A SOP Minimizer for Logic Functions Described by Many Product Terms Based on Ternary Trees

David Toman, Petr Fišer
Czech Technical University in Prague
email: tomandav@fel.cvut.cz, fiserp@fit.cvut.cz

Abstract

We introduce a new efficient minimization method for functions described by many (up to millions) product terms. The algorithm is based on processing a newly proposed efficient representation of a set of product terms – a ternary tree. The minimization procedure is based on a fast application of basic Boolean operations upon the ternary tree, combined with algorithms used in the Espresso minimizer. Minimization of incompletely specified functions is supported as well. The minimization method was tested on randomly generated large sums-of-products and collapsed ISCAS benchmark circuits. The performance of the proposed algorithm was compared with Espresso.

1 Introduction

Logic functions described by a sum-of-products (SOP) form frequently appear in the logic synthesis process. Even though "more efficient" function representations, like binary decision diagrams (BDDs) [1] or and-invert-graphs (AIGs) [2] were proposed and are widely used in logic synthesis tools, SOP forms still remain an ultimate solution to representing logic functions, mainly due to the fact that most of logic synthesis algorithms are based on processing SOPs. SOPs are usually a starting point for decomposition and technology mapping algorithms [3], [4].

The need for minimization of the number of terms of the SOP form is apparent. Starting with the basis of minimization algorithms stated in 1950's by Quine and McCluskey [5], many different minimizers have been developed [6][7][8][9][10]. All these methods suffer from their specific drawbacks when solving problems of different kinds. For example, Espresso [7] lacks in quality and runtime for functions with a large number of inputs (>100). BOOM [10] solves this problem efficiently, but, on the other hand, it needs to have the function's off-set specified explicitly, which limits its usability in cases of functions specified by their on-sets only.

In many logic synthesis processes, there often appear functions described by SOPs with an extremely large number of product terms (up to millions). For example, it is often advantageous to collapse a multi-level circuit network into a two-level representation, to obscure the original circuit's structure to the subsequent synthesis process [11], in hope of obtaining better results. In order to accomplish such a process, large SOPs often cannot be avoided. As a second example, there is often a need to compute a complement of a logic function described by a SOP [7]. The resulting SOP size grows exponentially with the number of input variables.

For these reasons, there is a crucial need for a memory-efficient compact representation of SOPs with many terms, as well as for a fast and efficient minimization algorithm for such function representation. As for the efficient representation of SOPs, zero-suppressed BDDs (ZDDs) were proposed [12]. However, no efficient minimization algorithms built upon ZDDs are known.

We propose a SOP representation based on a "*ternary tree*". The ternary tree concept was firstly proposed in [13], as an efficient storage of terms in SOP. Compared to BDDs where the size can grow exponentially with the number of input variables, size of ternary tree grows only linearly with the number of inputs in the worst case. The first simple ternary tree minimization algorithms were proposed in [16], [17]. Ternary trees most closely resemble SOP ternary decision diagrams (TDDs), briefly described in [14] and term trees [15]. We basically extend the SOP TDDs notion by introducing new operations and their new application areas.

In this paper we propose a ternary tree based minimization algorithm for incompletely specified functions described by SOPs. We show that the ternary tree representation of SOPs having many

product terms benefits from a lower computational complexity, compared to the standard tabular SOP representation.

2 Preliminaries & Problem Statement

Let us have a single-output Boolean function of n input variables. The input variables will be denoted as x_i, $0 \leq i < n$. Output values of the on-set terms (both minterms and product terms of higher dimensions may be used) are defined by a truth table. The function may be incompletely specified, i.e., some terms may be assigned to the don't-care set.

Thus, we have an n-variable Boolean function defined by a sum-of-product (SOP) form as an input. The number of product terms will be denoted as p. The *cover* of a function is a set of on-set terms implicating the whole function, possibly combined with some terms of the DC-set. Our aim is to minimize the size of the cover, i.e., the number of on-set terms in the result. The secondary aim could be the reduction of the number of literals in the terms, thus increasing the dimension of the terms.

3 Ternary Tree

The ternary tree, proposed in [13], is a structure used for storing of product terms. This is in contrast to, e.g., BDDs, which describe the function, but not its representation. Like in the case of BDDs the ordering of variables can have big impact on the ternary tree size (if a variable that appears in many terms is placed on the top of the tree, its size will be smaller than if the variable is placed to the bottom).

An example of a ternary tree for a 3-input function is shown in Figure 1. Three terms are contained in the tree, particularly (001), (-1-) and (10-). Each non-terminal node u may have three potential children, $lo(u)$, $dc(u)$, $hi(u)$. In our example, $lo(u)$ is the left child, $dc(u)$ the middle one and $hi(u)$ the right one.

When inserting a term into the tree, at the i-th level of the tree a branch is chosen according to the polarity (0, -, 1) of the i-th variable in the term. If the corresponding branch is present, we follow it, if not, the branch is newly created. The tree is thus gradually being constructed by appending product terms. The maximum size of the ternary tree is obviously $O(3^n)$, since there are 3^n different terms for an n-input function. However, the real ternary tree size is usually much less.

Complexities of operations performed using ternary trees are usually less than those performed using tabular SOP representations (up to n-times speedup in the best case for most operations). For example, the term look-up speed time complexity is $O(n)$, instead of $O(n.p)$ for the tabular representation [16] (the time consumption of some operations for the tabular representation could be further reduced by factor of 32 or 64 (depending on the platform word size) using parallelism on the bit level, but the overall time complexity remains still the same).

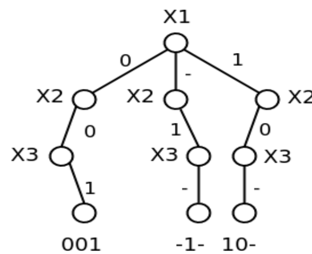

Figure 1: Ternary tree example

4 Ternary tree minimization

The basic ternary tree minimization algorithm was proposed in [16] for the first time, later improvements were published in [17]. In this paper, we extend the algorithm so it is capable of turning all implicants into primes and also capable of removing some of the redundant terms.

The overall minimization algorithm comprises of two steps: first, the SOP is processed by a fast minimization algorithm, in order to quickly reduce the number of SOP terms by applying basic rules of Boolean algebra. Then, the result is further refined by applying Espresso-like minimization steps.

4.1 Overall Minimization Algorithm

The overall minimization algorithm can be described by the following pseudo code:

```
Minimize (F,D,R)
{
  on_set = Create_TernaryTree(F); // create ternary trees
  if (D) dc_set = Create_TernaryTree(D);
  if (R) off_set = Create_TernaryTree(R);

  for (i = 0; i < n; i++) // perform the fast minimization
  {
    // n is the number of inputs

    on_set->Merge_Leaves();
    on_set->Rotate();

    dc_set->Merge_Leaves();
    dc_set->Rotate();

    off_set->Merge_Leaves();
    off_set->Rotate();
  }

  on_set = on_set->Absorb_Terms();
  dc_set = dc_set->Absorb_Terms();
  off_set = off_set->Absorb_Terms();

  fd_set = on_set->Absorb_Into(dc_set); // absorb additional variables
  fd_set = fd_set->Redundancy_Check_1(); // perform basic redundancy check

  if (!R) off_set = Get_Complement(on_set,dc_set); // extract the off-set

  fd_set = fd_set->Expand_Cover(off_set); // perform expansion
  fd_set = fd_set->Redundancy_Check_2(); // perform deeper redundancy check
  fd_set = fd_set->Reduce_Cover(); // perform reduction
  fd_set = fd_set->Expand_Cover(off_set); // perform the final expansion

  FD_min = fd_set->Dump_TernaryTree();
  return FD_min;
}
```

Algorithm 1: The overall minimization algorithm

First, ternary trees representing the on-set, don't care set and off-set (if defined) are constructed from the PLA description of the source function.

Then, the fast minimization algorithm is applied (see Subsection 4.2) to the on- and DC-set trees. This procedure substantially reduces the size of the trees, before they are processed further. The reason for processing the on-set and DC-set separately is in preventing of merging on- and DC-terms, that could result in redundancy.

In the next step, all terms of the on-set tree than can be merged are merged with the DC-set (based on the negation absorption and complement rules), further expanding the cover - for example if an on-set term is adjacent to a DC-term and one of the rules can be applied on the on-set term, then the expansion is performed. The result of this operation yields the fd-set. The fd-set is then composed of

all terms of the on-set and possibly some terms of the DC-set (when a variable of an on-set term is absorbed into a DC-term, then the resulting term contains minterms from both on-set and DC-set).

After these steps a basic redundancy check is performed upon the fd-set to simplify it even more, without having a significant impact on the time consumption.

The following step lies in extraction of the off-set (if the off-set is not known already) by exploitation of the complementation algorithm, because the knowledge of off-set is necessary to perform the expansion step efficiently.

Once the off-set is known, an expansion can be performed upon the fd-set, to obtain a prime cover.

After the expansion, another (deeper) redundancy check is performed upon the fd-set, capable of removing more redundant terms, but also taking more time than the basic redundancy check. The depth of this redundancy check can be selected by the user (this parameter influences the quality of the result and time complexity of the operation).

To move from the locally optimal solution in search for a global optimum, next step – the reduction follows.

The final step of the minimization is again an expansion, to find another local optimum and make the result as sparse as possible.

The reason why the reduction-expansion step is not performed iteratively (like in Espresso) is that the reduction still isn't that fast and efficient and it would take too long without yielding much improvement.

The individual minimization steps will be described into detail in the following subsection.

4.2 Fast Minimization

The first part of the fast minimization algorithm is based on applying basic absorption and complement property rules of Boolean algebra, targeting a reduction of the number of the ternary tree terminal nodes (leaves). This is achieved by the leaf merging and tree rotation. The method itself consists of iterative cutting of the root node and moving it to the bottom of the tree, where the leaves can then be merged, reducing size of the tree. Details of this method are further described in [17]. Example of this algorithm is shown in Figure 2 and Figure 3.
The two rules mentioned above can be expressed in the following way:

$$\text{The one-variable absorption rule: } a + ab = a \quad (1)$$

$$\text{The complement property rule: } ab + ab' = a \quad (2)$$

The minimization process can be iterated several times. We have found experimentally, that iterating n-times yields satisfactory results. Additional iterations usually do not bring a significant improvement. The asymptotic worst case time complexity of this algorithm is $O(n^2.p)$ ($O(n.p)$ in the best case scenario).

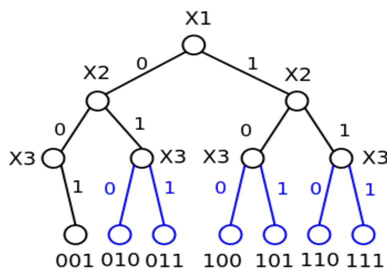

Figure 2: Ternary tree before leaf merging

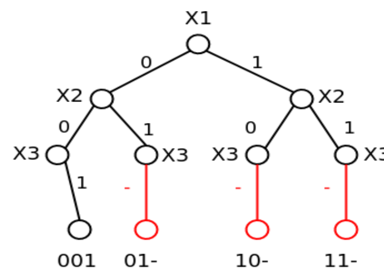

Figure 3: Ternary tree after leaf merging

After that we apply general absorption and negation absorption rules, to remove all the terms that are subsets of other terms and to absorb variables in terms that differ in more than one variable.

$$\text{The general absorption rule: } a + abcd = a \quad (3)$$

$$\text{The general negation absorption rule: } a + a'bcd = a + bcd \quad (4)$$

The worst case time complexity of this algorithm is $O(n.p^2)$. However, for dense trees, it is much less in practice ($O(p^2)$ in the best case, when $p \approx 3^n$). It is also difficult to estimate the absorption time

complexity as a function of the number of the ternary tree nodes, since it heavily depends on the tree structure.

The overall worst case time complexity of these two steps combined together is then $O(n^2.p + n.p^2)$. For details of this algorithm see [17].

4.3 Complementation

Knowledge of the off-set allows performing the expansion step in much shorter time. If the off-set is not explicitly given in the source file, it is computed as a complement of the union of the on-set and DC-set.

Similarly like in Espresso [7], the Shannon expansion theorem was used for this purpose. The complement is obtained by a recursive application of (5), until a trivial solution (complementation of a single variable) is reached. The worst case time complexity of the complementation algorithm is $O(2^n)$.

$$\overline{f} = \overline{x_j}.(\overline{f_{\overline{x_j}}}) + x_j(\overline{f_{x_j}}) \quad (5)$$

The whole complementation process is carried out in the following way:

1. Perform the Fast minimization upon the cover (which is the union of the on- and DC-set)
2. If the cover is tautology return empty complement (emerges at 5. from the recursion)
3. If the cover is non-satisfiable (i.e. if the tree is empty) return a term containing only DCs (emerges at 5. from the recursion)
4. Perform the recursive split with the variable that appears the most in the cover based on the Shannon expansion theorem (creates two new recursion branches starting at 1.)
5. Merge the results obtained from both recursion branches
6. Minimize the merged results using the Fast minimization algorithm
7. Return the merged and minimized results

4.4 Cover Expansion

The cover expansion is capable of turning all implicants in the given set into primes and is therefore one the most fundamental steps of the algorithm.

The expansion itself goes as follows - we traverse the tree and heuristically choose and enter the branches that have the highest number of DCs, because such branches also most probably contain terms with the highest number of DCs (the biggest terms). We then try to expand each of these terms by setting a selected variable to DC followed by checking, whether the modified term intersects the off-set. If not, the expanded term is a valid implicant and thus the expansion is committed. The whole process is repeated until there is no further improvement.

Selection of the term to be expanded is done based on the assumption that the biggest term has the highest chance of covering other terms which then can be removed from the set.

The strategy of selecting the expansion direction (i.e. the variable that will be set to DC) is simple - we always set to DC the most binate variable (greedy approach). The rationale of this choice is again that the expanded term may cover other terms more easily, or at least we will be able to absorb some variables in adjacent terms based on the negation absorption rule.

The worst case time complexity of this operation is only $O(n.p_1p_2)$ (where n is the number of variables, p_1 is the number of terms in the on-set and p_2 the number of terms in the off-set) because in the worst case we have to compare each term from the on-set with the whole off-set . This estimation seems to be however overpessimistic, because even though the worst case time complexity is the same as for the tabular representation, it is much less in the average case.

4.5 Redundancy Check

The procedure of removing redundant terms consists in computing an intersection of each term with all other terms and minimizing the result using the Fast minimization algorithm (see Subsection 4.2). If the minimized result is identical to the original term (which means that the original term is completely covered by other terms), then the term can be safely removed from the cover.

The order in which the redundancy check is performed upon the tree is guided by a heuristic which first enters the branches that have the lowest number of DCs, because such branches most probably contain terms with the smallest dimension and such terms contribute the most to the size of the cover.

The time complexity of this operation is in this case given by two facts - we need to compute an intersection of a term with other terms, which can be done in $O(n.p^2)$ and then continuously minimize the result, which can be done in $O(i.n.p^2)$ (where i is the number of minimization iterations, n is the number of variables and p the number of terms).

To control the result quality and time complexity of this operation, the i parameter can be indirectly controlled by the user by choosing how deep should the redundancy check go in search for a term cover (i.e. how big may the conjunction terms be). There is however no guarantee that this method will remove all the redundant terms from the cover, even when the depth of redundancy check is set to maximum (the number of input variables).

The worst case time complexity (when $i = n$) is then $O(n^2.p^2)$, which is again the same as for the tabular representation, but overall much lower in average case.

4.6 Cover Reduction

This step is very similar the redundancy check step because it uses practically the same algorithm with only slight differences.

Like during the redundancy check we compute the intersection of the examined term with other terms from the cover and minimize it. If the minimized result contains a term that is half of the size of the examined term, we can reduce this term by leaving only the uncovered part of it in the set while still covering all the minterms covered previously. By repeating the whole process over and over we can then reduce each term in the cover to its minimum feasible dimension. Once we have the reduced cover, the expansion step can be applied again to possibly obtain a better result.

Time complexity of this operation is the same as for the redundancy check - $O(i.n.p^2)$.

5 Experimental Results

The results of the minimization of collapsed benchmark circuits from [18], [19] and [20] are shown in Table 1. All the functions are completely specified, single-output functions are considered only. The benchmark name is given in the first table column, followed by the number of its inputs. The numbers of terms and literals of the source PLAs are shown in the third column. The TT-Min and Espresso minimization runtimes are given next. The result complexities are shown in the last two columns.
It is apparent that Espresso starts having problems minimizing the circuits where the size reaches 50,000 terms and isn’t able to deal with majority of the listed benchmarks at all, while TT-min manages to minimize significantly all of them. The instances where Espresso manages to deal with the benchmarks are without exception cases where the circuits are easily minimized.

Table 1: Minimization results of selected benchmarks

Benchmark			*Time [s]*		*Terms / Literals*	
Name	*Inputs*	*Terms / Literals*	*TT-min*	*Espresso*	*TT-min*	*Espresso*
taut1	25	5,000 / 50,015	8.24	3.76	1 / 0	1 / 0
taut2	25	50,000 / 686,138	212.32	251.88	20 / 20	20 / 20
taut3	25	100,000 / 1,623,180	2525.45	15885.35	65,567 / 886,180	-
g25_15	25	79,056 / 1,311,480	27.36	-	18,720 / 295,446	-
g25_19	25	58,968 / 950,004	26.71	-	16,785 / 261,618	-
leku-cd_15	25	79,056 / 1,311,480	36.48	-	12,114 / 189,717	-
leku-cd_19	25	58,968 / 950,004	32.96	-	12,096 / 188,172	-
s420_12	35	113,280 / 2,577,502	5.82	107.86	17 / 170	17 / 170
c432_2	36	786,562 / 19,910,685	8928.99	14842.05	109,192 / 1,211,341	-
c432_4	36	866,664 / 21,865,362	736.39	832.82	7,128 / 60,512	7,128 / 60,512

Results of minimization of randomly generated incompletely specified functions are shown in Table 2. The meaning of the columns is the same as in the previous case, except for the second column, which denotes the number of input/output DCs in the benchmark. It is apparent that Espresso gives better

results, but again fails to solve one of the benchmarks and it's time consumption rises much faster with the number of terms than for TT-min.

Table 2: Minimization of randomly generated benchmarks

Benchmark			*Time [s]*		*Terms / Literals*	
Inputs	*idc / odc*	*Terms / Literals*	*TT-min*	*Espresso*	*TT-min*	*Espresso*
20	35 / 35	1,000 / 12,964	32.88	2.51	658 / 8474	658 / 8474
20	35 / 35	2,000 / 26,083	50.37	9.48	1277 / 16407	1273 / 16362
20	35 / 35	5,000 / 65,069	52.03	41.27	3039 / 37442	2941 / 36177
20	35 / 35	10,000 / 129,813	109.26	162.06	5400 / 58214	3310 / 35474
20	35 / 35	20,000 / 259,953	52.63	-	1794 / 11722	-
20	35 / 35	50,000 / 649,332	52.57	10.07	1 / 0	1 / 0

Analysis of the TT-min algorithm steps for benchmark c432_4 (the 4th output function of the benchmark c432) is shown in the Table 3. These results show that the first two steps reduce the function size most significantly in this case. To achieve the same result as Espresso it was necessary to perform the expansion step however. The depth of the reduction/redundancy check was set to 3 (default value) in this case.

Table 3: Analysis of the TT-min algorithm steps

Benchmark: c432_4		
Step	*Time [s]*	*Terms / Literals*
Rotation	15.88	384,173 / 9,106,073
Absorption	224.93	291,036 / 4,095,394
Redundancy check 1	116.78	203,393 / 2,894,997
Complementation	290.25	-
Expansion 1	6.26	7,128 / 60,512
Redundancy check 2	49.01	7,128 / 60,512
Reduction	23.20	7,128 / 60,512
Expansion 2	1.94	7,128 / 60,512
Other	8.75	-

6 Conclusions

An algorithm for an efficient minimization of logic functions described by a sum-of-products form with many terms was proposed. The minimization method is based on processing a ternary tree, which has been found to be a very efficient representation of a set of product terms. Espresso-like minimization algorithms have been developed upon the ternary tree structure. As a result, the average case complexity of many algorithms is reduced, with respect to the standard tabular SOP representation (even though the worst case complexity is the same as for the tabular representation for most operations).

It was experimentally shown that for benchmarks with tens of thousands of terms Espresso usually yields the result in longer time than this method, or fails to produce any result whatsoever.

Another advantageous thing is that this method doesn't need to know the off-set to perform the basic operations and it could therefore find its application in cases where the complementation takes prohibitively long.

As the future work, we expect implementation of more efficient redundancy checks and reduction steps, which could possibly make this minimizer overall superior to Espresso even for small circuits, where Espresso still yields much better results, because it is capable of removing all the redundant terms from the cover.

Acknowledgement

This research has been supported by MSMT under research program MSM6840770014 and by the grant of the Czech Technical University in Prague, SGS10/117/OHK3/1T/18.

References

[1] S. B. Akers, "Binary decision diagrams", IEEE Trans. on Computers, Vol. C-27. No. 6, June 1978, pp. 509-516

[2] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting - a fresh look at combinational logic synthesis". In Proceedings of the 43rd Annual Conference on Design Automation, San Francisco, CA, USA, July 24 - 28, 2006, pp. 532-535

[3] S. Hassoun and T. Sasao, "Logic Synthesis and Verification", Boston, MA, Kluwer Academic Publishers, 2002, 454 pp.

[4] E.M. Sentovich et al. SIS: A System for Sequential Circuit Synthesis, Electronics Research Laboratory Memorandum No. UCB/ERL M92/41, University of California, Berkeley, CA 94720, 1992.

[5] E.J. McCluskey, "Minimization of Boolean functions", The Bell System Technical Journal, 35, No. 5, Nov. 1956, pp. 1417-1444

[6] S.J. Hong, R.G. Cain and D.L. Ostapko, "MINI: A heuristic approach for logic minimization", IBM Journal of Res. & Dev., Sept. 1974, pp.443-458

[7] R.K. Brayton et al., "Logic minimization algorithms for VLSI synthesis", Boston, MA, Kluwer Academic Publishers, 1984, 192 pp.

[8] R.L. Rudell and A.L. Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization", IEEE Trans. on CAD, 6(5): 725-750, Sept.1987

[9] P. McGeer et al., "ESPRESSO-SIGNATURE: A new exact minimizer for logic functions", Proc. DAC'93

[10] J. Hlavička and P. Fišer, „BOOM - a Heuristic Boolean Minimizer", Proc. ICCAD-2001, San Jose, Cal. (USA), 4.-8.11.2001, 439-442

[11] P. Fišer and J. Schmidt, "Small but Nasty Logic Synthesis Examples", Proc. 8th Int. Workshop on Boolean Problems (IWSBP'08), Freiberg, Germany, 18.-19.9.2008, pp. 183-190

[12] S. Minato, "Zero-suppressed BDDs for set manipulation in combinatorial problems". In Proceedings of the 30th international Conference on Design Automation (DAC), Dallas, Texas, USA, June 14 - 18, 1993, pp. 272-277

[13] P. Fišer and J. Hlavička, Implicant Expansion Method used in the BOOM Minimizer. Proc. IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS'01), Gyor (Hungary), 18.-20.4.2001, pp. 291-298

[14] T. Sasao, "Ternary Decision Diagrams - A Survey", Proc. of IEEE International Symposium on Multiple-Valued Logic, pp. 241-250, Nova Scotia, May 1997

[15] L. Jozwiak, A. Slusarczyk and M. Perkowski, "Term Trees in Application to an Effective and Efficient ATPG for AND-EXOR and AND-OR Circuits" , VLSI Design, Vol. 14, No 1, January 2002 , pp. 107-122

[16] P. Fišer, P. Rucký, and I. Váňová, "Fast Boolean Minimizer for Completely Specified Functions", Proc. 11th IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop 2008 (DDECS'08), Bratislava, SK, pp. 122-127

[17] Petr Fišer, David Toman, "A Fast SOP Minimizer for Logic Functions Described by Many Product Terms", Proceedings of 12th Euromicro Conference on Digital System Design (DSD'09), Patras, Greece, 27.8. – 29.8. pp. 757-764

[18] F. Brglez and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortan", Proc. of ISCAS 1985, pp. 663-698

[19] F. Brglez, D. Bryan and K. Kozminski, „Combinational Profiles of Sequential Benchmark Circuits", Proc. of ISCAS, pp. 1929-1934, 1989

[20] J. Cong and K. Minkovich: Optimality study of logic synthesis for LUT-based FPGAs, IEEE Trans. on CAD, vol. 26, pp. 230–239, Feb. 2007.