated part of the opposite lower bound.

The next definition uses a partition (L,R) of X_n rather than states $I \subseteq X_n$ such that it applies for both the top down and the bottom up approach of exact minimization. In case of starting the minimization from above, we have $I = L$ at a step considering state I (starting from below, we have $I = R$ and $L = X_n \setminus I$).

A lower bound on the minimal size of the BDD achievable from a partition (L,R) can be computed as follows. Let F be the considered BDD and $c_{\alpha\omega} = |\text{ref}(F,|L|)|$.

$$\begin{aligned} l\_b_{combined} &= \max\{c_{\alpha\omega} + r\_lower, |R|\} + 1 + \\ &\quad \max\{c_{\alpha\omega} - r\_upper, |L|\}, \end{aligned}$$

where r_lower is the number of output nodes in levels $|L|+1,\ldots,n$ not already representing a node in ref$(F,|L|)$ and r_upper is the number of output nodes in levels $1,\ldots,|L|$.

All states already excluded "early" by this lower bound are marked by the new algorithm. Transitions leading to such a marked state are not followed by the algorithm, saving again the computational cost for a variable shifting.

5. Algorithm

In this section we describe the implementation techniques used in the new approach. A sketch of the new algorithm called JANUS is given in Figure 2.

The algorithm is based on the implementation of [6], a top down approach, where only one lower bound was used. We assume the presence of all techniques applied there without further mentioning. Next, we describe some new unique implementation techniques applied in the new approach.

5.1. Lower Bound Computation

To determine the lower bounds introduced in Section 4.1 for the current BDD F, $|\text{ref}(F,k)|$ must be computed, where $k = |I|$ for l_b_α and $k = n - |I|$ for l_b_ω.

Computation of $|\text{ref}(F,k)|$ is done with two different methods: one touches only the nodes in the "upper part", i.e. in the first k levels, the other touches only the nodes in the "lower part", i.e. in the last $n-k$ levels. If the size of the upper part is smaller than that of the lower part, the first routine is called (since this is more promising in terms of expected runtime) and vice versa. Since commonly used BDD packages keep and continously update the level sizes in dedicated variables, the time needed to determine the size of the upper and lower part is very small, i.e. it can be neglected.

It is sufficient to calculate the lower bounds only once the first time a state I is encountered: assuming $|I| = k$, $\pi \in \Pi(I)$ and $F := \text{BDD}(f,\pi)$, the nodes in ref(F,k) represent the cofactors in all variables in I. The number of nodes representing such a cofactor must equal the arity of the cofactors range, i.e. it does not depend on which variable ordering in $\Pi(I)$ is used for F. The algorithm gains from this as follows: the invariant terms of the lower bounds are computed separately from min_cost_I. This saves the cost of unneccessary repeated computations.

5.2. Partial BDD Reconstruction

The BDD corresponding to a state is kept in memory only until the next state is considered, since otherwise the memory requirement would be much too large. Every BDD for the next state processed (i.e. expanded to its sucessors) must be reconstructed by variable shiftings.

Reconsidering the exact minimization algorithm described in Section 3, we observe: at step k, a BDD F is appropriate to represent a state I with $I = |k-1|$, iff F has a variable ordering π such that $\pi(I) = I$.

In [6] the variable ordering used for reconstruction of a BDD for a state I is that of the BDD for I in the previous step $k-1$. This variable ordering π respects the above condition $\pi(I) = I$. The BDD is reconstructed by a series of n upward variable shiftings: from left to right, the variables $\pi(1),\ldots,\pi(n)$ are shifted to levels $1,\ldots,n$.

Now suppose, this sequence of variable shiftings is reduced to only shifting, from left to right, the variables $\pi(1),\ldots,\pi(|I|)$ to levels $1,\ldots,|I|$. This yields a variable ordering π_I, which also respects the condition $\pi_I(I) = I$. The advantage is, that much less shifting operations are needed. A problem is the higher risk of "BDD explosions": since only the "upper parts" of the partially reconstructed BDD and the old BDD for state I coincide, the node number in the "lower part" of the partially reconstructed BDD can "blow up", which would result in a slow down of runtime and an increase of memory requirement. In our approach, this problem is addressed straightforwardly: whenever 0.7 times the size of the partially reconstructed BDD exceeds the size of the old BDD, we return to the old variable ordering π, which was used to represent state I in the previous iteration. Note, that this technique transfers directly to the case of bottom up minimization.

Additionally, before reconstruction, the algorithm computes the BDD in a cache of 10 BDDs with the smallest number of variable exchanges to set a required ordering. This is done in CPU time much less than the variable exchanges in fact would require. The idea of a BDD cache was introduced in [11] and is used here in exact BDD minimization for the first time.

The new approach significantly gains efficiency by using these techniques of partial reconstruction.

6. Experimental Results

All experimental results have been carried out on an Athlon processor running at 1.4 GigaHz using an upper memory limit of 300 MByte and a runtime limit

```
(1)  compute_optimal_ordering(BDD F, int n) {
(2)      minguess[hash(∅)] := 0;
(3)      π[hash(∅)] := an arbitrary initial order;
(4)      states[hash(∅)] := ∅;
(5)      clear next_states;
(6)      for k := 1 to n do
(7)          for each I ∈ states do   /* update minguess[hash(I)] until it reaches min_cost_I */
(8)              reconstruct(F, π[hash(I)], |I|);
(9)              for each x_i ∈ X_n \ I do
(10)                 I' := I ∪ {x_i}; if I' ∉ next_states then minguess[hash(I')] := ∞;
(11)                 if I' already excluded or lb_cost_I' ≥ minguess[hash(I')] then continue with for-loop;
(12)                 shift x_i to level |I| + 1;   /* Note, that k = |I| + 1 */
(13)                 newcost := label(F, |I| + 1) + minguess[hash(I)];
(14)                 if (I' ∉ next_states or newcost < minguess[hash(I')]) then
(15)                     minguess[hash(I')] := newcost;
(16)                     π[hash(I')] := current ordering;
(17)                     upper_bound := update_upper_bound();
(18)                     if I' ∉ next_states then
(19)                         states[hash(I')] := I';
(20)                         lower_bound[hash(I')] := compute_lower_bound(I');
(21)                         if lb_combined_I' ≥ upper_bound then exclude I';
(22)                     end–if
(23)                 end–if
(24)             end–do
(25)             exclude all states I' in next_states with lower_bound[hash(I')] ≥ upper_bound;
(26)             states := next_states;
(27)         end–do
(28)         reconstruct the ordering of upper_bound;
(29)     end–do
(30) }
```

Figure 2. The new algorithm JANUS (sketch)

of 20.000 CPU seconds. The new algorithm is called JANUS↑, if minimization progresses bottom up using l_b_ω and JANUS↓, if minimization progresses top down using the lower bound l_b_α. Both approaches use the "early pruning" techniques of Section 4.2. The implementation of the new algorithm is based on the implementation of the algorithm in [6], called FizZ. Both algorithms have been integrated in the CUDD package [17], which also contains an implementation similar to the algorithm JUNON [13]. By this we guarantee that all algorithms run in the same system environment.

In a series of experiments we applied both algorithms to the set of Benchmark circuits from LGSynth93 [5]. The results are given in Table 1. In the first column the name of the function is given. *in* (*out*) denotes the number of inputs and outputs of a function. Column *opt* shows the number of BDD nodes needed for the minimal representation. In columns *time* and *space* the runtime in CPU seconds and the space requirement in MByte for JUNON and the new approach JANUS↑ as well as for the approach FizZ and the new approach JANUS↓ are given.

As the results show, the algorithm JUNON, which is the best bottom up approach known so far, has much longer runtimes than the bottom up approach JANUS↑. It can be seen that JANUS↑ often accelerates runtime by a factor up to two orders of magnitude (see e.g. *sct, pcle, tcon*) in comparison to JUNON. However, JANUS↑ has longer runtimes than the top down approaches in most cases.

The new top down approach JANUS↓ is faster than FizZ, especially for larger examples achieving a reduction in runtime by up to 49% (see e.g. *mux*). On average, the reduction in runtime is 35.4%. The results show that the new threefold lower bound technique together with efficient implementation techniques is a very robust improvement, that significantly outperforms the original algorithm FizZ.

7. Conclusions

We presented an exact algorithm for determining the optimal variable ordering for BDDs. It uses an extended branch&bound technique to prune the state space by the use of three lower bounds. This technique enables us to often avoid repeatedly visiting states. The lower bounds have been derived by generalization of a lower bound known from VLSI design. Moreover, we described a faster method of lower bound computation as well as an efficient method of partial BDD reconstruction, which avoids many time consuming variable shiftings. Experimental results are reported that clearly demonstrate the efficiency of both the presented top down and bottom up approach. A comparison to the best minimization algorithm known so far shows that runtime can be reduced by up to 49%.

Table 1. Comparison of JUNON, FizZ and JANUS.

name	in	out	opt	bottom up				top down			
				JUNON		JANUS↑		FizZ		JANUS↓	
				time	space	time	space	time	space	time	space
cc	21	20	46	–	62M	412s	40M	117s	32M	84.9s	34M
cm150a	21	1	33	–	62M	348s	35M	610s	32M	311.1s	34M
cm163a	16	5	26	5.23s	2M	1.87s	$<$ 1M	1.17s	$<$ 1M	0.78s	$<$ 1M
cmb	16	4	28	0.02s	2M	0.04s	$<$ 1M	0.01s	$<$ 1M	0.05s	$<$ 1M
comp	32	3	95	–	–	8684s	146M	5606s	99M	3900s	125M
cordic	23	2	42	11.2s	258M	5.57s	$<$ 1M	3.05s	$<$ 1M	1.82s	$<$ 1M
cps	24	102	971	–	537M	9130s	85M	4396s	48M	2751s	58M
i1	25	16	36	–	–	74.1s	19M	29.4s	9M	18.77s	10M
lal	26	19	67	–	–	3818s	256M	677s	64M	504.4s	75M
mux	21	1	33	–	62M	348s	35M	610s	32M	310.8s	35M
parity	16	1	17	$<$ 0.01s	2M	0.01s	$<$ 1M	$<$ 0.01s	$<$ 1M	0.03s	$<$ 1M
pcle	19	9	42	7584s	15M	60.9s	8M	9.02s	2M	5.18s	3M
pm1	16	13	40	1.26s	2M	0.64s	$<$ 1M	0.55s	$<$ 1M	0.34s	$<$ 1M
s208.1	18	9	41	2116s	8M	42.2s	4M	8.44s	$<$ 1M	5.62s	2M
s298	17	20	74	934s	4M	25.6s	4M	13.46s	2M	9.06s	3M
s344	24	26	104	–	537M	3476s	229M	1446s	99M	950s	105M
s349	24	26	104	–	537M	2921s	229M	1447s	99M	950s	105M
s382	24	27	119	–	537M	3318s	263M	802s	67M	461s	71M
s400	24	27	119	–	537M	3322s	263M	802s	67M	456s	71M
s444	24	27	119	–	537M	3331s	263M	779s	67M	508s	78M
s526	24	27	113	–	537M	6017s	284M	1196s	93M	924s	105M
s820	23	24	220	–	258M	2467s	76M	2034s	53M	1235s	56M
s832	23	24	220	–	258M	2541s	69M	2076s	53M	1288s	56M
sct	19	15	48	8453s	15M	53.3s	8M	8.62s	2M	5.97s	3M
t481	16	1	21	0.16s	2M	0.15s	$<$ 1M	0.16s	$<$ 1M	0.13s	$<$ 1M
tcon	17	16	25	635s	4M	6.61s	4M	0.52s	$<$ 1M	0.28s	$<$ 1M
ttt2	24	21	107	–	537M	5281s	277M	950s	74M	578s	78M
vda	17	39	478	822s	5M	99.9s	5M	65.4s	3M	34.4s	3M

References

[1] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. on Comp.*, 45(9):993–1002, 1996.

[2] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, 35(8):677–691, 1986.

[3] R. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Trans. on Comp.*, 40:205–213, 1991.

[4] P. Buch, A. Narayan, A. Newton, and A. Sangiovanni-Vincentelli. Logic synthesis for large pass transistor circuits. In *Int'l Conf. on CAD*, pages 663–670, 1997.

[5] Collaborative Benchmarking Laboratory. *1993 LGSynth Benchmarks*. North Carolina State University, Department of Computer Science, 1993.

[6] R. Drechsler, N. Drechsler, and W. Günther. Fast exact minimization of BDDs. *IEEE Trans. on CAD*, 19(3):384–389, 2000.

[7] R. Drechsler and W. Günther. *Towards One-Pass Synthesis*. Kluwer Academic Publishers, 2002.

[8] F. Ferrandi, A. Macii, E. Macii, M. Poncino, R. Scarsi, and F. Somenzi. Symbolic algorithms for layout-oriented synthesis of pass transistor logic circuits. In *Int'l Conf. on CAD*, pages 235–241, 1998.

[9] S. Friedman and K. Supowit. Finding the optimal variable ordering for binary decision diagrams. In *Design Automation Conf.*, pages 348–356, 1987.

[10] H. Fujii, G. Ootomo, and C. Hori. Interleaving based variable ordering methods for ordered binary decision diagrams. In *Int'l Conf. on CAD*, pages 38–41, 1993.

[11] W. Günther and R. Drechsler. Improving EAs for Sequencing Problems. In *Genetic and Evolutionary Computation Conference*, 175–180, 2000.

[12] N. Ishiura, H. Sawada, and S. Yajima. Minimization of binary decision diagrams based on exchange of variables. In *Int'l Conf. on CAD*, pages 472–475, 1991.

[13] S.-W. Jeong, T.-S. Kim, and F. Somenzi. An efficient method for optimal BDD ordering computation. In *International Conference on VLSI and CAD*, 1993.

[14] L. Macchiarulo, L. Benini, and E. Macii. On-the-fly layout generation for PTL macrocells. In *Design, Automation and Test in Europe*, pages 546–551, 2001.

[15] A. Mukherjee, R. Sudhakar, M. Marek-Sadowska, and S. Long. Wave steering in YADDs: A novel non-iterative synthesis and layout technique. In *Design Automation Conf.*, pages 466–471, 1999.

[16] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Int'l Conf. on CAD*, pages 42–47, 1993.

[17] F. Somenzi. *CU Decision Diagram Package Release 2.3.1*. University of Colorado at Boulder, 2002.

[18] C. Yang and M. Ciesielski. BDS: a BDD-based logic optimization system. *IEEE Trans. on CAD*, 21(7):866–876, 2002.