A SET OF LOGIC DESIGN BENCHMARKS

Petr Fišer, Jan Hlavička
Department of Computer Science and Engineering
Czech Technical University
Karlovo nám. 13, 121 35 Prague 2
e-mail: fiserp@fel.cvut.cz, hlavicka@fel.cvut.cz

Abstract. *The paper presents a new set of logic design benchmarks, whose purpose is to evaluate the capabilities of very powerful design tools. Their input format is compatible with the MCNC benchmarks, i.e., a PLA matrix. They are defined by the on-set and off-set listed in a truth table, whereas the don't care set often representing the dominant part of the truth table, is not specified. The set includes a wide range of problem sizes measured by different criteria. The benchmarks contain randomly generated values, which guarantees stable statistical properties and opens the possibility of further extension.*

1 Introduction

Several sets of logic synthesis benchmarks have been proposed with the goal to evaluate the performance of two-level Boolean minimizers [1], logic function manipulators and other design tools. Best known are the MCNC benchmarks [6], ISCAS [2, 3, 9] and ITC benchmarks [7]. A common feature of these benchmarks is the relatively small size of the problems. Thus, these benchmarks are hardly adequate for testing minimizers capable of handling problems of large dimensions, like e.g. BOOM [4, 5, 9]. Moreover, the two-level MCNC benchmarks are specified in a PLA format, where together with the on-set all the don't care (DC) terms are explicitly listed and the off-set is computed as a complement of the union of the two listed sets. However, this specification is often unacceptable in practice, because larger functions mostly have an extremely high number of don't care states. A more natural specification of such a function is an explicit definition of the on-set and off-set, while the don't care terms need not be specified. Such large-size problems are encountered, e.g., in the design of built-in self-test (BIST) devices for VLSI circuits, in the design of control systems and many other areas of logic design.

For synthetic benchmarks it is not difficult to grow with the performance of the tools under test. For the natural benchmarks cited above, such a growth is not possible. Therefore we decided to generate a set of synthetic benchmarks for logic synthesis. Their most important features are

- input format compatible with the MCNC benchmarks
- wide range of problem sizes measured by a variety of criteria
- capability of further extension
- randomly generated values, which guarantee stable statistical properties.

As these benchmarks were designed in connection with the development of the BOOM minimization system, we have given them the name "*BOOM Benchmarks*".

The paper has the following structure. First the benchmark format is described, then the method of generating them is presented. Finally some results obtained from comparison runs of two minimizers are listed and commented.

2 BOOM Benchmarks Description

2.1 Benchmark Parameters

All BOOM benchmarks are generated by a random number generator for which we specify the numbers of input and output variables, the number of care terms and the percentages of don't cares in the input and output matrices. The occurrence of DCs in the input matrix signals that it contains terms of a dimension greater than zero. A don't care state in the output matrix indicates that the corresponding term has no meaning for the output function.

The current BOOM benchmark set consists of two sets of 5-output functions where the input matrix contains 20 % and 50 % of don't cares, respectively. No don't care states are explicitly marked in the output matrix. The number of input variables and care terms varies from 50 to 300 with a step of 50, which yields 36 different benchmark sizes. For each problem size and each don't care percentage ten samples were generated. Thus, altogether 720 benchmark files were prepared.

The filenames of BOOM benchmarks uniquely describe the size and nature of a problem. They are structured as follows:

`bb_`*`iv`*` x `*`ov`*` x `*`p_d`*`%_`*`n`*`.PLA`

where *iv* is the number of input variables, *ov* is the number of output variables, *p* is the number of care terms, *d* is the percentage of don't cares in the input matrix and *n* is the problem number (0-9). For example, a benchmark named `bb_100x5x200_20%_0.pla` has 100 input variables, 5 output variables, 200 care terms and 20 % of don't cares in the input matrix. All BOOM benchmarks are available for public use at [10].

2.2 Function Consistency

One of the main problems we encounter when generating random truth tables describing the on-set and off-set of the function is the problem of inconsistency of the function. A single-output function is inconsistent, if there is a non-empty intersection of the on-set and the off-set, i.e., its on-set and off-set are not orthogonal. A multi-output function is inconsistent, if one or more of its output functions are inconsistent. The behavior of an inconsistent function is non-deterministic and it cannot be implemented in hardware. Thus, we must prevent generating inconsistent functions as benchmark problems. The inconsistencies can be detected by a simple algorithm: we find all the intersecting term pairs and verify whether all their output values are identical. If not, the inconsistency is detected and removed. Obviously, the higher is the dimension of the terms, the more term pairs are likely to intersect. This causes problems when generating truth tables with a high percentage of don't cares in the input matrix and a high number of care terms. For such problems a naive random generation of the matrices is impossible. We have developed a method allowing us to produce truth tables with more input don't cares. In the algorithm first the function with *k*-times more terms than needed (*p*) is randomly generated. All the inconsistent terms are then removed, which obviously reduces the total number of terms. If fewer terms than *p* are left, further random terms are added and the procedure is repeated. Otherwise the function is trimmed to have the required number of terms.

For a high percentage of don't cares and a high number of care terms the value of *k* must be increased in order to find a solution in a reasonable time. Table 1 shows approximate maximum reachable percentages of don't cares in the input matrix for various problem sizes (*p* is the number of care terms, *n* is the number of input variables, the number

of output variables is set to 5). In this case, $k = 1$, thus no excessive terms are produced during the function generation.

Table 1. Maximum DC percentages for various problem sizes for $k = 1$

p / n	50	100	200	300
50	57 %	68 %	77 %	82 %
100	52 %	65 %	74 %	79 %
200	45 %	60 %	72 %	76 %
500	39 %	55 %	68 %	74 %

When *k* grows, terms with higher *d* can be produced. Table 2 shows the maximum reachable values of *d* for $n = 20$, $p = 500$ and different values of *k*.

Table 2. Maximum DC percentages for varying *k*

k	% of don't cares
1	12 %
2	28 %
3	30 %
4	31 %

2.3 Statistical Properties of the Benchmarks

As described above, the truth tables with don't cares are created by repeated addition and removal of function terms. We should ensure that at the end a matrix with the required percentage of don't cares is produced. In our algorithm the terms are created literal by literal, while it is decided each time with a given probability *d* whether the literal will be a don't care or not. Then, the resulting terms will contain *d* percent of don't cares in the average. The frequency rate of occurrence of individual percentages of don't cares in terms for a BOOM benchmark `bb_50x5x300_20%_1` is shown in Figure 1. We can see that the percentage rates approximately correspond to the normal distribution with the mean value 20.05 %, which is almost equal to the required don't care percentage.

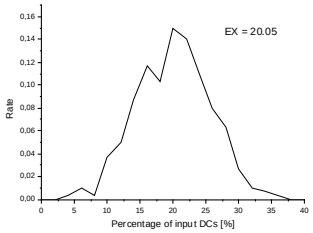

Figure 1. DC rates for a sample BOOM benchmark

3 Experimental Results of the Benchmarks

BOOM benchmarks were extensively tested on two Boolean minimizers – namely ESPRESSO [8] and BOOM [9] and the minimality of the results was compared. BOOM was always run iteratively, using the same total runtime as ESPRESSO needed to obtain a solution. The quality criterion selected for BOOM was the sum of the number of literals and the output cost.

In Table 3 the number of input variables *n* is increased horizontally and the number of care terms *p* is increased vertically. The first row of each cell contains the BOOM results,

and the second row shows the ESPRESSO results. All the values are the averages of ten benchmarks of the same size.

The results of the comparison are listed in the white portion of Tab. 3. We can see that in all but one problem size (150x200) BOOM found a better solution than ESPRESSO. The shaded area presents the results of problems that were not solved by ESPRESSO because the runtime was too long. Thus only the BOOM results obtained in one iteration are listed.

Table 3. Solving the BOOM benchmark problems by two minimizers - equal time

p / n	50	100	150	200	250	300
50	110/41/25 (58) 122/54/27/3.89	96/35/23 (90) 104/45/23/10.29	90/32/21 (147) 92/41/21/24.87	84/29/20 (199) 89/39/20/41.99	112/26/26/0.16	106/24/24/0.20
100	284/86/52 (46) 289/104/51/19.31	229/68/42 (94) 231/84/42/77.07	217/61/40 (140) 213/80/39/199.17	207/57/38 (140) 201/74/37/246.21	262/47/47/0.51	264/48/48/0.63
150	474/132/76 (43) 481/158/76/54.76	389/101/63 (101) 384/125/62/282.80	362/92/61 (116) 345/113/56/646.20	381/90/64 (64) 322/107/52/1066.14	453/73/73/1.01	441/72/72/1.27
200	678/177/101 (51) 686/209/101/162.62	553/137/83 (116) 539/165/81/730.91	492/125/75 (207) 480/149/72/1913.65	469/110/71 (277) 450/136/68/3372.66	656/98/98/1.87	625/94/93/2.15
250	1508/233/211/1.25	1163/167/164/1.66	1003/142/141/2.04	935/133/132/2.56	871/123/123/2.89	844/119/119/3.26
300	1938/291/258/1.87	1465/200/197/2.37	1260/172/171/2.93	1161/157/157/3.53	1104/150/150/4.18	1049/143/142/4.71

Entry format: *BOOM:* *#of literals / output cost / #of implicants (# of iterations)*
ESPRESSO: *#of literals / output cost / #of implicants / time in seconds*
Shaded area: *BOOM results: #of literals / output cost / #of implicants/ time in seconds*

4 Conclusions

A new benchmark set named “BOOM Benchmarks” was proposed together with a benchmark generation method. The benchmarks are primarily aimed at testing two-level Boolean minimization tools. However, they can also be used for testing, e.g., functional decomposition tools and others. The benchmarks are presented in the standard Berkeley PLA format, where only the on-set and off-set of the functions is specified. This allows us to describe logic functions with a large portion of don’t care states.

The current benchmark set consists of 720 problems of 36 different sizes (20 problems of each size, 10 with d = 20 % and 10 with d = 50 %). These benchmarks were tested on two Boolean two-level logic minimizers: BOOM and ESPRESSO. The quality of the results and the minimization runtime for each benchmark were compared.

References

[1] Brayton, R.K. et al.: Logic minimization algorithms for VLSI synthesis. Boston, MA, Kluwer Academic Publishers, 1984

[2] Brglez, F., Fujiwara, H.: A Neutral Netlist of 10 Combinational Benchmark Designs and a Special Translator in Fortran, Proc. Intl. Symp. on Circuits and Systems, Special Session on ATPG and Fault Simulation, June 1985

[3] Brglez, F. et al.: Combinational Profiles of Sequential Benchmark Circuits, Intl. Symp. on Circuits and Systems, May 1989, pp. 1929-1934

[4] Fišer, P. - Hlavička, J.: 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

[5] Hlavička, J., Fišer, P.: BOOM - a Heuristic Boolean Minimizer. Proc. International Conference on Computer-Aided Design ICCAD 2001, San Jose, California (USA), 4.-8.11.2001, pp. 439-442

[6] http://www.cbl.ncsu.edu/CBL_Docs/Bench.html

[7] http://www.cerc.utexas.edu/itc99-benchmarks/bench.html

[8] http://eda.seodu.co.kr/~chang/download/espresso/

[9] http://service.felk.cvut.cz/vlsi/prj/BOOM/

[10] http://service.felk.cvut.cz/vlsi/prj/BoomBench/