Combining Ordered Best-First Search with Branch and Bound for Exact BDD Minimization*

Rüdiger Ebendt[1] Wolfgang Günther[2] Rolf Drechsler[1]

[1]Institute of Computer Science
University of Bremen
28359 Bremen, Germany
{ebendt,drechsle}@informatik.uni-bremen.de

[2]CL DAT TDM VM
Infineon Technologies
81730 Munich, Germany
wolfgang.guenther@infineon.com

Abstract— Reduced ordered Binary Decision Diagrams (BDDs) are frequently used in logic synthesis. In this paper we present a new algorithm to determine an optimal variable ordering of BDDs. In this, we combine ordered best-first search, i.e. the A^*-algorithm, with a classical Branch and Bound (B&B) algorithm. A^* operates on a state space, large parts of which are pruned by a best-first strategy expanding only the most promising states. Combining A^* with B&B allows to avoid unnecessary computations and to save memory.

Experimental results demonstrate the efficiency of our new approach.

I. Introduction

Reduced ordered *Binary Decision Diagrams* (BDDs) were introduced in [3] and are well known from techniques for logic synthesis that recently have been presented. These new techniques provide a design style based on multiplexors, which often can be realized at very low cost (e.g. as *Pass Transistor Logic* (PTL)). In addition, these techniques allow to consider layout aspects during the synthesis step and by this guarantee high design quality, e.g. [13].

As is well-known, the size of BDDs is often very sensitive to a chosen variable ordering. In [3] an example has been given, where the BDD size of a function varies from linear to exponential dependent on the ordering of the variables. Since the problem of improving a given ordering has been proven to be an NP-complete problem [1], in the past many heuristic approaches have been proposed that are based on structural information [9] or on dynamic reordering of BDDs [14]. But these methods often yield BDDs up to twice the size of the best known solution. Especially in applications like logic synthesis, this is a significant drawback, since a reduction in the number of BDD nodes directly transfers to a smaller chip area.

For this reason, exact algorithms have been suggested. All approaches presented so far were based on the framework of [8], which was limited to a few variables. Several extensions of this approach have been proposed [11, 12, 5, 7, 6]. However, the new approach presented here uses a search algorithm so far mainly used in *Artificial Intelligence* (AI): the so-called A^*-algorithm. The A^*-algorithm prunes large parts of state spaces and today has become an integral part of standard AI search techniques. Using this successful AI technique, the problem of finding an optimal variable ordering is reduced to the problem of computing an optimal path in a state space. This state space is much smaller than the space of all orderings. It is explored by expanding states to its successors, until a goal state is found. The new method always determines the best, i.e. the most promising state before the next expansion. In this, unlike previous methods which used a static schema to enumerate the orderings to extend, our new approach determines the order of states to expand on the basis of a dynamic evaluation function. With that large parts of the state space are pruned. This pruning effect is higher than in any other approach for exact BDD minimization presented so far. Moreover, an operation frequently needed in exact BDD minimization, the reconstruction of BDDs, is refined and accelerated. The best-first ordering of states further speeds up its runtime significantly.

To prune the search space more effectively, we combine this new approach with already existing *Branch and Bound* (B&B) techniques. This combination of A^* and B&B is an improvement of A^* itself, i.e. it can be transferred to other applications as well. Another improvement of A^* newly presented here addresses the memory requirement of the approach, yielding reductions in memory consumption up to 61.6%.

Experiments show that significant runtime reductions can be observed on benchmark functions, i.e. for the functions considered an improvement up to 58.0% has been obtained.

II. Background

A. BDDs

Boolean variables, denoted by Latin letters, are bound to values in $\mathbf{B} := \{0,1\}$. As is well-known, a Boolean function $f\colon \mathbf{B}^n \to \mathbf{B}$ over the variable set X_n can be represented by a BDD. This is 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. Redundant nodes, i.e. nodes not needed to represent f, can be eliminated. Therefore, BDDs allow a unique (i.e. canonical) and very compact representation of Boolean functions.

In the following, only reduced, ordered BDDs are considered and for briefness these graphs are called BDDs. 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 [3]. A variable ordering often is denoted as a permutation π over X_n, and then extended to a permutation over 2^{X_n}. Let $V \subseteq X_n$. An ordering, whose first $|V|$ members constitute V, can be characterized by the condition $\pi(V) = V$, i.e. V is a fixed point of π. In the following we assume shared BDDs for multi-output functions $f\colon \mathbf{B}^n \to \mathbf{B}^m$ (see Fig. 1) with *Complement Edges* (CEs) [2] without mentioning it further. Note that all results reported here directly transfer to BDDs without CEs.

B. State Space Search by A^*-Algorithm.

A state space problem consists of determining a sequence of operators

$$\xrightarrow{o_1}, \xrightarrow{o_2}, \ldots, \xrightarrow{o_n}$$

that, when applied to the initial state, yields a goal state. An important method to *guide* the search on a state space is *heuristic search*. With every state q a quantity $h(q)$ is associated, which

*This work was supported in part by DFG grant DR 287/8-1.

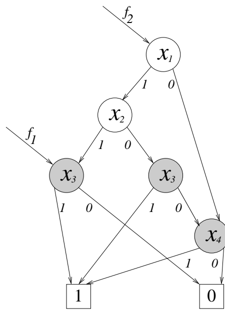

Fig. 1. A shared BDD. Grey nodes are representing cofactors wrt x_1 and x_2.

estimates the cost of the cheapest path from q to a goal state. This allows us to search in the direction of the goal states. The A^*-*algorithm* [10] bases the choice of the next state to expand on two criteria: the cost of the cheapest known path from the initial state up to state q, denoted $g(q)$, and the estimate $h(q)$, which for A^* has to be a lower bound on the cost of an optimal path from q to a goal state (this minimal cost is denoted $h^*(q)$). A^* maintains a prioritized queue OPEN, which is ordered with respect to increasing values $\varphi(q) = g(q) + h(q)$, thus *combining* the two criteria a) cost of the cheapest path to q known so far and b) expected cost of the remaining part of the path from q to a goal node. In the beginning, this queue only contains the initial state. In each step, a state q with a *minimal* φ-value is expanded and dequeued. During expansion, the successor states of q are generated and inserted into the queue according to their φ-values. For this, the values g and h of the successor states are computed dynamically. If q' is a successor of q, q' is associated with its cost $g(q') = g(q) + c(q,q')$, i.e. g accumulates transition costs. In this, for a state q, $g(q)$ is computed as the sum of the cost $c(r,r')$ of all transitions $r \longrightarrow r'$ occurring on the cheapest known path to q.

A successor state q' might be generated a second time, if q' has more than one predecessor state. If a cheaper path from the initial state to q' is found in this case, $g(q')$ is updated. These updates of the "g-part" of φ to the costs of a newly found cheaper path to q continuously compensate for the fact that the character of the "h-part" is only estimative. The cheapest known path to q' is denoted $p(q')$ and is also updated respectively. The algorithm terminates, if the next state to expand is a goal state t. The estimate $h(t) = h^*(t)$ must be zero. In this case, the path found up to t, i.e. $p(t)$, is of minimal cost C^*, which we also express with $\varphi^*(t) = g(t) = g^*(t) = C^*$. This minimum cost path $p(t) = p^*(t)$ is reported as solution.

III. Best-First Search Algorithm for Exact BDD Minimization

Suppose, we want to minimize a BDD representing a Boolean multi-output function $f\colon \mathbf{B}^n \to \mathbf{B}^m$. In our new approach, we use the A^*-algorithm to search for the optimal variable ordering of a BDD. Instead of searching the whole set of variable orderings (which contains $n!$ orderings), the algorithm operates on a state space. This space is 2^{X_n}, the set of variable sets, which is a space of a size growing much slower with n than the size of the set of variable orderings. A state of this space is a set of variables $q \subseteq X_n$. In the following we explain how the problem of finding an optimal variable ordering can be expressed as the problem of finding a minimum cost path from an initial state to a goal state in this state space.

We consider sets of variables q successively growing from $\emptyset$ to X_n: q is extended at each transition by a variable $x_i \in X_n \setminus q$, i.e. $q \xrightarrow{x_i} q \cup \{x_i\}$. The algorithm starts in the initial state $\emptyset$ and progresses until the goal state X_n is reached. As described before in Section B, A^* finds a path $p^*(X_n)$ from $\emptyset$ to X_n with minimal cost. This cost is the accumulated transition cost for transitions $\emptyset \xrightarrow{x_{i_1}} \{x_{i_1}\} \xrightarrow{x_{i_2}} \cdots \xrightarrow{x_{i_n}} X_n$. The sequence of variables occurring on path $p^*(X_n)$ defines a variable ordering.

The basic idea of our approach is the following: We want the above ordering annotated along the minimum cost path to be optimal, i.e. we want the minimal cost for the terminal state X_n (i.e. $\varphi^*(X_n)$) to be the number of nodes in the BDD with this annotated order $x_{i_1} < x_{i_2} < \cdots < x_{i_n}$. Then the sequence of variables in $p^*(X_n)$ is an optimal variable ordering, which yields the minimal BDD size.

For this purpose, an appropriate cost function is chosen: assume a state q always is associated with a BDD the first $|q|$ variables of which are ordered according to the first $|q|$ positions of the variable ordering, which is annotated at the transitions of the currently considered path to q. With that, this BDD respects a variable ordering π with $\pi(q) = q$. As cost of the transition $q \xrightarrow{x_i} q \cup \{x_i\}$ we now count the nodes labeled with variable x_i in the BDD which is corresponding to the successor state $q \cup \{x_i\}$. (The variable x_i resides at the $(|q|+1)$-th level of this BDD.) In g we accumulate the transition costs along the newly found path to the successor. Further, for the initial state we set $g(\emptyset) = 0$. Now, by this inductive definition of g, the accumulated cost $g(q)$ for each state $q \subseteq X_n$ associated with a BDD F as described above is the number of BDD nodes labeled with a variable in q. The cost for the goal state X_n now is (as intended) the size of the BDD associated with it. This BDD has the variable ordering annotated along the minimum cost path by construction. Thus, this variable ordering must in fact be optimal.

The heuristic function $h\colon 2^{X_n} \longrightarrow \mathbb{N}$ used in our approach is the one used in [5, 7], a lower bound adapted from VLSI design. For state q, it counts the number of cofactors with respect to the variables in q (see Fig. 1). It can be computed effectively with a top down graph traversal on the BDD, counting the number of direct references from the upper nodes to the nodes in the lower part of the BDD [5]. This is the tightest lower bound known today.

IV. Algorithm

In this section we describe the implementation techniques used in the new A^*-approach to exact BDD minimization, called A^{state}. A sketch of the algorithm is given in Fig. 2. The algorithm is based on the implementation of [6], the best exact BDD minimization algorithm known so far, outperforming [7]. Thus we assume the presence of all techniques known from this method and all previous approaches without further mentioning, e.g. the use of functional aspects like symmetries. Next, we describe new implementation techniques unique to our A^*-based method.

A. Combination with B&B

In this section we present two techniques to combine A^* with B&B. References to line numbers all refer to the according lines in Fig. 2. First, we cite a well-known result from search theory [10].

Theorem 1 *Consider a state q in an A^*-algorithm operating with evaluation function φ. (C^* again denotes the minimal cost.) Then we have:*

If $\varphi(q) > C^*$ then state q will not be expanded.

Note that $\varphi(q)$ can change during algorithm run. In our context, $C^* + 1$ is the minimal BDD size, as φ, g and h count inner nodes only. Thus we have to add one for the constant node. In a B&B algorithm to determine a minimal cost, a lower and an upper bound on the minimum are constantly updated during the algorithm run every time new information is available to tighten the bounds. Next we *combine* the A^*-approach to exact BDD minimization with B&B by continuously updating an upper bound (denoted *upper_bound*) during algorithm run every time a smaller BDD size is found.

```
(1)  compute_optimal_ordering(BDD F, int n) { /* F represents f: B^n → B^m */
(2)      upper_bound := update_upper_bound(); /* first upper bound: size of initial BDD */
(3)      g[hash(∅)] := 0; h[hash(∅)] :=  number of output nodes m;
(4)      states[hash(∅)] := ∅; insert ∅ into OPEN;
(5)      while 1 do    /* until doomsday */
(6)          determine q ∈ OPEN with minimal g[hash(q)] + h[hash(q)];
(7)          if g[hash(q)] + h[hash(q)] + 1 = upper_bound then
(8)              reconstruct ordering that yielded upper_bound ; doomsday:=1;
(9)          end–if
(10)         if q = X_n then reconstruct ordering as the sequence of the path to q; doomsday:=1; end–if
(11)         if doomsday then exit while loop; end–if
(12)         reconstruct an appropriate ordering π with π(q) = q with path reconstruction and establish π on F;
(13)         upper_bound := update_upper_bound(); /* upper bound: smallest BDD size seen so far */
(14)         for each x_i ∈ X_n \ q do
(15)             q' := q ∪ {x_i};
(16)             if q' ∉ states then g[hash(q')] := ∞; end–if
(17)             newcost := label(F, |q'|) + g[hash(q)]; /* label(F, |q'|) is computed without variable shifting */
(18)             if q' ∉ states or newcost < g[hash(q')] then
(19)                 g[hash(q')] := newcost;
(20)                 p(q') := x_i; /* needed for the path reconstruction */
(21)                 if q' ∉ states then states[hash(q')] := q'; end–if
(22)                 if h[hash(q')] not yet computed and g[hash(q')] + n − |q'| + 1 ≤ upper_bound then
(23)                     shift x_i to level |q| + 1;
(24)                     upper_bound := update_upper_bound();
(25)                     h[hash(q')] := compute_lower_bound(q');
(26)                 end–if
(27)                 if h[hash(q')] computed and g[hash(q')] + h[hash(q')] + 1 ≤ upper_bound then
(28)                     if OPEN too crowded, resize it and garbage-collect outdated entries;
(29)                     insert q' into OPEN;
(30)                 end–if
(31)             end–if
(32)         end–for
(33)     end–while
(34) }
```

Fig. 2. The new Algorithm A^{stute} (sketch)

A.1 Delayed State Insertion

Our integration of B&B into A^* is based on the following observation: Clearly, we have

$$upper_bound \geq C^* + 1. \tag{1}$$

Now let q be a state with $\varphi(q) + 1 > upper_bound$. With (1) we have $\varphi(q) + 1 > C^* + 1$ and thus, by Theorem 1, q will not be expanded by A^*.

In the new approach, this result is used as follows: we insert a state q into OPEN iff $\varphi(q) + 1 \leq upper_bound$ (see line (27)). Otherwise, we delay the insertion until $g(q)$ has become small enough (by continuous updates during algorithm run) to meet this condition. By this strategy, we avoid the inclusion of many states into OPEN which never will be expanded. This reduces the memory requirement of the algorithm. Note that in line (22) we also delay the computation of $h(q)$ to save unnecessary computations.

A.2 Early Termination

Integrating B&B also allows to further prune the state space, based on the following consideration: Let q be a state which is chosen as the next best state, i.e. q will be expanded. Inversion (and adding one to each side of the inequality on the left side) of the statement in Theorem 1, together with (1) yields

$$\varphi(q) + 1 \leq C^* + 1 \leq upper_bound. \tag{2}$$

Now let $\varphi(q) + 1 = upper_bound$. We have $upper_bound = C^* + 1$ by (2), i.e. the upper bound has already reached the minimum.

In the new approach, this result is used as follows: if the total cost of the state currently considered (plus one, for the constant node) already equals the upper bound, we terminate and reconstruct the variable ordering which yielded this upper bound (see lines (7)-(9)). This "early termination" results in a further pruning of the search space.

B. Reconstruction Techniques

In Algorithm A^{stute}, a large reduction in memory requirement is achieved by the following strategy: Instead of storing and continuously updating the whole sequence of variables on the cheapest known path from the initial state to a state q, only *one* variable is stored in $p(q)$. This is the variable denoted at the transition of the predecessor to q, which yielded the cheapest known path to q so far. The cheapest known path then can be reconstructed inductively, when required.

As all exact BDD minimization algorithms suggested so far, the new A^*-based approach needs to reconstruct BDDs respecting a certain variable ordering: prior to expansion of a state q, a BDD respecting an ordering π with $\pi(q) = q$ is reconstructed in line (12) of Algorithm A^{stute}. The first $|q|$ positions of such an ordering are obtained with the new path reconstruction technique. The remaining last $n - |q|$ positions must be chosen such that an unacceptable large increase of the size of the lower part of the BDD is avoided. In our new approach, we set the ordering of the variables in the lower part such that the same relative ordering of variables as in the *initial* ordering is established. While the approach in [7] tried to avoid moving away from good orderings, our approach is directly oriented towards an ordering known to be good. Our experiments show that this is more effective, see Section V.

V. Experimental Results

All experimental results have been carried out on a machine with an Athlon processor running at 1.4 GHz, with a main memory of 1.5 GByte and a runtime limit of 20,000 CPU seconds. The new algorithm in its final stage of development is called A^{stute}. In our experiments we also included an earlier version of the algorithm without the path and BDD reconstruction techniques described in Section IV.B: this algorithm uses the best known standard techniques instead and is called A^{stir}.

The implementation of the new algorithms A^{stute} and A^{stir} is based on the implementation of NEO [6], which outperforms

TABLE I
COMPARISON OF NEO, A^{stir} AND A^{stute}

name	in	out	opt	NEO			A^{stir}			A^{stute}		
				time	space	avg. swaps	time	space	avg. swaps	time	space	avg. swaps
cc	21	20	46	69.4s	36M	48.52	51.6s	85M	14.48	51.1s	36M	14.60
cm150a	21	1	33	220s	37M	47.39	120s	88M	7.22	119s	39M	7.63
comp	32	3	95	3520s	121M	125.65	1894s	649M	29.86	1477s	288M	33.96
cordic	23	2	42	1.93s	$<$ 1M	25.47	1.35s	$<$ 1M	18.82	1.29s	$<$ 1M	19.17
cps	24	102	971	1893s	61M	53.85	1373s	186M	39.97	1380s	78M	40.05
i1	25	16	36	22.1s	10M	51.28	17.2s	25M	12.87	18.0s	11M	12.98
lal	26	19	67	440s	79M	73.65	243s	378M	25.01	242s	145M	25.50
mux	21	1	33	220s	36M	47.27	120s	88M	7.24	119s	39M	7.68
pcle	19	9	42	5.40s	3M	24.19	3.61s	6M	12.79	3.75s	3M	13.60
pm1	16	13	40	0.62s	$<$ 1M	22.33	0.61s	$<$ 1M	10.25	0.54s	$<$ 1M	11.23
s208.1	18	9	41	4.33s	2M	24.20	3.84s	4M	17.79	3.72s	3M	17.10
s298	17	20	74	7.31s	2M	29.80	6.17s	6M	17.52	6.00s	3M	17.56
s344	24	26	104	906s	111M	70.60	537s	363M	28.74	527s	146M	29.10
s349	24	26	104	905s	111M	70.60	537s	363M	28.74	527s	146M	29.10
s382	24	27	119	409s	75M	44.38	313s	359M	14.06	390s	143M	14.07
s400	24	27	119	409s	75M	44.38	312s	359M	14.06	390s	143M	14.07
s444	24	27	119	473s	75M	46.98	314s	362M	14.25	324s	144M	14.16
s510	25	13	146	5490s	414M	74.90	3568s	1264M	25.05	3308s	596M	24.55
s526	24	27	113	705s	111M	63.74	365s	365M	30.26	391s	149M	30.25
s820	23	24	220	848s	59M	59.16	734s	180M	43.46	733s	76M	43.45
s832	23	24	220	889s	59M	58.79	731s	181M	43.46	730s	76M	43.41
sct	19	15	48	6.36s	3M	25.59	3.86s	6M	19.85	3.83s	3M	19.90
tcon	17	16	25	0.79s	$<$ 1M	21.56	0.55s	1M	7.69	0.52s	$<$ 1M	7.69
ttt2	24	21	107	435s	82M	53.54	345s	360M	30.55	362s	144M	30.59
vda	17	39	478	32.3s	3M	27.66	26.0s	6M	35.50	25.8s	4M	35.54

the approach of [7]. All algorithms have been integrated in the CUDD package [15]. By this we guarantee that they run in the same system environment. In a series of experiments we applied all algorithms to the set of benchmark circuits from LGSynth93. The results are given in Table I. In the first column of Table I the name of the function is given. *in* (*out*) denotes the number of inputs (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 the algorithms are given. For every algorithm, column *avg. swaps* gives the number of variable swaps needed on average to set the variable ordering for the BDD representing the next state.

As the results in Table I show, the new algorithm A^{stute} has a reduced memory requirement when compared to the earlier version A^{stir} by up to 61.6% (e.g. see *lal*), on average the reduction is 57.5%. In this, with the memory configuration of todays computer systems, the A^*-based approach remains practical.

The new BDD reconstruction technique of A^{stute} reduces runtime by up to 22.0% compared to A^{stute} (e.g. see *comp*). The new method successfully avoids BDD explosions and achieves a significant speed up.

The new algorithm A^{stute} is much faster than the algorithm NEO. In comparison to NEO, we obtained reductions in runtime of up to 58.0% (e.g. see *comp*, *mux*, *cm150a*). On average we have a gain of 37.8%. One reason for this large gain is the reduction in the number of state expansions, which can be up to 40.9%: e.g. for *s526*, A^{stute} expands only 852179 states, whereas NEO, analogous to state expansions, extends 1442625 variable orderings. As can be expected from search theory [4], the best-first strategy can yield large reductions of state expansions especially for the "harder" instances (complex circuits), which directly transfers to a reduction of runtime.

In our experiments, we identified a second reason for the high speedup of A^{stute} compared to NEO: typically, many states with the same φ-value are reconstructed for expansion one after the other. Such states are represented by BDDs with a very similar variable ordering: in our experiments, the number of variable swaps needed to transform a given variable ordering into the ordering for the next state to expand is reduced by up to 83.9% in comparison to NEO. E.g., for *cm150a* the average number of swaps needed by NEO is 47.39, but only 7.63 for A^{stute}. On average we have a reduction in the number of variable swaps of 54.9%.

ACKNOWLEDGEMENTS

The authors would like to thank Sven Lamberti who contributed to this work during an undergraduate course at the University of Kaiserslautern.

REFERENCES

[1] B. Bollig and I. Wegener, "Improving the variable ordering of OBDDs is NP-complete," *IEEE Trans. on Comp.*, Vol. 9(45), pp. 993-1002, 1996.
[2] K. Brace, R. Rudell, and R.E. Bryant, "Efficient Implementation of a BDD Package", in *Design Automation Conf.*, pp. 40-45, 1990.
[3] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. on Comp.*, Vol. 8(35), pp. 677-691, 1986.
[4] R. Dechter and J. Pearl, "Generalized best-first search strategies and the optimality of A^*," *Journal of the Association for Computing Machinery*, Vol. 1(34), pp. 1-38, 1987.
[5] R. Drechsler, N. Drechsler, and W. Günther, "Fast exact minimization of BDDs," *IEEE Trans. on CAD*, Vol. 3(19), pp. 384-389, 2000.
[6] R. Ebendt, "Reducing the number of variable movements in exact BDD minimization," in *Int'l Symp. on Circuits and Systems*, Vol. 5, pp. 605-608, 2003.
[7] R. Ebendt, W. Günther, and R. Drechsler, "Combination of lower bounds in exact BDD minimization," in *Design, Automation and Test in Europe*, pp. 758-763, 2003.
[8] S. Friedman and K. Supowit, "Finding the optimal variable ordering for binary decision diagrams," in *Design Automation Conf.*, pp. 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*, pp. 38-41, 1993.
[10] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybern.*, Vol. 2, pp. 100-107, 1968.
[11] N. Ishiura, H. Sawada, and S. Yajima, "Minimization of binary decision diagrams based on exchange of variables," in *Int'l Conf. on CAD*, pp. 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*, pp. 546-551, 2001.
[14] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *Int'l Conf. on CAD*, pp. 42-47, 1993.
[15] F. Somenzi, *CU Decision Diagram Package Release 2.3.1*, University of Colorado at Boulder, 2002.