SINGLE-LEVEL PARTITIONING SUPPORT IN BOOM-II

Petr Fišer, Hana Kubátová

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

Abstract: We propose a modification of our Boolean minimizer BOOM-II enabling a single-level partitioning. The disadvantage of all the present logic synthesis systems is that the minimization and decomposition phases are strictly separated; the minimization process is independent on the subsequent decomposition. We propose a method where the two-level minimization is driven by some decomposition or other constraints. Here a two-level nature of a solution is retained, however, the circuit is divided into several stand-alone blocks, each block having several outputs. Our aim is to minimize the number of inputs for each block, as well as the blocks' logic.

Keywords: Boolean functions, minimization, decomposition, logic design

1. INTRODUCTION

The Boolean minimization is an essential process in many phases in logic synthesis [Hassoun and Sasao, 2002]. Many two-level Boolean minimizers were developed so far, originating from the Quine-McCluskey's algorithm. Latter heuristic algorithms, like ESPRESSO [Brayton, *et al.*, 1984] with its improved versions [McGeer, *et al.*, 1993] were developed. They are capable handling relatively large Boolean functions in a reasonable time, for a price of a non-minimal solution.

Lately we have developed a two-level heuristic Boolean minimizer BOOM [Hlavička and Fišer, 2001], [Fišer and Hlavička, 2003a]. This minimizer is capable to deal with functions with a large number of input variables (up to thousands) in a very short time. One disadvantage of BOOM is its relatively long runtime for functions with many outputs. Hence, we have developed FC-Min in succession [Fišer, Hlavička and Kubátová, 2003b]. It is suitable for problems with a large number of output variables, however, for low-output functions it often fails to produce good results. To make a universal minimizer, efficient for problems of all dimensions, we have combined these two methods into BOOM-II. Here the two algorithms can be used simultaneously, their ratio being adjusted according the nature of the source problem.

As a result of the standard minimization using BOOM-II we obtain a two-level implementation of the minimized circuit, particularly a set of sum-of-product (SOP) forms, one for each output. To implement such a circuit in hardware, doing further decomposition into a multi-level network is necessary, since many-input gates, occurring is the SOP forms, often cannot be realized. Moreover, the decomposition often significantly reduces the resulting logic.

In a logic design flow process the Boolean minimization is often being conducted independently on decomposition and technology mapping phases; the aim of the two-level minimization algorithm is to reduce the number of literals or products in the resulting SOP (sum-of-product) form to minimum. However, this measure often does not represent the real complexity of the circuit, since it can be approximated not before the decomposition is done. Hence, the minimization and decomposition phases should be linked somehow, preferably the minimization should be driven by the decomposition and technology mapping constraints.

In this paper we propose an extension of BOOM-II allowing the minimization to be driven by some constraints. For decomposition purposes, the resulting circuit is divided into several stand-alone blocks. Particularly, we try to decompose the solution into a given number of blocks, while keeping the

number of the inputs entering each block minimal, and, if possible, maximally disjoint.

Any other constraints can be applied to the minimization process. Namely, in the DFT (design for testability), the size of the output cones (i.e., the number of inputs influencing one output) can be reduced by applying the specific constraints. The minimization process can be influenced to produce circuits with a balanced input load, and similarly.

The paper is structured as follows: after the Introduction the principles of the single-level partitioning are given. The structure of BOOM-II is briefly described in Section 3. Section 4 describes the necessary modifications of BOOM-II algorithms needed to support the constraint-driven minimization. The experimental results are presented in Section 5, Section 6 concludes the paper.

2. SINGLE-LEVEL PARTITIONING

As a result of a two-level Boolean minimization we obtain a circuit consisting of an AND-plane generating the product terms and an OR-plane summing the products to obtain the output values. When using a group minimization, we try to share the products among the outputs.

Unfortunately, such a two-level netlist often is very difficult to implement in hardware. If the target technology is, e.g., a PLA, or FPGA, the number of both its inputs and outputs is limited. Thus, some kind of decomposition has to be performed.

The single-level partitioning concept is based on dividing the resulting circuit into a given number of blocks, so that their two-level nature is retained. The blocks share the primary inputs, each block generates several outputs. The products cannot be shared among the blocks. This is illustrated by Figure 1. Here a logic function of 7 inputs x_1-x_7 and 6 outputs y_1-y_6 is decomposed into two 5-input and 3-output blocks while each block is a two-level (AND-OR) circuit.

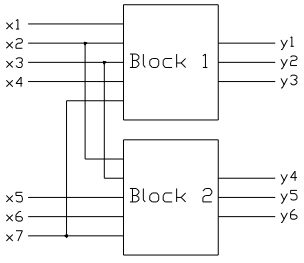

Fig. 1. A single-level partitioning

In our partitioning-based minimization method we try to reduce the number of inputs entering the blocks as well. The method is based on a fact that a function, can be implemented in many ways. Each two-level SOP form of a given function consists of a set of essential products, which have to be present in every representation of the function [Hassoun and Sasao, 2002], and a set of implicants that could vary, as long as they together cover the whole on-set. Most of Boolean functions we encounter in praxis have only a few essentials, often none. Thus, we are able to select its implicants (products) with a big freedom. Moreover (and most importantly), in many cases not all the function's inputs are needed to produce a particular output.

Definition. A *support* of a single-output function is a set of its input variables needed to represent the function. The *minimum* support of a function is its support with the minimal cardinality. The definition can be easily extended to multi-output functions.

Example. Let us consider a 5-input single-output Boolean function described by a truth table, where the on-set (1) and off-set (0) of the function is defined. The minterms not listed in the table are assigned as don't cares. Let be the input variables named x_0 - x_4.

Table 1. The example of the support

x_0	x_1	x_2	x_3	x_4	y
1	1	1	0	0	1
1	0	1	0	0	1
1	0	0	0	0	1
0	1	0	1	0	0
1	0	1	0	1	0

The minimum support of the function is the set {x_0, x_4}, since only these two input variables are needed to distinguish between the 0 and 1 output values. The inputs $x_1 - x_3$ are not necessary to use to implement the function. When performing the single-level partitioning, our goal is to construct a support S of an *n*-input function, while |S| < n, generally, |S| should be as small as possible, or at least should not exceed a given limit. In most cases, reducing the support of a function yields worse minimization results (in terms of the complexity of the resulting circuit), since we decrease the amount of information on the function. Thus, some kind of trade-off has to be found here.

The single-level partitioning minimization process consists of two major issues: deciding how to assign the outputs of the multi-output function to the given blocks (i.e., how to group the outputs) and how to find the supports of the blocks. These issues will be addressed in Section 4.

3. BOOM-II

BOOM-II had come in succession to BOOM, as a combination of an original BOOM and the FC-Min minimizer. It combines two antipodal approaches to the Boolean minimization. The major part of BOOM is a CD-Search algorithm, where the implicants of

each single function are being generated. The basis of FC-Min is a Find-Coverage procedure, where the group implicants are being produced directly. Both these algorithms can be executed in an iterative way; the implicant generation process is run several times, while all the implicants are being gathered together. After that, a covering problem is solved using all the implicants and the irredundant cover of the source function is computed using them.

In BOOM-II the runs of the two algorithms are being alternated. At the beginning of each iteration it is decided which algorithm should be run to generate a new set of implicants. A probability of running each particular algorithm can be freely adjusted, according to the nature of the source function. The implicants obtained from the two methods are being put together, after several iterations the minimization is stopped and the CP is solved. The flowchart of BOOM-II is shown in Fig. 2.

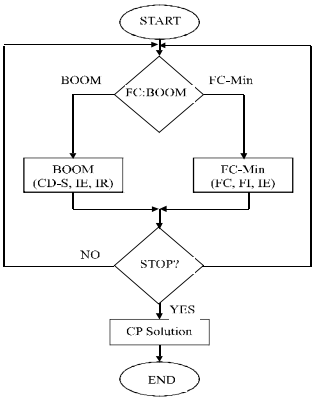

Fig. 2. Flowchart of BOOM-II

3.1. Principles of BOOM

The BOOM minimizer consists of several successive phases. At the beginning the multi-output function is split into single-output functions. For each function a set of implicants covering the whole on-set is produced in the CD-Search phase. These implicants are then expanded into prime implicants and then reduced to obtain group implicants, i.e., implicants of more than one function. After that the covering problem is solved and an output reduction is performed. The important phases will be described here briefly, for more detailed description see [Fišer and Hlavička 2003a].

The Coverage-Directed Search (CD-Search). This is the main and most contributive part of the BOOM algorithm. It generates an irredundant set of implicants covering the on-set of a single-output function. Unlike the other Boolean minimization methods (ESPRESSO) the implicants are being constructed top-down, i.e., by reducing a universal hypercube until it becomes an implicant. It does not start with the source implicants – the algorithm uses them just as guidance.

At the beginning the literal occurring in the on-set most frequently is found. Such a literal forms an *n-1* dimensional cube (for an *n*-input function) describing the half of the Boolean space containing the majority of the on-set (maximum of 1s). We compare this cube with the off-set to find out whether it is an implicant of the function, i.e., whether it does not intersect the off-set. If it is not an implicant, we search for the second most frequent literal and add it to the previous one. Again, we check if it is an implicant and repeat the process. When an implicant is generated, we remove the on-set terms that are covered by it and repeat the whole process until the whole on-set is covered.

The Implicant Expansion (IE) Phase. The CD-Search algorithm is a greedy heuristic and the implicants need not be prime (PI). Thus, they should be expanded to reduce the number of literals in these terms. Several IE methods were proposed, all of them are based on a simple removal of literals from all the terms.

The Implicant Reduction (IR) Phase. Here the PIs are being reduced into group implicants. This phase is similar to the CD-Search. Literals are being added to the present terms, so that the term becomes an implicant of the maximum number of output functions.

3.2. FC-Min Principles

The FC-Min minimizer has been developed to efficiently handle functions with a large number of output variables. Here the minimization is being conducted in a reverse way than the standard minimizers do. First, the cover of the on-set is found, independently on the source implicants. After that the minimized implicants are produced by joining the source implicants. This process is directed towards satisfying the cover. After that the implicants are expanded to reduce the number of literals.

We will briefly describe these two phases, for more information, see [Fišer, Hlavička and Kubátová, 2003b].

The Find Coverage Phase. It is an essential phase of the FC-Min algorithm. Here the whole cover of the on-set of the multi-output function is found, using the output part of the source function only. An example of such a cover is shown in Fig. 3. There is a 5-input and 5-output function defined by 10 terms. The rest are assigned as don't cares. The result of the Find Coverage algorithm is a cover consisting of six terms, $t_1 - t_6$. Each element in this cover describes properties of an implicant. For example, t_1 must be an implicant of y_3 and y_4, and cover the ones in the 4^{th}, 6^{th} and 8^{th} row. To solve the coverage finding problem we use a greedy heuristic as well, since it is NP.

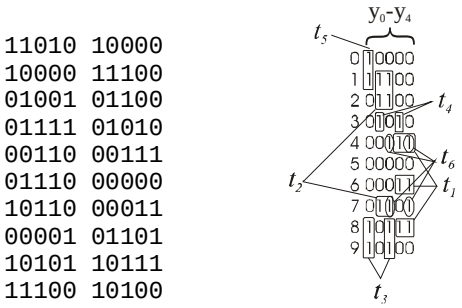

Fig. 3. Cover of the output matrix

Implicant Generation Phase. In this phase we generate the implicants from the cover. Considering the conditions described above, particularly the definition of the rows (vectors) each cover element should cover, a simple rule for the implicants can be derived: the *minimum implicant* satisfying the particular cover can be constructed as a minimum *supercube* of all the input vectors corresponding to the rows of the cover of t_i. Moreover, this supercube must not intersect any term that is not included in the particular cover, since it would cover some zeros then. In our example, a minimum implicant t_1 would be (-01--), since

```
00110
10110
10101
-01--
```

Implicant Expansion Phase. The Implicant generation phase produces the minimal implicants, thus the satisfactory implicants having the maximum of literals. They can be further expanded to reduce the number of literals.

3.3. Covering Problem Solution

After the implicants are generated, the covering problem has to be solved to obtain an irredundant cover of the on-set. Solving it exactly is mostly impossible, since the number of implicants is often large. Thus, we use a scoring function based greedy incremental heuristic [Coudert, 1994].

We construct a covering matrix *A*, its dimensions will be (*r*, *s*). Its columns correspond to the implicants, rows to the individual on-set terms that have to be covered. We set *A[i, j]* = 1 if the implicant *j* covers the on-set term *i*, *A[i, j]* = 0 otherwise. For each row its *strength of coverage* is computed as

$$\mathrm{SC}(x_i) = \frac{1}{\sum_{j=1}^{s} A[i, j]} \qquad (1)$$

Then the *column contribution* is computed for each column:

$$\mathrm{CC}(y_j) = \sum_{i=1}^{r} A[i, j] \cdot SC(x_i) \qquad (2)$$

After that the implicant (column) with the maximum contribution value is selected into the solution, the contribution values are recomputed and the process is repeated until the whole on-set is covered.

The Output Reduction consists in solving *m* separate covering problems for an *m*-output function. In this phase the number of terms cannot be reduced, however it determines the irredundant set of implicants for each function, and thus removes redundant connections between the AND and OR planes in the resulting two-level network.

4. THE CONSTRAINT-DRIVEN MINIMIZATION

In this section we will describe modifications of the previously described algorithms, allowing us to influence the minimization result in a desired way.

First, the multi-output function has to be divided into stand-alone blocks, or at least the set of its outputs has to be partitioned somehow. Then, the modified algorithm is run for each block.

4.1. The Types of Constraints

The minimization constraints need not be only the partitioning demands. We will briefly describe some of them:

Partitioning Constraints. In order to combine the minimization process with a partitioning, we split the source circuit by its outputs. Further, we try to keep the support of each block (i.e., the number of input variables in the minimized function) minimal.

DFT Constraints. To synthesize an easily testable circuit, we try to reduce the sizes of the cones in the minimized circuit. Thus, we try to minimize the support of each output variable separately.

Load Balancing. The low-power design is becoming more and more important nowadays. In some cases it is desirable to design circuits with a balanced load of its inputs. This means that the number of branchings of the circuit's inputs should be kept balanced. Moreover, for the low-power design, the number of branchings should be kept minimal. This condition may be contrary to the partitioning constraints, thus some trade-off has to be found.

4.2. Modification of BOOM Algorithm

CD-Search. This is the essential phase that has to be modified. The algorithm is based on a gradual addition of literals into the terms. The candidate literals are being selected using a scoring function; originally it was the frequency of occurrence. Thus, it is easy to modify this scoring function to manifest the constraints.

For the partitioning purposes we modify the scoring function, so that the frequency of the literal that is already included in the processed block is multiplied by the CD-Search partitioning force PF_{CD}, and is thus

preferred to other literals. The higher this force is, the smaller is the number of input variables entering the blocks. For DFT, further modification is very similar to the previous one: we prefer input variables that are already included in the current partial SOP form of the currently processed variable. When applying the load balancing, we penalize variables entering the other blocks.

Implicant Expansion. In this phase the literals are being removed from the terms. It could be modified to adopt some constraints as well, e.g., by preferring a removal of the literal that would yield a reduction of the number of inputs entering the block (for partitioning). In praxis we do not do it, since the expansion phase produces several PIs from one non-PI, and the "advantageous" implicants are being produced anyway.

Implicant Reduction. Here, as well, many group implicants are being produced from one PI. Modifying the scoring function defining the candidate literals for inclusion is possible, however we have found that the effect of this modification is negligible.

For a more thorough description of the partitioning-based BOOM method see [Fišer and Hlavička, 2002]

4.3. Modification of FC-Min

Find-Coverage. Since this phase does not directly influence the selection of what literals would be included in the solution, its modification would be meaningless. However, it strictly defines what terms would be shared among what output variables – it defines the group implicants. Therefore, it determines what outputs would be grouped together in the final solution.

Until now we haven't described the way how we group the outputs into the blocks. This decision could be made at random (as it was being done in BOOM), or we can exploit the Find Coverage phase to make the partitioning.

Before the whole modified minimization is run, we run the FC-Min for the original circuit and determine from the result, which outputs share the maximum of implicants. These outputs are then grouped together into one block. Then we remove these outputs from the source function and repeat the whole process, until all the outputs belong to some block. The number of outputs to be grouped is determined by the size of each block (which is customizable).

Implicant Generation. This phase is fully deterministic and cannot be influenced in any way.

Implicant Expansion. In this phase the number of literals in the final set of SOP forms is being significantly reduced, by up to 70%. Thus, here we can decide what literals will be included in the solution. For the partitioning based minimization, literals of input variables that are not included in the currently processed block are tried for removal at first, and only when no such a removal is possible, literals of variables that are entering the processed block are tried for removal.

For the load-balancing and minimization, literals of variables included in other blocks are removed first, then the rest. In a DFT design, we remove literals that are not included in current SOP forms of output variables, for which the currently processed term is an implicant.

4.4. Covering Problem Solution

Modifying the CP solution algorithm is of a key importance to reach good results. Consider that the CP solver selects only a small number of implicants from a huge implicant pool and constructs the final solution. Thus, if the algorithm was not modified, it could spoil all the effort of the previous phases.

In fact, any CP solver can be used, where only one condition has to be fulfilled: it has to be a greedy additive heuristic, i.e., the implicants have to be added to the solution one by one. Modifying an exact solver could also be possible, however it would extremely complicate the construction of a cost function here.

The CP is solved for each block individually, so that we prevent sharing the implicants among the blocks.

We will consider the CP algorithm described in Subsection 3.3. To apply the partitioning, additional weights are assigned to the implicants, thus the weights modify the contributions. Input variables used in particular blocks are recorded during the process. The weights of the implicants are proportional to the number of new input variables they would add to the currently processed block if they were selected into the solution. The more input variables is newly added into a given block by a term, the less likely this term will be selected.

In particular, the weight is being multiplied by a customizable PF_{COV} factor and the cost of the term divided by this value.

For the load minimization purposes the weights can be modified so that implicants containing inputs entering other blocks will be penalized.

5. EXPERIMENTAL RESULTS

To illustrate the effects of the partitioning and load minimization supports, we have processed a randomly generated function having 50 input variables, 40 output variables and 150 defined terms. Functions of such dimensions occur, i.e., in a design of control systems, BIST design (Fišer, Hlavička and Kubátová, 2003c). Our aim was to minimize the circuit and implement it using four 10-output blocks.

First, we have run BOOM-II without any modification; only the original circuit was randomly divided into 4 10-output blocks. As a result, we have

obtained four two-level circuits; their summary complexity was equal to 2375 gate equivalents (De Micheli, 1994). Each of the primary inputs was used (i.e., the support = 50), and each of them was entering each of the four blocks.

After that, we have determined a proper decomposition of the circuit by running FC-Min and applied the partitioning forces to the CD-Search and Covering problem solution. The FC-Min Implicant expansion phase was modified to support the partitioning as well. The results of the minimization are shown in Table 2, together with the results of the unmodified algorithm. The PF_{CD} and PF_{COV} forces were both set to 1.

Table 2. The experimental results

	no partitioning	with partitioning
total support	200	114
used inputs	50	47
branching inputs	50	36
branchings	150	67
maximum load	4	4
average load	4	2.34
GEs	2375.5	2874.5
block 0 inputs	50	30
block 1 inputs	50	30
block 2 inputs	50	26
block 3 inputs	50	28

The "total support" row shows the sum of the supports of all the four blocks, i.e., the total number of wires entering the blocks. A great reduction can be seen here, when the partitioning is applied. Without the partitioning all the inputs enter all the blocks (4 x 50 = 200). With partitioning, this value is reduced to nearly one half.

The number of input variables used is reduced as well. When the partitioning was applied, 3 inputs were found to be not needed to interpret the function.

Even when no load minimization was applied here, the simple partitioning reduces it as well. The number of branchings was reduced from 150 to 67, while the average load of the inputs was reduced from 4 to 2.34.

As the most important fact we can observe that the number of inputs of each block was significantly reduced – from the total 50 to at most 30. Thus, after the partitioning is applied we are able to construct the whole circuit using four 30-input and 10-output stand-alone blocks, which was not possible before.

The increase of the area is only 17%, which is acceptable in most cases (see the GEs row).

6. CONCLUSIONS

We have presented principles of a constraint-based minimization in BOOM-II, with emphasis on a partitioning of the circuit. Since BOOM-II is a very complex system, many changes had to be done. A good partitioning scheme can be obtained using the FC-Min algorithm, which is a part of BOOM-II. Then, using the BOOM-II algorithms modified to support the partitioning, the circuit can be efficiently divided into several stand-alone blocks, directly in the two-level minimization process.

The principles can be extended for a low-power design, design-for-testability, and many other constraint-driven designs.

ACKNOWLEDGEMENT

This research has been in part supported by the GA102/04/2137 grant and MSM 212300014.

REFERENCES

Brayton, R.K., et al. (1984). *Logic minimization algorithms for VLSI synthesis*, 192 pp., Boston, MA, Kluwer Academic Publishers.

Coudert, O. (1994). Two-level logic minimization: an overview, *Integration, the VLSI journal*, **Vol. 17-2**, pp. 97-140

De Micheli, G. (1994). *Synthesis and Optimization of Digital Circuits.* McGraw-Hill

Fišer, P. and Hlavička, J. (2002). A Flexible Minimization and Partitioning Method. *Proc. 5th Int. Workshop on Boolean Problems*, Freiberg (Germany), pp. 83-90

Fišer, P. and Hlavička, J. (2003a). A Heuristic Boolean Minimizer, *Computers and Informatics*, **Vol. 22, No. 1**, pp. 19-51

Fišer, P., Hlavička, J. and Kubátová, H. (2003b). FC-Min: A Fast Multi-Output Boolean Minimizer, *Proc. Euromicro Symposium on Digital Systems Design (DSD)*, Antalya (TR), pp. 451-454

Fišer, P., Hlavička, J. and Kubátová, H. (2003c). Column-Matching BIST Exploiting Test Don't-Cares. *Proc. 8th IEEE Europian Test Workshop (ETW)*, Maastricht (The Netherlands), pp. 215-216

Hassoun, S. and Sasao, T. (2002). *Logic Synthesis and Verification*, Boston, MA, Kluwer Academic Publishers, 454 pp.

Hlavička, J. and Fišer, P. (2001). BOOM - a Heuristic Boolean Minimizer. *Proc. International Conference on Computer-Aided Design (ICCAD)*, San Jose, California (USA), pp. 439-442

McGeer, P. et al. (1993). ESPRESSO-SIGNATURE: A new exact minimizer for logic functions, *Proc. DAC'93*