A Fast SOP Minimizer for Logic Functions Described by Many Product Terms

Petr Fišer, David Toman
Czech Technical University in Prague
Department of Computer Science and Engineering
e-mail: fiserp@fel.cvut.cz, tomand1@fel.cvut.cz

***Abstract* - We introduce a fast and 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. A significant speedup of the look-up of the term operation is achieved, with respect to a standard tabular function representation. The minimization procedure is based on a fast application of basic Boolean operations upon a ternary tree. Minimization of incompletely specified functions is supported as well. The minimization method was tested on randomly generated large sums-of-products, collapsed ISCAS benchmark circuits and SAT problems. The performance of the proposed algorithm was compared with Espresso. A very advantageous application of the new minimization algorithm has been found – if it is used for pre-processing a function having a large number of product terms, run prior to Espresso, the total minimization runtime is significantly reduced, whereas the result quality is not affected.**

Keywords - logic functions, two-level minimization, ternary tree, SOP, Espresso

I. INTRODUCTION

The problem of minimization of representations of combinational logic functions occurs in many areas of the logic design [1], in the built-in self-test (BIST) design [2], in a design of control systems, etc.

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) [3] or and-invert-graphs (AIGs) [4] 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 [1].

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 (MINI [6]), ending up in Espresso [7] with its later improvements [8, 9]. Scherzo [21], employing BDDs to represent prime implicants implicitly, allowed an exact minimization of rather difficult functions. Afterwards, a two-level Boolean minimizer Boom [10, 11] has been developed.

All these methods suffer from their specific drawbacks when solving problems of different kinds. For example, Espresso lacks in quality and runtime for functions with a large number of inputs (>100). BOOM 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 (and possibly also don't-care sets) only.

In a nowadays logic synthesis process, 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 [12, 13]. Then there is a big chance that the synthesis will perform much better, in case the structure of the original network is "misleading" for the synthesis. Binary decision diagrams [3] were considered as a solution doing the same job; however, for functions with many input variables, BDDs may become extremely (and unpredictably) large.

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 this function representation. As for the efficient representation of such SOPs, zero-suppressed BDDs (ZDDs) were proposed [14]. However, no efficient minimization algorithms built upon ZDDs are known.

We propose a SOP representation based on a "*ternary tree*". The ternary tree was firstly proposed in [15], as an efficient storage of terms in SOP. Ternary trees most closely resemble SOP ternary decision diagrams (TDDs), briefly described in [16] and term trees [17]. We basically extend the SOP TDDs notion by introducing new operations and their new application areas.

In this paper we propose a minimization algorithm for incompletely specified functions described by SOPs, working upon a ternary tree representation of SOPs.

The paper is structured as follows: after the introduction, the problem statement is given. Section III describes the

ternary tree structure and its properties. The ternary tree based minimization method is described in Section IV, the comparison results are shown in Section V. Section VI concludes the paper.

II. 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. These don't cares could be efficiently exploited in the minimization process.

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. Our aim is to minimize the number of terms (p). The secondary aim could be the reduction of the number of literals in the terms, thus increasing the dimension of the terms. Some of the possible function's don't cares are assigned to the on-set, in order to efficiently reduce the number of terms. Don't-cares that cannot be exploited by the minimization are retained, so they can be used in the post-minimization (see Section V).

III. Ternary Tree

We propose a ternary tree representation of an SOP form. This representation brings many benefits, with respect to a standard tabular (truth table) representation. For example, the term look-up speed time complexity is O(n), instead of O($n.p$) for the tabular representation [18].

The ternary tree depth is equal to the number of inputs of the function (n). Let us define a total ordering < over the set of input variables, the function *var*(i) gives the input variable of the function in the i-th order. Each level of the ternary tree corresponds to one variable, according to the ordering. Each non-terminal node in a level i corresponds to a "partial" product term, where values of only *var*(0)…*var*(*i-1*) variables are defined. Terminal nodes correspond to completely described terms.

An example of a ternary tree for a 3-input function is shown in Fig. 1. Three terms are contained in the tree, particularly 0-0, 10- and 11-. Each non-terminal node u may have three potential children, *lo*(u), *dc*(u), *hi*(u). In our example in Fig. 1, *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.

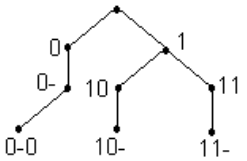

Figure 1. Ternary tree example

IV. Ternary Tree Minimization

The basic ternary tree minimization algorithm was firstly proposed in [18]. In this paper, we extend the algorithm so it is capable of minimizing incompletely specified functions and performing more complex Boolean operations upon the ternary tree.

A. Basic Algorithm

The basic minimization algorithm is based on applying absorption and complement property rules of Boolean algebra only, targeting the reduction of the number of the ternary tree terminal nodes (leaves). Particularly, if a non-terminal node at the (n-1)-th level has two successor nodes (which are thus terminals), they always may be merged to obtain one DC terminal, either by applying the absorption rule (in a case of a 0- or 1-terminal together with a DC terminal) or a complement property rule (in a case of a 0- and 1-terminal).

The principles of the reduction are illustrated by the following example. Let us consider a function $y = x_1 + x_2 + x_3$ described by a set of its on-set minterms. The ternary tree containing these minterms is shown in Fig. 2.

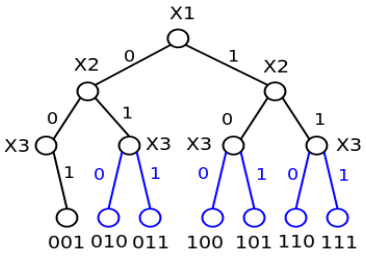

Figure 2. Minimization example (1)

There are seven terminal nodes representing the on-set minterms. It can be easily seen that minterm couples (010, 011), (100, 101) and (110, 111) may be merged, to obtain DC terminals. See Fig. 3.

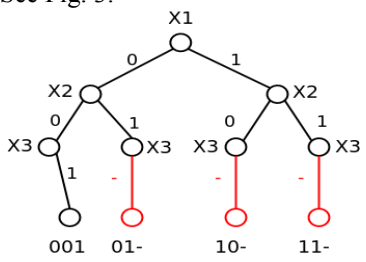

Figure 3. Minimization example (2)

The Complement property rule of Boolean algebra has been applied to the variable x_3, yielding a reduction of the number of terms from seven to four. Since the reduction operations may be performed on the leaves of the tree only, no other tree reduction steps can be done at this time, thus another phase of the minimization algorithm follows – the tree rotation.

The tree rotation is done by cutting off the root node, which yields three separate trees (at most). Then, the root variable is appended to all leaves of the trees. The rotation of the tree from Fig. 3 is shown in Fig. 4. The tree is split into two trees only, since the root of the original trees has only two successors (0 and 1).

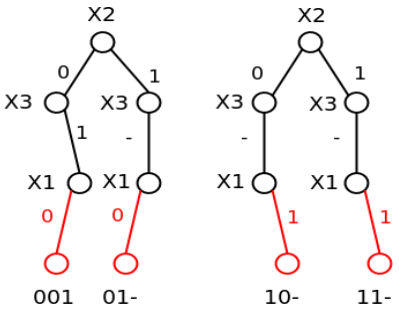

Figure 4. Minimization example (3)

After the splitting the trees are merged together, by traversing these trees from their roots in parallel and merging nodes. The result of the tree merging is shown in Fig. 5. Notice that the four terminals remain unchanged; the rotated tree describe the same set of terms as in Fig. 3.

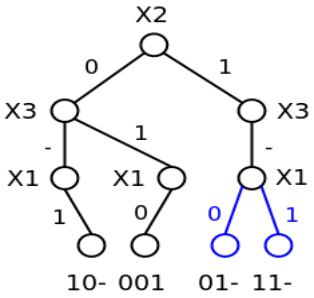

Figure 5. Minimization example (4)

Now the new terminals may me merged by applying the rules of Boolean algebra described above. Each terminal merging results in a removal of a particular (terminal) variable from a term.

The tree is rotated n-times (where n is the number of variables). The resulting ternary tree after 3 rotations is shown in Fig. 6, representing three terms (001, -1-, 10-), which is the minimum representation of the source function.

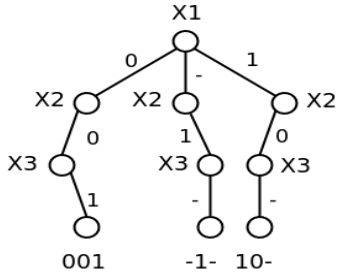

Figure 6. Minimization example (5)

The same minimization process may be performed upon a tabular representation of the function; nevertheless using the tree representation is more efficient, in terms of the computational time. The leaf merging procedure corresponds to merging (and thus eliminating) the truth table rows, which differ in the last variable only. Rotation of the tree corresponds to the rotation of the truth table columns. The asymptotic complexity of the proposed algorithm is computed as follows: each rotation with terminals merging operation involves traversing the whole tree once, thus exploring $n.p$ nodes at most (where n is the number of input variables and p the number of terms). Thus, the overall asymptotic complexity of the algorithm is $O(n^2.p)$, when the tree is rotated n-times. If the tabular representation was used, merging and eliminating the rows involves a comparison of each pair of rows, which takes $O(n.p^2)$ steps. This is performed for n columns n-times, thus the overall complexity of the minimization algorithm is $O(n^2.p^2)$. The gain obtained by using the tree representation of the function is apparent now: the minimization method is advantageous especially for functions with many on-set terms, which it is targeted to.

B. Applying Absorption Rules

The minimization method described above performs merely two operations of Boolean algebra:

The one-variable absorption rule: $a + ab = a$ (1)
The complement property rule: $ab + ab' = a$ (2)

It is not able to disclose more complex relations, like absorption of more than one variable.

For this reason, an algorithm applying general absorption and negation absorption rules follows, to further reduce the tree size.

The algorithm recursively traverses the tree for each term, and checks if the term is absorbed by other terms. If so, the absorbed term is removed from the tree. The tree traversal is pruned for branches differing in a polarity of a variable (i.e., for non-intersecting terms).

The negation absorption rule ($a+a'b = a+b$) is being applied as well in the process. The tree is traversed in the same way as in the previous case, but the tree traversal continues even when (exactly) one conflicting variable is encountered. When a terminal node is reached by this way,

the negation absorption is detected and the absorbed variable is removed.

Applying the absorption rules involves comparing every term (terminal node) with all other terms. In the tabular representation of the set of terms, the complexity of this operation is $O(n.p^2)$, where n is the number of input variables and p the number of terms. The average complexity of the absorption detection is $\frac{1}{2}.n.p^2$. It is rather difficult to express the complexity of application of absorption rules upon the ternary tree explicitly, since it heavily depends on the tree structure. The worst case complexity is, of course, $O(n.p^2)$ as well. However, for dense trees, it is much less in practice.

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.

To make a comparison, we have implemented the absorption detection algorithm upon a standard tabular representation as well. The two approaches were tested on randomly generated PLAs of a given size. The results are shown in Table I. The *i* column gives the number of circuit's inputs, the *p* column the number of defined terms and the *idc* column the percentage of input don't cares (reflecting average dimensions of the terms). A significant speedup is achieved by using the ternary tree (TT) representation, especially for functions with many variables.

The measurement was performed on a PC with a 2GHz CPU.

Note that presenting any experimental comparison of the rotation operation runtimes for the tabular and ternary tree representations would be meaningless, since the rules (1) and (2) are applied in the general absorption detection procedure in the tabular representation implicitly.

TABLE I. COMPARISON OF THE ABSORPTION COMPLEXITY

Benchmark			*TT time [s]*	*Table time [s]*
i	*p*	*idc*		
20	1000	10%	0.55	1.35
20	2000	10%	1.24	4.56
20	5000	10%	3.86	25.05
20	10000	10%	9.21	167.06
20	20000	10%	22.43	1069.91
20	1000	25%	0.56	1.33
20	2000	25%	1.30	4.26
20	5000	25%	4.09	23.35
20	10000	25%	9.76	170.22
20	20000	25%	22.61	918.79
20	1000	50%	0.43	1.25
20	2000	50%	0.92	3.92
20	5000	50%	1.68	11.36
20	10000	50%	2.28	23.28
20	20000	50%	2.09	39.51

C. Incompletely Specified Functions

The proposed minimization algorithm is also capable of handling incompletely specified functions. For these, two ternary trees are constructed, one for the on-set (F), one for the don't-care set (D). The trees are processed separately, i.e., rotated, while operations (1) and (2) are performed for the leaves (see Subsection 4.2) and absorbed terms are removed. Then, the on-set and dc-set trees are merged together, while dc-terminals are distinguished. After that, dc-terminals are merged with on-terminals, if possible. For details of the algorithm, see the following subsection.

D. The Overall Minimization Algorithm

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

```
Minimize (F,D) {
   on_set = Create_TernaryTree(F);
   dc_set = Create_TernaryTree(D);

   for (i = 0; i < n; i++) {
      // n is the number of inputs
         on_set->Merge_Leaves();
         on_set->Rotate();

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

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

   fd_set = Merge(on_set, dc_set);
   fd_set = fd_set->Absorb_Terms();

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

Algorithm 1. The minimization algorithm

First, the two ternary trees are constructed from the PLA description of the source function. Then, the trees are being rotated and the terminals are being merged (as described in Subsection IV.A), which is done *n*-times (where *n* is the number of input variables). This procedure substantially reduces the size of the trees, before they are processed further. We have shown experimentally in [18] that rotating the tree more than *n*-times improves the result only slightly.

After the rotations, the general absorption procedure follows (see Subsection IV.B). There are several reasons for doing the absorption removal after the rotations, not inside of the rotation cycle, which could also be possible. First of all, performing the absorption detection inside of the cycle would significantly increase the algorithm time complexity, especially for dense trees (see Subsection V.B). Moreover, the rotation does not affect the absorption

detection procedure and vice versa. Thus, running this procedure repeatedly inside of the rotation cycle is meaningless.

At the end, the on-set and dc-set trees are merged and the absorption procedure is applied again. The reason for processing the on-set and DC-set separately is in preventing of merging on- and DC-terms, that will result in redundancy. For example, a consensus term coming from merging an on- and DC-term cannot be detected and removed from the final solution by algorithms used.

The algorithm may be run in an iterative mode, where the whole process is repeated several times. The solution quality may be significantly improved by this way, for the cost of an increased runtime. A more thorough analysis of the minimization process follows.

E. Analysis of the Minimization Process

Here we present an analysis of the influence of the two major phases (tree rotation and absorption rules application) on the final result quality and the runtime. The results are summarized in Table II. A collapsed ISCAS s420 benchmark circuit [20] was processed in three alternatives of the minimization algorithm. The processed PLA has 35 inputs and more than 113,000 product terms. First, only the tree rotation was performed, no general absorption was applied. Then, on the contrary, only the absorption was applied, without performing the tree rotation. After all, the complete algorithm, as it is proposed in Subsection 4.4, was run. All these three alternatives were run iteratively, until no change of the solution quality was observed.

It can be seen from the result, that applying both the tree rotation and absorption detection procedure is essential for obtaining a satisfactory result in a reasonable time. Then, the repeated application of the minimization algorithm has been proven to be beneficial. Performing the rotation prior to the absorption detection is essential for reducing the overall runtime.

TABLE II: ANALYSIS OF THE ALGORITHM STEPS

Iter.	Rotation only		Absorption only		Rotation + absorption	
	terms	*time [s]*	*terms*	*time [s]*	*terms*	*time [s]*
1	3180	4.07	59257	6.06	3180	4.28
2	1711	4.26	31368	8.62	1679	4.38
3	1407	4.39	14399	9.90	1394	4.68
4	1355	4.48	8402	10.15	1342	4.98
5	1330	4.62	4945	10.40	1282	5.01
6	1318	4.71	4053	10.63	1249	5.11
7	1315	4.82	2607	10.86	1249	5.23
8	1312	4.95	2061	11.08	1249	5.25
9	1309	5.04	1854	11.21		
10	1309	5.14	1651	11.48		
11			1604	11.61		
12			1559	11.89		
13			1558	12.02		
14			1550	12.34		
15			1511	12.62		

V. SOME MORE EXPERIMENTAL RESULTS

All the following experiments were run on a 2 GHz PC with 4 GB memory.

A. Comparison with Espresso

Here we present a comparison of the ternary tree based algorithm (TT-Min) with Espresso [7], for randomly generated benchmark problems, collapsed ISCAS benchmark circuits [19, 20], and SAT problems [22]. Single-output functions were used in all the experiments, since TT-min does not support a group minimization yet.

Since TT-Min definitely does not have the minimization strength as Espresso does, the TT-Min result quality is obviously worse. On the other hand, it is able to process extremely large PLAs very fast. It is able to minimize large PLAs "at least somehow", in cases where Espresso completely fails due to a prohibitively long runtime or high memory consumption.

Another important application area of TT-Min emerges here: when TT-Min is used for a pre-minimization, which is run prior to Espresso minimization, the total minimization time is reduced, whereas the result quality remains unchanged (in case of completely specified functions), or it is affected only slightly.

The results obtained by processing randomly generated circuits (PLAs) are shown in Table III. First the circuit parameters are given (i = number of inputs, p = number of terms, *idc* = percentage of don't cares in terms, *odc* = percentage of output don't cares). *idc* describes average dimensions of the terms, *odc* describes the percentage of don't care terms. After the PLA description, the results obtained by processing this PLA by TT-Min, Espresso, and a combination of TT-Min and Espresso follow. The numbers of product terms in the result and the minimization times are shown.

It can be seen that for these benchmark circuits TT-Min clearly outperforms Espresso in the runtime. Naturally, the quality of the result obtained by TT-min is worse. However, when TT-Min is used in combination with Espresso, the result is obtained in a shorter time, while the result quality is not affected that much.

For some circuits Espresso failed to produce any result at all, due to a prohibitively long runtime (more than 3 hours) or an excessively big memory consumption causing Espresso crash.

Results obtained from a minimization of ISCAS [19, 20] benchmark circuits are shown in Table IV. Here original multi-level networks were collapsed into their two-level representations. This was done by constructing separate BDDs for each output and for each BDD all paths from the root to the 1-terminal were traversed, to obtain respective PLAs.

The Table IV structure is partially retained from Table III. The benchmark circuit name with the respective output number identifies the processed single-output PLA.

The numbers of the benchmark inputs and defined terms follows, and then the TT-Min, Espresso and TT-Min + Espresso results are presented.

It can be seen that in some cases (c432) TT-Min returns the result faster than Espresso, but combining the two minimizers becomes counterproductive here, since the summary runtime becomes bigger than the Espresso runtime. On the contrary, the "s420_12" benchmark circuit illustrates the effectiveness of using TT-Min for the pre-minimization. An equal result is obtained in almost 18-times shorter time, compared to running Espresso only. For the "s1423" circuit TT-Min failed to overpower Espresso, even in the runtime, probably due to a big number of input variables (91).

As the last set of experiments, another practical minimization task was performed – solving the satisfiability (SAT) problem by converting the function's CNF (product of sums) description into a SOP, by directly rewriting the sum terms into product terms. The original CNF is unsatisfiable, if the resulting SOP is a tautology, which is detected by minimization of the SOP. A set of standard SAT DIMACS benchmarks [22] was processed in this experiment. The resulting SOP was processed by TT-Min, Espresso, and their combination. The results are shown in Table V. The original CNF size is shown first (# of inputs, # of clauses). After the conversion to the SOP, the number of product terms is equal to the number of clauses. Next, solution times for TT-Min, Espresso, and their combination are shown. The time savings of using the combination of the minimizers is shown in the last column. More than 500 SAT instances were processed altogether. The average time saving was approximately 7%. It never happened that that the runtime of the combination of the two minimizers was bigger than the runtime of sole Espresso.

A representative example of advantageousness of running TT-Min prior to the Espresso minimization is shown in Fig. 6. Here completely specified PLAs having constantly 20 inputs were processed by Espresso only and by TT-Min and Espresso in a sequence. The number of function's terms varied from 100 to 100,000. It can be seen that running TT-Min prior to Espresso brings time benefits. The obtained result qualities were almost equal (the maximum difference was less than 1%).

Advantages of running TT-Min prior to Espresso are clear from the figure; the more product terms the PLA has, the bigger the time difference of running Espresso alone and with the TT-Min preprocessing.

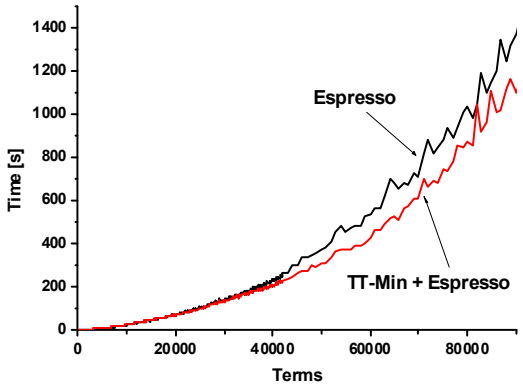

Figure 6. Runtime improvement w.r.t. running only Espresso

B. *Real Time Complexity*

The time complexity of the proposed minimizer was experimentally evaluated, for varying numbers of source function's inputs and defined terms. Any comparison with Espresso is irrelevant here, since the result quality obtained by Espresso is much better. On the other hand, the Espresso runtimes are in orders of magnitude bigger. For an illustrative example see Tables III and IV. The dependencies of the TT-Min runtime on the numbers of input variables and defined terms are shown in Fig. 7 and Fig. 8, respectively. The first experiment was performed on completely specified functions having 10,000 terms, the number of input variables varied from 10 to 300. For the second experiment, 20-variable completely specified functions with varying number of defined terms were processed. One iteration of the minimization algorithm was performed in the experiment.

It can be seen that the time complexity grows almost linearly with the number of defined terms (Fig. 8), while the runtime also does not rapidly grow with the number of inputs (Fig. 7). This is partially in contradiction with the theoretical asymptotic time complexity presented in Section IV. The overall asymptotic complexity of the minimization algorithm is $O(n.p^2 + n^2.p)$, in case of n tree rotations are used. These experimental results show that the real time complexity does not approach the asymptotic one, especially for the case of the number of terms. This is due to the fact that for dense trees (i.e., PLAs with many terms), the algorithms' complexities near to linear.

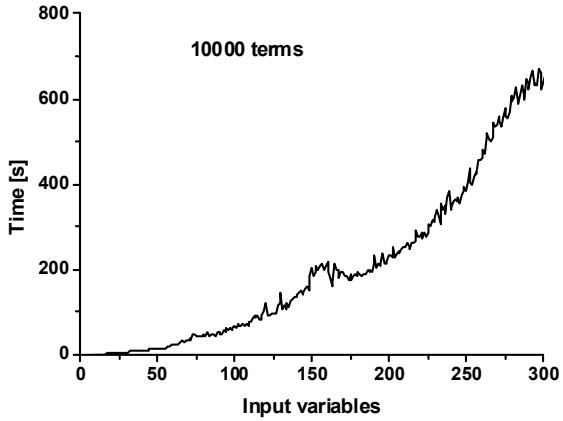

Figure 7. TT-Min runtime as a function of the number of input variables

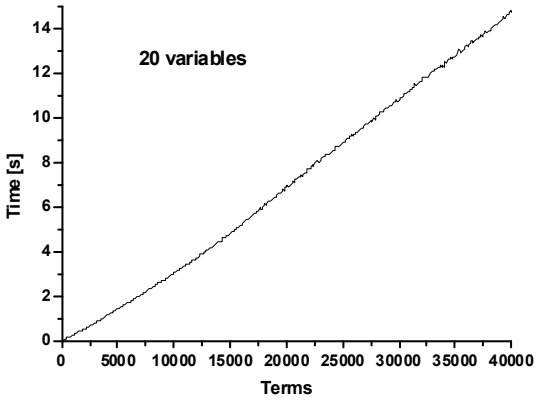

Figure 8. TT-Min runtime as a function of the number of defined terms

VI. Conclusions

An algorithm for a fast 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. Even though the proposed minimization algorithm is not able to outperform Espresso in quality of the result, it is much faster. Moreover, it can be used in cases where Espresso fails to return any result at all. The proposed minimizer has found another application area – when it is used for a pre-minimization run prior to Espresso, the result is obtained in much shorter time compared to running Espresso alone, while the result quality is not much affected.

As the future work, we expect implementing multi-output minimization support.

Since the ternary tree was found to be a really memory and time efficient representation of a set of terms, application of more complex minimization steps (e.g., those of Espresso) upon this representation should be considered as well.

Acknowledgment

This research has been supported by MSMT under research program MSM6840770014 and GA102/09/1668.

References

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

[2] Agarwal, Kime, Saluja: “A tutorial on BIST, part 1: Principles”, IEEE Design & Test of Computers, vol. 10, No.1 March 1993, pp.73-83, part 2: Applications, No.2 June 1993, pp.69-77

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

[4] 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

[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 H. Kubátová, “Flexible Two-Level Boolean Minimizer BOOM II and Its Applications”, Proc. 9th Euromicro Conference on Digital Systems Design (DSD'06), Cavtat, (Croatia), 30.8. - 1.9.2006, pp. 369-376

[12] 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

[13] P. Fišer, P. Kubalík, and H. Kubátová, “An Efficient Multiple-Parity Generator Design for On-Line Testing on FPGA”, Proc. 11th Euromicro Conference on Digital Systems Design (DSD'08), Parma (Italy), 3. - 5.9.2008, pp. 94-99

[14] 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

[15] 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

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

[17] 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

[18] 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

[19] 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

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

[21] O. Coudert, "Doing two-level logic minimization 100 times faster", Proc. of the sixth annual ACM-SIAM symposium on Discrete algorithms, 1995, pp.112-121

[22] http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

TABLE III: RANDOMLY GENERATED CIRCUITS

Benchmark				*TT-Min*		*Espresso*		*TT-min + Espresso*	
i	*p*	*idc*	*odc*	*terms*	*time [s]*	*terms*	*time [s]*	*terms*	*time [s]*
20	10,000	0%	0%	6,505	19.80	60,331	1834.84	60,454	1600.14
20	50,000	0%	20%	40,386	92.39	30,062	1482.38	30,059	1297.34
20	16,000	30%	0%	14,945	9.89	9,247	680.94	9,294	474.84
20	10,000	30%	30%	6,669	5.34	6,034	120.82	6,495	73.42
20	20,000	30%	30%	13,012	14.36	6,163	549.66	10,078	449.25
20	20,000	25%	0%	18,933	22.61	16,361	949.08	16,366	1004.18
20	100,000	20%	0%	86,128	159.30	N/A	> 3 h	N/A	> 3 h
20	2,000	65%	0%	1	0.04	1	0.07	1	0.04
20	2,300	50%	0%	1,973	0.44	510	121.00	510	110.10
20	2,400	50%	0%	1,901	0.47	N/A	N/A	N/A	N/A
23	50,000	50%	30%	32,807	152.82	N/A	N/A	N/A	N/A
23	100,000	50%	30%	40,834	171.56	N/A	N/A	N/A	N/A
23	200,000	50%	30%	22,058	123.36	N/A	N/A	N/A	N/A
30	200	70%	20%	160	0.12	160	1799.19	160	4849.20
30	300	70%	20%	240	0.20	239	51875.30	239	12401.80
32	100	70%	20%	79	0.22	79	1100.48	79	254.98

TABLE IV: COLLAPSED ISCAS BENCHMARKS

Benchmark			*TT-Min*		*Espresso*		*TT-min + Espresso*	
name	*terms*	*inputs*	*terms*	*time [s]*	*terms*	*time [s]*	*terms*	*time [s]*
c432_1	20127	36	12863	6.8	2313	8.7	2313	16.7
c432_3	42092	36	41716	23.7	64	50.0	64	115.76
s420_12	**113280**	**35**	**1342**	**5.47**	**17**	**97.02**	**17**	**5.53**
s1423_32	13584	91	6202	13.3	424	1.83	424	14.4
s1423_33	23143	91	10554	25.51	425	4.59	425	27.29
s1423_49	188729	91	48392	231.58	3980	98.72	3980	262.80
s1423_50	154112	91	38913	158.89	3188	64.88	3188	179.58

TABLE V. SAT RESULTS

n/c	TT-Min [s]	Espresso [s]	TT-Min+Espresso [s]	Improvement
50/218 (1)	0.08	50.40	40.84	19.0%
50/218 (2)	0.09	30.19	29.28	3.2%
50/218 (3)	0.09	36.13	34.54	4.4%
50/218 (10)	0.08	77.84	65.48	15.9%
75/325 (1)	0.30	22601.6	22029.2	2.5%
75/325 (10)	0.30	21554.1	18715.0	13%