Fast Exact Minimization of BDDs

Rolf Drechsler   Nicole Drechsler   Wolfgang Günther

Institute of Computer Science
Albert-Ludwigs-University
79110 Freiburg im Breisgau, Germany
drechsle@informatik.uni-freiburg.de

Abstract

We present a new exact algorithm for finding the optimal variable ordering for reduced ordered Binary Decision Diagrams (BDDs). The algorithm makes use of a lower bound technique known from VLSI design. Up to now this technique has been used only for theoretical considerations and it is adapted here for our purpose. Furthermore, the algorithm supports symmetry aspects and makes use of a hashing based data structure. Experimental results are given to demonstrate the efficiency of our approach. We succeeded in minimizing adder functions with up to 64 variables, while all other previously presented approaches fail.

1 Introduction

Recently, several design methods have been proposed that are based on *ordered Binary Decision Diagrams* (BDDs) [7]. The resulting circuits have very nice properties, like e.g. testability [2, 1] and low power [17]. For synthesis approaches based on *Pass Transistor Logic* (PTL) BDDs seem to be a good starting point. First promising results on how to transform a decision diagram to a circuit based on PTL have been reported in [24, 9, 3].

One drawback of BDDs is that they are very sensitive to the variable ordering, i.e. the size of the representation may vary from linear to exponential. Therefore in the last few years several methods have been presented to determine good orderings. However, finding the optimal variable ordering starting from a given BDD representation is known to be NP-hard [5].

Existing methods for the determination of good variable orderings can be classified into three categories. The first are initial heuristics starting from a circuit [13], the second are gradual improvement heuristics based on the exchange of variables in the BDD [15, 14, 21, 19, 20], and the third are exact methods to find an optimal ordering [12, 15, 16]. Obviously, it is desirable to determine the exact result. In applications like the ones discussed above it is even more important to find the *best* variable ordering since it turned out by experiments that the best greedy approaches are up to a factor of two worse than the optimal result. But exact methods were so far only applicable to functions with less than 20 variables, due to their exponential runtime behaviour. Since for larger functions no exact method can be applied, in the past few years several methods have been proposed that are time consuming, but get high quality results, e.g. based on simulated annealing [18, 4] and evolutionary algorithms [10]. The drawbacks of these methods were that they had very bad runtime behaviour or/and were only applicable to small functions. Additionally, these probabilistic approaches are sensitive to several chosen parameters and for this are not very robust.

In this paper we present an approach to exact BDD minimization based on the classical algorithm from [12] with branch&bound technique [15, 16]. The key part of our algorithm is a lower bound technique used in [8] for (theoretical) lower bound proofs. This technique has been improved. It is used for the first time in a practical application, i.e. a theoretical method has been implemented as an automated tool. Moreover, we tuned the underlying data structures with respect to execution time and memory requirement. Our algorithm works well for multi-output functions and makes use of symmetries (see also [16, 22]). By this, we were able to find the optimal variable ordering for all benchmarks from LGSynth93 with less than 24 variables. Furthermore, for partially symmetric functions, like adders, we were able to determine the exact result up to 64 variables. For those functions all previously presented approaches fail.

The paper is structured as follows: First, we introduce our notations and definitions. We then review earlier approaches on exact minimization of BDDs in Section 3 and explain the differences to our work. The lower bound that is used in our algorithm is given in Section 4 where also a sketch of the algorithm and some technical details are illustrated. In Section 5 we show our experiments. Finally, the work is summarized.

2 Preliminaries

Boolean variables can assume values from $\mathbf{B} := \{0, 1\}$ and are denoted by Latin letters. Indices from a given index set $\{1, .., n\}$ can be used for the variables, e.g. $x_1, .., x_n$. In the following we consider Boolean functions $f : \mathbf{B}^n \to \mathbf{B}^m$ over the variable set $X_n = \{x_1, \ldots, x_n\}$. A subset of variables is denoted by I ($I \subseteq X_n$).

As well-known, each Boolean function $f : \mathbf{B}^n \to \mathbf{B}$ can be represented by a *Binary Decision Diagram* (BDD) [7],

35th Design Automation Conference ®
Copyright ©1998 ACM
1-58113-049-x-98/0006/$3.50   DAC98 - 06/98 San Francisco, CA USA

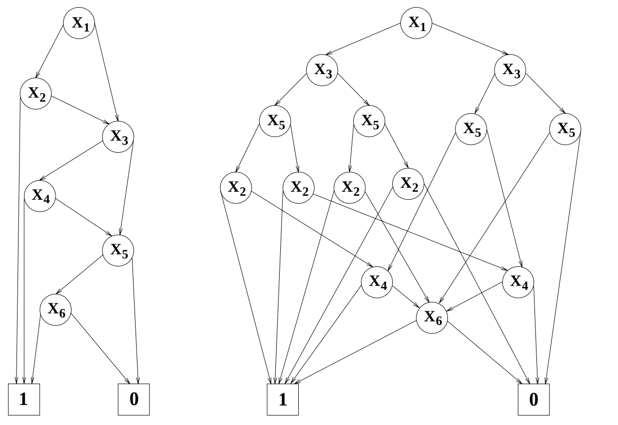

Figure 1: BDDs of the function $f = x_1x_2 + x_3x_4 + x_5x_6$

i.e. a directed acyclic graph where a Shannon decomposition

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

is carried out in each node.

A BDD is called *ordered* if each variable is encountered at most once on each path from the root to a terminal node and if the variables are encountered in the same order on all such paths. A BDD is called *reduced* if it does not contain vertices either with isomorphic sub-graphs or with both edges pointing to the same node. Reduced and ordered BDDs are canonical, i.e. for each Boolean function the BDD can be uniquely determined.

BDDs are defined analogously for multi-output functions $f : \mathbf{B}^n \to \mathbf{B}^m$ as for the case of single-output functions: A BDD G_j for each component function f_j $(1 \leq j \leq m)$ is used for the *shared* BDD representation G for f. The variable ordering is fixed for all G_js.

For functions represented by reduced, ordered BDDs efficient manipulations are possible [7]. In the following only reduced, ordered BDDs are considered and for briefness these graphs are called BDDs.

It has been shown in [5] that improving the variable ordering of BDDs is NP-complete. However, the BDD's size largely depends on the variable ordering, i.e. it may vary from linear to exponential.

Example 1 Let $f = x_1x_2 + \ldots + x_{2n-1}x_{2n}$. If the variable ordering is given by $(x_1, x_2, \ldots, x_{2n})$ the size of the resulting BDD is $2n$. On the other hand if the variable ordering is chosen as $(x_1, x_{n+1}, x_2, x_{n+2}, \ldots, x_{2n})$ the size of the BDD is $\Theta(2^{n-1})$. Thus, the number of nodes in the graph varies from linear to exponential depending on the variable ordering. In Figure 1 the BDDs of the function $f = x_1x_2 + x_3x_4 + x_5x_6$ with variable orderings $(x_1, x_2, x_3, x_4, x_5, x_6)$ and $(x_1, x_3, x_5, x_2, x_4, x_6)$ are illustrated. The left (right) outgoing edge of each node marked by x_i denotes $f_{x_i=1}$ $(f_{x_i=0})$. As can be seen the choice of the variable ordering largely influences the size of the BDDs.

In the following we make use of *Complement Edges* (CEs) [6] without mentioning it further. (Note that all results directly transfer to BDDs without CEs.)

In this paper we denote a permutation of variables with π. If $x_i = \pi(k)$ for a variable x_i, then x_i is the kth element of the variable ordering π, i.e. x_i is in the kth *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.

The number of nodes of a BDD for function f and permutation π is denoted by $\#nodes(f, \pi)$. The number of nodes in level k marked with variable x_i under permutation π is given by $\#nodes_{x_i}(f, \pi)$.

We now consider the following problem:

How can we determine an optimal variable ordering for a BDD representing a given Boolean function f, such that the BDD's size is a global minimum?

3 Previous Work

To make the paper self-contained we give an overview on existing exact minimization algorithms for BDDs. Since improving the variable ordering is NP-complete, all exact minimization algorithms presented have exponential runtime behaviour.

The trivial idea is to construct the BDDs for all $n!$ variable orderings. The optimal variable ordering is the one leading to the smallest BDD. This approach is only applicable to very "small" functions with up to 10 variables.

In [12] an exact algorithm has been presented where the number of variable orderings which have to be considered could be reduced significantly. The main idea behind that algorithm is the following lemma:

Lemma 1 Let $f : \mathbf{B}^n \to \mathbf{B}^m$, $I \subseteq X_n$, $k = |I|$, and $x_i \in I$. Then there exists a constant c such that $\#nodes_{x_i}(f, \pi) = c$ for each $\pi \in \Pi(I)$ with $\pi(k) = x_i$.

More informally the lemma states that the number of nodes in a level is constant, if the corresponding variable is fixed in the ordering and no variables from the lower and upper part are exchanged. Notice, that this holds independently of the ordering of the variables in the upper and lower part of the BDD.

The lemma can be used as follows: for $I \subseteq X_n$, let

$$min\_cost_I = \min_{\pi \in \Pi(I)} \sum_{i=1}^{|I|} \#nodes_{x_i}(f, \pi),$$

and let π_I denote some permutation leading to that minimum. Assume for a fixed $I \subseteq X_n$ with $|I| = k$ that we know $min\_cost_{I'}$ for all $I' \subseteq I$ with $|I'| = k - 1$. Then we get

$$min\_cost_I = \min_{x_i \in I}(min\_cost_{I \setminus \{x_i\}} + \#nodes_{x_i}),$$

with $\#nodes_{x_i}$ being the number of nodes in level $n - k + 1$ under some permutation π with $\pi(I \setminus \{x_i\}) = I \setminus \{x_i\}$ and $\pi(n - k + 1) = x_i$. So the optimal ordering can be computed iteratively by computing for increasing k's min_cost_I for each k-element subset I, until $k = n$. The representation in [12] is done by extended truth tables.

Remark 1 For increasing k the considered level $n - k + 1$ moves up. By this the BDD is build bottom up. In fact, the BDD can also be built top down. As $\Pi(I) = \Pi(X_n \setminus I)$, Lemma 1 can be used the same way in that case. This is an essential aspect of our algorithm, because of the calculation of the lower bound (see next section). Notice that all exact algorithms presented so far use a bottom up construction.

```
1   compute_optimal_ordering ($f(x_1, \cdots, x_n)$) {
2      $\pi_\emptyset$ = an initial ordering;
3      insert $I = \emptyset$, $min\_cost_\emptyset$=0, $\pi_\emptyset$ into $table$;
4      for($k = 1; k \leq n; k++$) {
5         $k$: level under consideration;
6         $next\_table$ = $\emptyset$;
7         for(each $I \in table$) {
8            set_permutation ($\pi_I$);
9            for(each $x_i$ in $X_n \setminus I$) {
10              if ($x_i$ is not top of ((symmetry group of $x_i$) $\cap$ ($X_n \setminus I$)) )
11                 goto 9 and consider next $x_i$;
12              shift $x_i$ to level $k$;
13              $\pi$: current ordering;
14              $I' = I \cup \{x_i\}$;
15              $cost = min\_cost_I + \#nodes_{x_i}(f, \pi)$;
16              if ($I' \notin next\_table$ or $cost < min\_cost_{I'}$) {
17                 store $I'$, $cost$, $\pi$ into $next\_table$;
18                 $upper\_bound$ = update_upper_bound();
19                 $lower\_bounds_{I'}$ = compute_lower_bound($x_i$);
20              }
21              if ($\#nodes(f, \pi) > 1.5 \cdot$ (size before shifting $x_i$))
22                 undo shifting of $x_i$;
23           }
24        }
25        clear all elements in $next\_table$ with $lower\_bounds_{I'} \geq upper\_bound$;
26        $table$ = $next\_table$;
27     }
28     set_permutation(ordering of $upper\_bound$);
29  }
```

Figure 2: Sketch of the algorithm

An extension of [12] has been presented in [15]. This approach makes use of BDDs instead of truth tables. Additionally, the search space is reduced by a branch&bound strategy, where parts of the search tree are pruned.

The most efficient algorithm presented so far is an extension of [15] described in [16]. There the search space is reduced if the function has symmetric variables and a better lower bound is used. Additionally, the exchange of variables is performed efficiently and the algorithm works well for multi-output functions. The algorithm also uses hash tables as the underlying data structure. For non-symmetric functions, execution times are speeded up by a factor of two in comparison to [15], for symmetric functions even more. In the following only comparisons to [16] are given, since this is the fastest and most memory effective algorithm presented so far.

4 Exact Minimization

In this section we describe the flow of our exact algorithm. In principle, the branch&bound mechanism from [16] is used. The gain of our algorithm results from the following:

- The lower bound which is used to prune the search space is derived from a theoretical technique in circuit complexity (see [8]). Our method obtains much better lower bounds and by this speeds up the algorithm tremendously. As a consequence of our lower bound calculation, the BDD *has to* be build top down instead of bottom up (see Remark 1). Details are given in Section 4.2.
- The BDD's size may become exponential while processing one subset $I \subseteq X_n$. This growth is reduced by undoing the sifting of some specified variables.
- The size of the hash table is dynamically adapted. Thus, we are able to reduce the memory used during computation.

4.1 Algorithm

A sketch of the exact algorithm is given in Figure 2. First, the BDD for a given Boolean function is constructed using sifting operations during symbolic simulation. The present BDD size is used as an initial upper bound for the exact minimization procedure[1].

For each subset $I \subseteq X_n$ the best variable ordering π_I and the corresponding BDD size min_cost_I are stored in hash table *table*. Lines 2 and 3 of Figure 2 show the initialization phase of *table*. In line 4 the iteration over all levels in the BDD starts. Then each $I \subseteq X_n$ which has an entry in *table* is considered (line 7). (I has no entry in *table*, if the lower bound was larger or equal than the upper bound in previous steps.) The remaining variables marked by x_i which are not in I are then shifted to level k. Notice, that each symmetry group is considered only once (see also lines 10/11). The cost of the current ordering is computed in line 15. Lines 16 to 20 describe the computation of *min_cost* for the current subset I' and then these subsets are stored in *next_table*. Obviously, the new upper bound is given by the current

[1]The algorithm can often be further speeded up, if the upper bound is determined by a preprocessing based on evolutionary algorithm principles [11].

minimal BDD size. The lower bounds for all I's (line 19) are determined as given in Section 4.2. To save time and memory during computation, in line 21 and 22 the shifting of variable x_i is reset if the size of the BDD grew too much by the last shifting. If the lower bound for subset I' is larger or equal than the current upper bound, I' need not to be considered anymore, i.e. it is deleted from *table* (line 25).

In the last iteration ($k = n$), $\mathit{min\_cost}_{X_n}$ is computed, which is the minimal BDD size over all variable orderings.

4.2 Lower Bound Technique

In this section we describe our lower bound technique that is the key part of our algorithm. In [8] lower bounds for the size of BDDs have been proven using lower bound techniques from VLSI design. First, it is shown how to adapt this theoretical technique.

The argument of [8] is shortly reviewed giving a lower bound on the BDD's size for $f : \mathbf{B}^n \to \mathbf{B}$. (Notice that the same argumentation holds for multi-output functions.) Let (L, R) be a partition of X_n. Then, any input assignment $x : X_n \to \mathbf{B}$ can be split into a *left input assignment* $l : L \to \mathbf{B}$ and a *right input assignment* $r : R \to \mathbf{B}$. We use the notation $x = lr$. A set $\mathbf{F}_{(L,R)}$ of left input assignments is called a *fooling set* for the partition (L, R) iff for each two distinct $l, l' \in L$ there is a right input assignment r with $f(lr) \neq f(l'r)$. Fixing a real parameter $\omega \in (0, 1)$ and a subset $Y \subseteq X_n$ a partition (L, R) is called a *balanced partition* iff

$$\lfloor \omega \cdot |Y| \rfloor \leq |Y \cap L| \leq \lceil \omega \cdot |Y| \rceil.$$

The main tool in [8] for proving lower bounds on the size of BDDs is:

Lemma 2 If, for each balanced partition (L, R) (with respect to some ω and Y), f has a fooling set $\mathbf{F}_{(L,R)}$ of size c, then the BDD for f has size at least c.

Following [8] the proof of this lemma is based on

Lemma 3 If, for one partition (L, R), f has a fooling set $\mathbf{F}_{(L,R)}$ of size c, then for any variable ordering π with

$$L = \{\pi(i) : 1 \leq i \leq |L|\},$$

$$R = \{\pi(i) : |L| < i \leq n\},$$

the BDD with variable ordering π has at least c nodes in levels $|L| + 1, \ldots, n$.

For our application this can be used as follows: For some $I \subseteq X_n$, the size c of the fooling set $\mathbf{F}_{(I, X_n \setminus I)}$ equals the number of nodes in levels $|I| + 1, \ldots, n$ referenced directly from the nodes in levels $1, \ldots, |I|$.

When computing the lower bound for a multi-output function $f : \mathbf{B}^n \to \mathbf{B}^m$, we already know the minimum number $\mathit{min\_cost}_I$ of nodes in levels $1, \ldots, |I|$. So the lower bound can be computed as

$$\mathit{lower\_bound} = \mathit{min\_cost}_I + \max\{c + m_R, n - |I|\} + 1,$$

where m_R is the number of output nodes in levels $|I| + 1, \ldots, n$ and $n - |I|$ is the number of variables in $X_n \setminus I$, since there will be at least one node for each variable in $X_n \setminus I$. The constant node is always needed.

The computation of $c + m_R$ is done by traversing the levels $|I| + 1$ to n of the BDD, counting for each node of these levels the number of references from a visited node.

```
compute_lower_bound(level: k) {
   for($i = k + 1; i \leq n; i + +$)
      for(any node $v$ in level $i$)
         copy reference count of $v$;
   $lb1 = 0$;
   for($i = k + 1; i \leq n; i + +$)
      for(any node $v$ in level $i$) {
         if (references($v$)$\geq$ 1)
            $lb1 + +$;
         decrement references of successors;
      }
   $lb2 = n - k$;
   return ($\mathit{min\_cost}_I + \mathit{max}(lb1, lb2) + 1$);
}
```

Figure 3: Computation of lower bound

The number $c + m_R$ then is the number of nodes for which the counted number of references and the actual reference count differ[2].

A sketch of the algorithm is given in Figure 3.

4.3 Data Structure

The use of an efficient data structure largely influences the performance of our algorithm. Therefore the most important details are outlined.

For each set $I \subseteq X_n$ the best variable ordering π_I and the corresponding BDD size $\mathit{min\_cost}_I$ are stored in a hash table. I is used as a key and table size s is always chosen prime. We use double hashing, with first hash function $h_1(k) = (k \bmod s)$ and second hash function $h_2(k) = 1 + (k \bmod (s - 2))$. Every time the number of elements in the hash table exceeds $0.7 \cdot s$, the table size is increased to the next prime $\geq 2 \cdot s$.

The data structure also considers symmetry aspects of the function to be minimized (see also [16, 22, 23]). Each symmetry group is stored as a circular list where each variable points to the next variable in the same symmetry group. Symmetry groups are determined by symmetric sifting [19, 20]. This also helps to get a better upper bound by an improved initial ordering. To check whether a given variable x_i is the topmost of its symmetry group intersected with a set $S \subseteq X_n$ of variables, we check whether x_i is the one with the smallest index among those in S belonging to x_i's symmetry group. Therefore we only have to follow the circular list from x_i. If a variable x_j ($j \neq i$) occurs with $x_j \in S$ and index less than x_i, then x_i is not the topmost one. If $j = i$, then we have visited all variables of the symmetry group, and x_i was the topmost variable.

5 Experimental Results

All experimental results have been carried out on a *SUN Ultra 1-140* using an upper memory limit of 300 MByte and a runtime limit of 120.000 CPU seconds. Our algorithm has been integrated in the CUDD package [23]. We compare our algorithm to the original exact algorithm in the CUDD package, called JUNON [16]. By this we guarantee that both algorithms are run in the same system environment. Our algorithm has been implemented as the program *FiïZ*.

[2]In commonly used BDD packages, the actual reference count is stored for each node to detect unused nodes.

name	in	out	opt	S	JUNON		FĩzZ	
					time	space	time	space
adder8	16	8	36	8	7.3s	2M	1.1s	<1M
adder12	24	12	56	12	779s	289M	24.4s	<1M
adder16	32	16	76	16	—	—	190s	2M
adder20	40	20	96	20	—	—	882s	4M
adder24	48	24	116	24	—	—	5498s	15M
adder28	56	28	136	28	—	—	26489s	34M
adder32	64	32	156	32	—	—	120104s	77M
mult5	10	10	388	10	5.1s	<1M	7.3s	<1M
mult6	12	12	1098	12	45.9s	<1M	121s	<1M
mult7	14	14	3082	14	493s	<1M	2119s	<1M
mult8	16	16	8658	16	4992s	2M	59219s	3M

Table 1: Arithmetical circuits

In a first series of experiments we consider arithmetic functions, i.e. adders and multipliers (see 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 that are used for the minimum representation. In column S the number of symmetry sets is given. E.g., a 1 in this column denotes that the function is totally symmetric, and if the number is equal to the number of inputs then the function has no symmetric variables. In columns time and space the runtime in CPU seconds and the space requirement in MByte for JUNON [16] and our approach are given, respectively. For the multipliers JUNON obtains smaller runtimes, since in this case lower bound techniques do not work well, due to exponential size of the multiplier under all orderings [8], and furthermore the computation of the lower bound is more time consuming in our algorithm. On the other hand for the adder function our algorithm clearly outperforms the approach from [16]. The argument of the lower bound works well for adder functions, since many orderings have exponential size. Thus the construction of BDDs with "worse" orderings can be cancelled in an early construction phase. We can minimize the adder function with **32** inputs in less than 200 CPU seconds, while JUNON fails, since it needs too much memory. Due to the efficient data structure described above, FĩzZ can handle this example within 2 MByte of main memory. The limiting resource for FĩzZ with respect to adder functions is the execution time as can be seen in example *adder32* whose runtime is almost in the given range. Nevertheless, FĩzZ is able to compute minimum BDDs for adder functions with up to **64** inputs.

In a second series of experiments we applied our algorithm to a larger set of benchmark circuits from LGSynth93. The results are given in Table 2. As can easily be seen our algorithm is much faster in most cases (see e.g. *pcle*) and also uses much less memory (see e.g. *s400*). It clearly turns out that our lower bound technique is more powerful. Even though the lower bound computation is more expensive in one step, the overall performance is much better since large areas of the search space do not have to be considered. Especially for larger functions with more than 20 variables often JUNON cannot determine the exact result within the given time bound. In contrast our algorithm can handle these difficult problem instances due to its lower bound technique.

Based on our algorithm for several circuits the exact BDD sizes could now be determined, while only heuristical minimization results have been known before.

6 Conclusions

We presented a new exact algorithm for determining the optimal variable ordering for BDDs. It is based on a lower bound technique known from VLSI design. Furthermore, we described an efficient data structure. Experimental results have been reported that clearly demonstrate the efficiency of the approach. A comparison to the best algorithm known so far shows that runtime can often be speed up by a factor of 400 and additionally, the memory requirements can be reduced. Furthermore, we were able to compute exact results for problem instances where all previously published methods fail.

References

[1] P. Ashar, S. Devadas, and K. Keutzer. Path-delay-fault testability properties of multiplexor-based networks. *Integration the VLSI Jour.*, 15(1):1–23, 1993.

[2] B. Becker. *Synthesis for Testability: Binary Decision Diagrams*, volume 577 of *LNCS*. Symp. on Theoretical Aspects of Comp. Science, 1992.

[3] V. Bertacco, S. Minato, P. Verplaetse, L. Benini, and G. De Micheli. Decision diagrams and pass transistor logic synthesis. In *Int'l Workshop on Logic Synth.*, 1997.

[4] B. Bollig, M. Löbbing, and I. Wegener. Simulated annealing to improve variable orderings for OBDDs. In *Int'l Workshop on Logic Synth.*, pages 5b:5.1–5.10, 1995.

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

[6] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient implementation of a BDD package. In *Design Automation Conf.*, pages 40–45, 1990.

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

[8] R.E. 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.

name	in	out	opt	S	JUNON		FỉzZ	
					time	space	time	space
cc	21	20	46	21	—	34M	689s	32M
cm150a	21	1	33	21	315s	34M	3607s	32M
cm163a	16	5	26	13	59s	<1M	7.3s	<1M
cmb	16	4	28	3	0.2s	<1M	0.2s	<1M
comp	32	3	95	16	—	—	48631s	99M
cordic	23	2	42	11	152s	140M	23.8s	<1M
cps	24	102	971	21	—	289M	45809s	48M
i1	25	16	36	16	—	—	186s	9M
lal	26	19	67	21	—	—	4415s	64M
mux	21	1	33	21	318s	34M	3619s	32M
parity	16	1	17	1	<0.1s	<1M	<0.1s	<1M
pcle	19	9	42	18	23275s	9M	57.1s	2M
pm1	16	13	40	10	14.3s	<1M	3.5s	<1M
s208.1	18	9	41	18	6822s	5M	56.5s	<1M
s298	17	20	74	17	3427s	2M	88.5s	2M
s344	24	26	104	24	—	289M	10820s	99M
s349	24	26	104	24	—	289M	10746s	99M
s382	24	27	119	24	—	289M	6554s	67M
s400	24	27	119	24	—	289M	6568s	67M
s444	24	27	119	24	—	289M	6709s	74M
s526	24	27	113	24	—	289M	8443s	86M
s820	23	24	220	21	—	140M	17320s	53M
s832	23	24	220	21	—	140M	17854s	53M
sct	19	15	48	18	26210s	9M	58.9s	2M
t481	16	1	21	8	2.4s	<1M	1.2s	<1M
tcon	17	16	25	17	2159s	2M	3.3s	<1M
ttt2	24	21	107	24	—	289M	6727s	74M
vda	17	39	478	17	4177s	2M	633s	2M

Table 2: Comparison of exact algorithms

[9] P. Buch, A. Narayan, A.R. Newton, and A.L. Sangiovanni-Vincentelli. On synthesizing pass transistor networks. In *Int'l Workshop on Logic Synth.*, 1997.

[10] R. Drechsler, B. Becker, and N. Göckel. A genetic algorithm for variable ordering of OBDDs. *IEE Proceedings*, 143(6):364–368, 1996.

[11] R. Drechsler and N. Göckel. Minimization of BDDs by evolutionary algorithms. In *Int'l Workshop on Logic Synth.*, 1997.

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

[13] 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.

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

[15] 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.

[16] 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.

[17] L. Lavagno, P. McGeer, A. Saldanha, and A.L. Sangiovanni-Vincentelli. Timed shannon circuits: A power-efficient design style and synthesis tool. In *Design Automation Conf.*, pages 254–260, 1995.

[18] M.R. Mercer, R. Kapur, and D.E. Ross. Functional approaches to generating orderings for efficient symbolic representations. In *Design Automation Conf.*, pages 624–627, 1992.

[19] D. Möller, P. Molitor, and R. Drechsler. Symmetry based variable ordering for ROBDDs. *IFIP Workshop on Logic and Architecture Synthesis, Grenoble*, pages 47–53, 1994.

[20] S. Panda, F. Somenzi, and B.F. Plessier. Symmetry detection and dynamic variable ordering of decision diagrams. In *Int'l Conf. on CAD*, pages 628–631, 1994.

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

[22] D. Sieling. Variable orderings and the size of OBDDs for partially symmetric Boolean functions. In *SASIMI*, pages 189–196, 1996.

[23] F. Somenzi. *CUDD: CU Decision Diagram Package Release 2.1.2.* University of Colorado at Boulder, 1997.

[24] K. Yano, Y. Sasaki, K. Rikino, and K. Seki. Top-down pass-transistor logic design. *IEEE Jour. of Solid-State Circ.*, 31(6), June 1996.