REDUCING THE NUMBER OF VARIABLE MOVEMENTS IN EXACT BDD MINIMIZATION

Rüdiger Ebendt

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

ABSTRACT

Ordered Binary Decision Diagrams (BDDs) are frequently used in logic synthesis.

In this paper a new exact BDD minimization algorithm is presented, which is based on state space search. In contrast to all previous approaches, in which variables are moved through the BDD when exploring the state space, the new method makes use of a new technique to expand states to its successor states without expensive variable movements. Experimental results are given to show the efficiency of the approach.

1. INTRODUCTION

Recently, the interest in logic synthesis based on FPGAs and Pass Transistor Logic (PTL) has been renewed. These techniques and the resulting circuits have several desirable properties, amongst them improved power dissipation, circuit speed and area as well as high design quality by consideration of layout aspects during synthesis [15, 3, 7, 14, 13, 5]. These techniques make use of ordered *Binary Decision Diagrams* (BDDs) as introduced in [2]. Their size often critically depends on the chosen variable ordering.

In applications like logic synthesis it is most important to determine a good ordering, since a reduction in the number of BDD nodes directly transfers to a smaller chip area.

Heuristical approaches [9, 16] cannot guarantee an optimal result and often yield BDDs up to twice the size of the best known solution. Since this is a serious problem, exact algorithms have been proposed [8, 11, 12, 4, 6]. Due to the NP-completeness of the problem of improving a given variable ordering [1], all suggested approaches so far have longer runtimes compared to heuristics. A main reason for the long runtimes of all known approaches is that they require many time-consuming movements of variables through the BDD. Hence, there is a strong demand for faster solutions.

In this paper a new exact algorithm for the computation of an optimal variable ordering is presented. As all other exact algorithms presented so far, the core technique is based on [8] in combination with branch&bound. But in contrast to previous approaches, a new state expansion technique without expensive movement of variables through the BDD is used. Experiments show that significant runtime reductions can be observed on benchmark functions, i.e. for the functions considered an improvement of up to 63.9% 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 acyclic graph where a Shannon decomposition

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

into two cofactors in x_i is carried out in each node, yielding a "then-successor" via a 1-edge and an "else-successor" via a 0-edge (as an example for a BDD see Figure 2, the annotated variable sets for now can be ignored and are explained later in Section 4.1). 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", e.g. $x_1 < x_2 < x_3 < x_4$ in Figure 2) on every path from the root to a terminal node. For more details see [2]. Variable orderings are often denoted by permutations π. For simplicity, the author uses the notation $x_i = \pi(k)$, 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. Permutations π are extended straightforwardly to also map subsets of X_n to subsets of X_n. Further, the minimal number of nodes in a BDD labeled by a variable in $I \subseteq X_n$ is denoted with $\mathit{min_cost}_I$. BDDs are defined analogously for multi-output functions $f\colon \mathbf{B}^n \to \mathbf{B}^m$, using a graph for each of the m single-output functions $(f_i^n)_{1 \le i \le m}$ for the *shared* BDD representation. In the following shared BDDs with *Complement Edges* (CEs) are assumed without mentioning it further. (Note that all results reported here directly transfer to BDDs without CEs.) For a BDD F, let $\mathrm{label}(F, x_k)$ denote the number of nodes in F labeled by x_k.

3. PREVIOUS WORK

To keep the paper self-contained, the fastest approaches to exact BDD minimization known so far [4, 6], which are based on the framework of [11], are briefly reviewed.

Suppose the BDD for a multi-output function $f\colon \mathbf{B}^n \to \mathbf{B}^m$ is minimized. In brief, the optimal variable ordering is computed iteratively by computing for increasing k's $\mathit{min_cost}_I$ for each k-element subset I of X_n, until $k = n$: then, the BDD has a variable ordering yielding a BDD size of $\mathit{min_cost}_{X_n}$. 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 $x_i \in X_n \setminus I$. The subject is to compute *min_cost*$_{I'}$ for each successor I'. This is done by a gradual scheme of continuous minimum updates following [8], which computes *min_cost*$_{I'}$ by use of the minimum values for the predecessors of I'. These values have been computed in the last step $k-1$ and are simply retrieved from a hash table. Therefore, following [4, 6], the only terms needed to compute for a state I, which is going to be expanded in step k are $\mathrm{label}(F_i, x_i)$ for each $x_i \in X_n \setminus I$, where F_i is a BDD representing f with variable ordering π_i such that $\pi_i(I) = I$ and $\pi_i(|I|+1) = x_i$.

At the end of step k, all states for which a lower bound on the BDD size achievable from that state exceeds or equals the current known upper bound on the minimal BDD size, are excluded. In [4] an effective lower bound known from VLSI design was used and in [6], three lower bounds were used in parallel, further enhancing this approach. For more details on exact BDD minimization with branch&bound see [11, 4, 6].

4. ALGORITHM

In this section the state expansion technique used in the new approach is described. The implementation of the algorithm is based on the implementation of recent approaches [4, 6]. Hence the new method uses techniques, that have been applied there. Amongst them are the use of symmetries, customized hash tables and the efficient technique for fast BDD reconstruction of [6]. The presence of all these techniques is assumed without further mentioning. A sketch of the new algorithm called NEO is given in Figure 1. With "hash()" the use of the double hash function as suggested in [4] is indicated. It is applied both to variable sets, implemented as bitmasks of n bits which are interpreted as longword size integers and to the addresses of BDD nodes, interpreted as word size integers.

4.1. Theoretical Background and Idea

Reconsidering the exact minimization algorithm described in section 3, it can be observed: In step k, the algorithm expands all states I with $|I| = k-1$ to all possible successor states $I' = I \cup \{x_i\}$ for $x_i \in X_n \setminus I$. The terms $\mathrm{label}(F_i, x_i)$ (where F_i is a BDD representing f with variable ordering π_i such that $\pi_i(I) = I$ and $\pi_i(|I'|) = x_i$) now must be computed. In previous approaches this was done by actually constructing the BDDs F_i requiring (at least) the movement of the variables x_i through the BDD to the $(|I|+1)$-th level. This is a very time consuming operation with the potential risk of increasing the number of BDD nodes ("BDD explosion"). The new approach applies a different technique for this, using an argument, which follows [8]:

Let $I \subseteq X_n$ and $x_i \in X_n \setminus I$. Let F be a BDD with a variable ordering π such that $\pi(I) = I$ and $\pi(|I|+1) = x_i$. Assume, F is representing a multi-output function $f\colon \mathbf{B}^n \to \mathbf{B}^m$ which can be written as family $({f_i}^n)_{1\le i\le m}$. Let $\mathrm{dep}(f, I, x_i)$ denote the set of those cofactors of the functions $({f_i}^n)_{1\le i\le m}$ in all variables in I, which are functions depending essentially on x_i.

Lemma 4.1 The number of nodes in the $(|I|+1)$-th level of F is equal to $|\mathrm{dep}(f, I, x_i)|$.

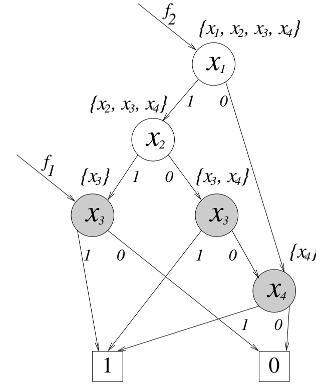

Figure 2: BDD for Example 4.1.

Example 4.1 Consider the BDD given in Figure 2. The cofactors in the variables in $I = \{x_1, x_2\}$ are represented by the grey nodes. Annotated are the sets of variables the cofactors essentially depend on. There are two cofactors depending essentially on x_3 and in fact two nodes residing on the third level for the given ordering $x_1 < x_2 < x_3 < x_4$. With the annotated sets it can be seen (*without* actually moving a variable), that there would also be two nodes labeled x_4 in the third level for the ordering $x_1 < x_2 < x_4 < x_3$, since again two cofactors essentially depend on x_4.

This argument is used in step k (expanding a state I with $|I| = k-1$) of Algorithm NEO in Figure 1 as follows: Let us assume a BDD F for I with an ordering π respecting $\pi(I) = I$. Computing the terms $\mathrm{label}(F_i, x_i)$ simplifies to counting the cofactors which depend essentially on x_i. This can be done with a procedure traversing the nodes in the levels $|I|+1, \ldots, n$, which does not involve movements of variables through the BDD. Hence, no expensive graph reconstruction operations are needed anymore.

4.2. Implementation

Assume, the routine compute_dependencies sketched in Figure 3 is called for a BDD F representing f with variable ordering π such that $\pi(I) = I$ and *level* $= |I|$. First, for every node v in the levels *level* $+1, \ldots, n$ a bit mask is computed, where the bit corresponding to the index of a variable is set iff the function represented by v essentially depends on this variable. The mask will represent the set of variables, the function represented by this node essentially depends on like annotated in Figure 2. This is done bottom up first considering the index of the variable, the node is labeled with (since the function represented clearly depends on that variable). Then, all bits set in the mask of the then-successor of v *or* in the mask of the else-successor of v are set in the mask of v (a bitwise "or"-operation). The annotated sets in Figure 2 demonstrate the idea of this construction.

Afterwards a counter for every variable $x_i \in X_n$ is increased, iff a cofactor in all variables in $I = \{x_{\pi(1)}, \ldots, x_{\pi(\mathit{level})}\}$ essentially depends on x_i. This can now be tested by inspecting the bit masks of the nodes representing these cofactors (the set of these nodes is denoted $\mathrm{cof_nodes}(F, \mathit{level})$ in line (16) in Figure 3). Hence, in this array of counters, *all* terms $\mathrm{label}(F_i, x_i) = |\mathrm{dep}(f, I, x_i)|$ for $x_i \in X_n \setminus I$ are computed during *only one* call to this routine *without* the computational cost of actually constructing each BDD F_i, i.e. without variable movements and the risk of the BDD size increasing.

```
(1)  compute_optimal_ordering(BDD F, int n) { /* F represents f: B^n → B^m */
(2)      minguess[hash(∅)] := 0; states[hash(∅)] := ∅;
(3)      π[hash(∅)] := an arbitrary initial order; upper_bound := update_upper_bound();
(4)      clear next_states;
(5)      for k := 1 to n do
(6)          for each I ∈ states do    /* update minguess[hash(I)] until it reaches min_cost_I */
(7)              reconstruct F according to the first |I| positions of ordering π[hash(I)];
(8)              upper_bound := update_upper_bound();
(9)              compute_dependencies(F, |I|);
(10)             for each x_i ∈ X_n \ I do
(11)                 I' := I ∪ {x_i}; if I' ∉ next_states then minguess[hash(I')] := ∞;
(12)                 newcost := |dep(f, I, x_i)| + minguess[hash(I)];
(13)                 if I' ∉ next_states or newcost < minguess[hash(I')] then
(14)                     minguess[hash(I')] := newcost;
(15)                     π[hash(I')] := the ordering as if x_i had moved to level |I| + 1;
(16)                     if I' ∉ next_states then
(17)                         states[hash(I')] := I';
(18)                         move x_i to level |I| + 1;    /* Note, that k = |I| + 1 */
(19)                         upper_bound := update_upper_bound();
(20)                         lower_bound[hash(I')] := compute_lower_bound(I');
(21)                     end–if
(22)                 end–if
(23)             end–do
(24)             exclude all states I' in next_states with lower_bound[hash(I')] ≥ upper_bound;
(25)             states := next_states;
(26)         end–do
(27)         reconstruct the ordering of upper_bound;
(28)     end–do
(29) }
```

Figure 1: The new algorithm NEO (sketch)

4.3. Revisiting States without Variable Movements

With the new technique, *all but one* variable movements needed to expand all predecessor states I to a particular state I' become obsolete: The only variable movement which is still left to perform *only once* the first time a successor state I' is encountered is the one in line (18) in Algorithm NEO in Figure 1, triggered by the conditional statement in line (16). (This movement prepares the computation of the lower bound. For more details about the computation of the lower bound see [4].) Without the new expansion technique, much more variable movements would be necessary: A successor state I' with $|I'| = k$ is being visited up to k times in the progress of one step of the minimization algorithm, since k distinct states I, J with $|I| = |J| = k-1$ have the successor I' in common. With the new expansion technique, revisiting I' does *not* involve any further variable movements.

5. EXPERIMENTAL RESULTS

All experimental results have been carried out on a system with an Athlon processor running at 1.4 GigaHz using an upper memory limit of 300 MByte and a runtime limit of 20.000 CPU seconds. The new algorithm is called NEO and is compared to the best approaches to exact BDD minimization known so far, called FizZ [4] and the top-down version of JANUS [6]. All algorithms have been implemented with the CUDD package [17] and were tested in the same system environment.

In a series of experiments all algorithms were applied to the set of benchmark circuits from LGSynth93. The results are given in Table 1. In the first column the name of the function is given. Column n denotes the number of inputs of a function. Column *opt* shows the number of BDD nodes needed for the minimal representation. In the columns *time* the runtimes in CPU seconds for FizZ, JANUS and the new approach NEO are given. The last column *space* shows the space requirement of the new approach NEO in MByte. Due to space limitation, this paper is focussed on the comparison of runtimes. The author remarks, that the space requirement of NEO is almost exactly the same as for JANUS given in [6].

As the results show, the new approach NEO is faster than FizZ and JANUS in almost every case. Especially for larger examples a reduction in runtime by up to 63.9% in comparison to FizZ is achieved (see e.g. *mux*, *cm150a*, *cps*). The gain in comparison to JANUS is up to 31.3% (see e.g. *s820*, *s832*). On average, the reduction in runtime is 47.5% in comparison to FizZ. The average gain compared to JANUS is 18.8%. The results show that the new state expansion technique is a very robust improvement, that significantly outperforms the algorithms FizZ and JANUS.

6. CONCLUSIONS

A new exact algorithm for determining an optimal variable ordering for BDDs has been presented. It uses a new technique for state expansion, where time-consuming movements of variables through the BDD are not needed. Experimental results are reported that clearly demonstrate the efficiency of the presented approach. A comparison to the best minimization algorithms known so far shows that runtime can be reduced by up to 63.9%.

```
(1)  compute_dependencies(BDD F, int level) {
(2)    for i := n downto level + 1 do
(3)      for each node v in level i do
(4)        I := 0; /* I is a mask of n bits */
(5)        set the (v.index)-th bit in I;
(6)        if i < n then
(7)          set all bits in I, which are set in
(8)          masks[hash(v.then)] or
(9)          masks[hash(v.else)];
(10)       end–if
(11)       masks[hash(v)] := I;
(12)     end–do
(13)   end–do
(14)   Set all elements of the array label to 0;
(15)   for i := level + 1 to n do
(16)     for each node v in cof_nodes(F, level) do
(17)       for j := 0 to n − 1 do
(18)         I := masks[hash(v)];
(19)         if the j-th bit is set in I then
(20)           label[j] := label[j] + 1;
(21)         end–if
(22)       end–do
(23)     end–do
(24)   end–do
(25) }
```

Figure 3: Computing the dependencies

7. ACKNOWLEDGEMENTS

The author would like to thank R. Drechsler for open-minded discussions and for a helpful introduction to BDD based approaches to logic synthesis. This work benefited much from both of that. Special thanks go to W. Günther, who supported the author with the source code of FizZ. The author also wishes to thank him for the sharing of ideas. It was W. Günther who suggested to use the BDD cache of [10] in the context of exact BDD minimization. This idea was first used in [6] and is used again in the implementation of the approach presented here. Last but not least thanks also go to P. Malik, R. Malik and O. Mayer for motivation and support.

8. 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] 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.

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

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

[6] R. Ebendt, W. Günther, and R. Drechsler. Combination of lower bounds in exact BDD minimization. Accepted for DATE 2003.

[7] 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*, 1998.

Table 1: Comparison of FizZ, JANUS and NEO.

name	n	opt	FizZ time	JANUS time	NEO time	NEO space
cc	21	46	117s	84.9s	69.4s	36M
cm150a	21	33	610s	311.1s	220s	37M
cm163a	16	26	1.17s	0.78s	0.92s	<1M
cmb	16	28	0.01s	0.05s	0.33s	<1M
comp	32	95	5606s	3900s	3520s	121M
cordic	23	42	3.05s	1.82s	1.93s	<1M
cps	24	971	4396s	2751s	1893s	61M
i1	25	36	29.4s	18.77s	22.1s	10M
lal	26	67	677s	504.4s	440s	79M
mux	21	33	610s	310.8s	220s	36M
parity	16	17	<0.01s	0.03s	0.27s	<1M
pcle	19	42	9.02s	5.18s	5.40s	3M
pm1	16	40	0.55s	0.34s	0.62s	<1M
s208.1	18	41	8.44s	5.62s	4.33s	2M
s298	17	74	13.46s	9.06s	7.31s	2M
s344	24	104	1446s	950s	906s	111M
s349	24	104	1447s	950s	905s	111M
s382	24	119	802s	461s	409s	75M
s400	24	119	802s	456s	409s	75M
s444	24	119	779s	508s	473s	75M
s526	24	113	1196s	924s	705s	111M
s820	23	220	2034s	1235s	848s	59M
s832	23	220	2076s	1288s	889s	59M
sct	19	48	8.62s	5.97s	6.36s	3M
t481	16	21	0.16s	0.13s	0.50s	<1M
tcon	17	25	0.52s	0.28s	0.79s	<1M
ttt2	24	107	950s	578s	435s	82M
vda	17	478	65.4s	34.4s	32.3s	3M

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] R. Murgai, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Logic Synthesis for Field-Programmable Cate Arrays. Kluwer Academic Publishers, 1995.

[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.