Weighted Don't Cares

Anna Bernasconi
Università di Pisa, Italy
annab@di.unipi.it

Valentina Ciriani
Università degli Studi di Milano, Italy
valentina.ciriani@unimi.it

Petr Fiser
Czech Technical University in Prague, Czech Republic
fiserp@fit.cvut.cz

Gabriella Trucco
Università degli Studi di Milano, Italy
gabriella.trucco@unimi.it

Abstract

In this paper we introduce and discuss the new concept of *weighted don't cares*, i.e., we propose to enrich the notion of don't cares, by assigning them a *weight*. These weights might be used to guide and refine the choices operated by the minimization algorithms in handling the don't care conditions. We then propose, and experimentally validate, the first synthesis tool for functions with weighted don't cares, called wBOOM. Experimental results show that wBOOM covers, on average, 66% more weighted don't cares than the classical synthesis tool BOOM.

1 Introduction

Logic optimization of digital circuits often benefits from the use of don't care conditions [10]. A *don't care* of a Boolean function $f : \{0,1\}^n \to \{0,1\}$, is a point v of the Boolean space $\{0,1\}^n$ where the value of the function is not specified, i.e., it could be either 1 or 0. The final value assigned to these points is usually decided by the minimization algorithm used to synthesize the function. For instance, if we want to represent f as a minimal sum of products, we will set to 1 all don't cares that allow to enlarge the dimension of the implicants and to get a more compact algebraic form. On the contrary, we will set to 0 all don't cares that would require the insertion of new products in the final form.

In this paper we propose to enrich the notion of don't cares, by assigning them a *weight*. Thus, we define and study the new concept of *weighted don't cares*. These weights might be used to guide and refine the choices operated by the minimization algorithms in handling the don't care conditions. Our idea comes from the observation that, in some synthesis scenarios, possibly different from the classical sum-of-products (SOP) minimization, some don't care points might help to reduce the area of the final circuit more than others. In other words, instead of treating all don't cares in the same way, we propose to enforce the minimization algorithms giving them some criteria to choose which don't cares should be preferentially covered. Weighted don't cares could also be applied in scenarios where there are don't cares that, for some reason, should be chosen before others. Observe that the weights can be decided before the synthesis phase, or can be assigned dynamically by the minimization algorithms during logic synthesis.

Weighted don't cares can be applied to different scenarios. In particular, in this paper we analyze a concrete application of the concept of weighted don't cares in the special synthesis framework of decomposition of Boolean functions onto overlapping subspaces [3, 1, 2, 4, 9, 10].

We then deal with another important issue, which is the development of the first synthesis tool for functions with weighted don't cares. We have considered the two-level Boolean minimizer called BOOM [8, 5, 6], and we have derived a new version, called wBOOM, that handles weighted don't cares, and uses the weights for choosing the don't cares that will be covered in the final circuit implementation of the function. We have experimentally evaluated this new tool, with interesting results.

The paper is organized as follows. In Section 2 we discuss the concept of weighted don't cares, and discuss some possible applications. In Section 3 we describe a minimization tool, wBOOM, that is sensitive to the presence of weighted don't cares. Experiments are reported in Section 4, in Section 5 we discuss the efficiency of wBOOM theoretically. Section 6 concludes the paper.

2 Weighted Don't Cares

A completely specified Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ can be simply represented by the subset of $\{0,1\}^n$ containing the points v such that $f(v) = 1$, i.e., the so-called *ON-set* of f. The set of all other points, i.e., the points v such that $f(v) = 0$, is called the *OFF-set* of f. Hereafter we will often denote the ON-set of the function f with the function f itself.

Let us now consider an incompletely specified function f, i.e., a Boolean function such that $f : \{0,1\}^n \rightarrow \{0,1,-\}$. A point v of the Boolean space $\{0,1\}^n$ such that $f(v) = -$ is called *don't care*. Thus, the *don't care set* (or DC-set) of f contains all the don't cares for f. When we synthesize an incompletely specified function f, the algebraic form for f must fulfill these requirements:

- it *must* cover all points of the ON-set;
- it *must not* cover any point of the OFF-set;
- it *might* cover some points from the DC-set, in order to ease the minimization and to get a more compact form.

In other words, the ON-set is the set of points that must be covered, the OFF-set represents the points that must NOT be covered, while we do not have precise requirements for the points in the DC-set, and we can choose whether covering or not covering them.

In this paper we want to enrich the notion of don't cares, by assigning them a weight and by using these weights to treat the don't care points differently. Our idea comes from the observation that some don't care points could be *more important* than other in the sense that, when covered, they might help, for example, in reducing the area of the final circuit more than other don't cares. So, instead of treating all don't cares in the same way, we propose to enforce the minimization algorithms, giving them some criteria to choose which don't cares should be preferentially covered. We introduce the following definition.

Definition 1 *Let f be an incompletely specified function, and let min, max ($min < max$) be two positive integers. The* weight *of a don't care point v of f is an integer $weight(v)$, $min \leq weight(v) \leq max$, that reflects the* degree of preference *of the don't care v, i.e., the convenience of covering v in the final circuit.*

Thus, don't cares with a bigger weight should be preferred to the other don't cares, and should be covered by the algebraic form for f.

2.1 Application

Weighted don't cares can be applied to different scenarios. For instance, one could decide to minimize a given function in steps. That is, instead of minimizing the whole function, that could be too big to handle, one could decide to minimize subsets of its ON-set, and then take the sum (i.e., the OR) of the resulting algebraic forms. Observe that these subsets are not necessarily disjoint, as it happens for instance when the function to be minimized is given in a PLA form, and we decide to minimize subsets of its products, in cascade. This "cascade minimization" of the function immediately suggests the use of weighted don't cares. Indeed, points already covered during the first minimization steps do not need to be covered, if encountered again in the next minimization phases. Thus, these points naturally become don't cares during the minimization process. It is also evident that, in order to avoid redundancies in the final circuit that would compromise its testability, the minterms of a function should not be covered by too many products. Therefore, we could assign a weight to the don't cares generated during

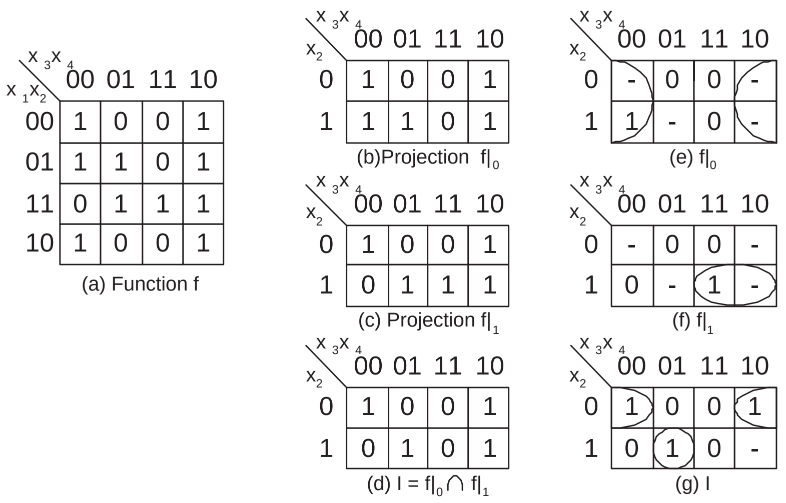

Figure 1: *The Boolean function f of the running example.*

the minimization steps, reflecting the number of times that a product has already covered that minterm: minterms covered only a few times should get a higher weight, while minterms already covered by many products should be assigned a low degree of preference, in order to condition the choices of the minimization algorithm.

We now describe a concrete application of the concept of weighted don't cares. This application arises in a very natural way from the framework of decomposition of Boolean functions onto overlapping subspaces [3, 1, 2, 4, 9, 10]. Suppose that we want to minimize a completely specified Boolean function f depending on n binary variables $x_1, x_2, \ldots, x_n$. Consider a variable x_i and the two subspaces of $\{0,1\}^n$ where $x_i = 1$ and where $x_i = 0$, whose characteristic functions are x_i and $\overline{x}_i$, respectively. If we project the given function f onto the two subspaces, we obtain the two Shannon cofactors $f|_1$ and $f|_0$, that represent the projections of f onto the spaces x_i and $\overline{x}_i$. Consider now the intersection of the two sets, i.e., consider the set $I = f|_0 \cap f|_1$. Observe that I, $f|_1$ and $f|_0$ do not depend on x_i. Thus, the original function f can be represented by the following algebraic form:

$$f = x_i f|_1 + \overline{x}_i f|_0 + I\,.$$

Consider for example the Boolean function f in Figure 1(a) depending on the four variables $x_1, \ldots, x_4$, whose ON-set is given by $f = \{0000, 0010, 0100, 0101, 0110, 1000, 1010, 1101, 1110, 1111\}$, and let $x_i = x_1$. We have $f|_0 = \{000, 010, 100, 101, 110\}$ (Figure 1(b)), $f|_1 = \{000, 010, 101, 110, 111\}$ (Figure 1(c)), and $I = \{000, 010, 101, 110\}$ (Figure 1(d)).

Note that the points in I are points that are also in the ON-sets of $f|_0$ and $f|_1$. For this reason we set such points as don't cares in $f|_0$ and $f|_1$, in facts they can be used for minimization of $f|_0$ and $f|_1$, but are not necessary since are already covered by I.

Therefore, in our example we have (Figures 1(e), 1(f), and , 1(d)):

$$\begin{aligned}
\text{ON-set } (f|_0) &= \{100\} \qquad \text{DC-set } (f|_0) = \{000, 010, 101, 110\}\\
\text{ON-set } (f|_1) &= \{111\} \qquad \text{DC-set } (f|_1) = \{000, 010, 101, 110\}\\
\text{ON-set } (I) &= \{000, 010, 101, 110\}\,.
\end{aligned}$$

Now, we note that each minterm of $f|_1$ corresponds to one minterm of f where $x_i = 1$, each minterm of $f|_0$ corresponds to one minterm of f where $x_i = 0$, while each minterm in I corresponds to two minterms in f (one with $x_i = 0$ and the other with $x_i = 1$). If we minimize $f|_1$ and $f|_0$ as SOP forms, obtaining for instance SOP_1 and SOP_0, respectively, we can note that if a point P of I is covered by both SOP_1 and SOP_0, then P can be set as a don't care in I. Thus, we propose to minimize the functions $f|_0$, $f|_1$ and I in this order: $f|_0$, $f|_1$ and finally I.

Consider again our running example, and suppose to minimize $f|_0$ in SOP form. We obtain $SOP|_0 = \overline{x}_4$; thus we have covered the only minterm, 100, in the ON-set, and the three points $000, 010, 110$ from the DC-set.

Let us now minimize $f|_1$. Our main observation is that now we can set a weight in the DC-set of $f|_1$, introducing two subsets of don't cares: *DC-set with high preference* $= \{000, 010, 110\}$ and *DC-set with low preference* $= \{101\}$. Indeed, if SOP_1 covers one of the points in $\{000, 010, 110\}$ (already covered by SOP_0) this point will be then set as a don't care for the intersection set I.

According to our idea, a *weighed minimizer* would prefer the cover given by the product x_2x_3 to the cover given by x_2x_4. Therefore, we have $SOP_1 = x_2x_3$.

We can now compute the don't care set for I as $I \cap (SOP_0 \cap SOP_1)$ (Figure 1(g)):

$$\text{ON-set } (I) = \{000, 010, 101\} \qquad \text{DC-set } (I) = \{110\}\,.$$

Thus we derive the SOP form $SOP_I = \overline{x}_2\overline{x}_4 + x_2\overline{x}_3x_4$, instead of $SOP_I = \overline{x}_2\overline{x}_4 + x_2\overline{x}_3x_4 + x_3\overline{x}_4$, that we would have computed from the alternative choice $SOP_1 = x_2x_4$. Finally, with the right choice of don't cares for $f|_1$, we obtain $f = \overline{x}_1 SOP_0 + x_1 SOP_1 + SOP_I = \overline{x}_1(\overline{x}_4) + x_1(x_2x_3) + (\overline{x}_2\overline{x}_4 + x_2\overline{x}_3x_4)$ containing 10 literals, instead of $f = \overline{x}_1(\overline{x}_4) + x_1(x_2x_4) + (\overline{x}_2\overline{x}_4 + x_2\overline{x}_3x_4 + x_3\overline{x}_4)$ containing 12 literals.

Observe that in the two particular decomposition techniques we have discussed, the weights of the don't cares are assigned dynamically during the synthesis phase.

3 Weighted BOOM: a Synthesis Tool for Functions with Weighted Don't Cares

The heuristic two-level (SOP) minimizer BOOM was proposed in [8, 5]. Later it was extended to handle multi-output functions more efficiently [6]. Basically, the algorithm consists of two steps: *generation of implicants* and *covering problem solution*. The major contribution of BOOM lies in the implicant generation phase. First of all, it is *randomized*. Thus, different results may be obtained from different runs on the same source data. This is exploited in the *iterative minimization* process, where the implicant generation phase is run repeatedly. Different implicants are collected in an *implicant pool*. The covering problem is solved at the end, using all the implicants, to obtain the final solution. This offers a possibility of trade-off between the result quality and runtime – better results may be obtained for a cost of runtime. This is visualized by Figure 2. A typical growth of the number of implicants (thin line) and decrease of the number of terms in the solution (bold line) in time are shown. Notice the different scales for these two curves.

The *covering problem* (CP) may be either solved exactly or approximately, using some heuristic to select implicants into the solution. As for the exact method, AURA-II [7] was implemented. This algorithm allows for choosing *any* implicant cost function and generates optimum solutions minimizing this cost. The implicant cost in the original BOOM is set to the number of literals. Note that there can also be more optimum solutions. Then AURA-II returns the first one found.

The approximate heuristic employed in BOOM is purely greedy, it constructs the solution by gradually adding implicants to it. The heuristic has several decision stages, where the candidate implicants are gradually filtered out:

1. Select implicants covering most of yet uncovered on-set terms.
2. From these, select implicants covering on-set terms that are difficult to be covered (they are covered by the minimum of implicants).
3. From these, select the ones with the least cost (the number of literals).
4. If there are still more possibilities, choose one randomly.

The Weighted BOOM (wBOOM) benefits from the excess of implicants entering the CP phase. Only few terms from the implicant pool form the solution, see Figure 2. Here, e.g., the solution consists of less than 100 terms out of 7,000 generated ones after 5,000 iterations. It is also very likely that many different solutions of equal quality exist.

The number of produced implicants may be further increased by introducing *mutations* into the implicant generation phase. Implicants that are valid, but normally unlikely to be selected into the solution, are produced in this way. For details see [5].

We can easily favour weighted DCs by influencing the CP cost function. We have decided for keeping the number of solution terms and literals as the primary criterion for our purpose.

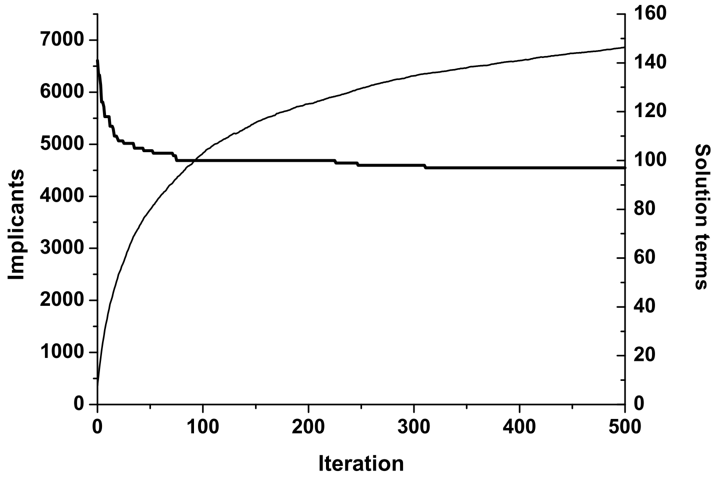

Figure 2: *The implicant generation progress in BOOM*

But next, if there are more equally valued solutions, the solution covering most of the wDCs will be preferred. This can be achieved by a very simple modification of the cost function: instead of being the number of literals, it will be defined as follows:

$$cost(term) = literals(term) * big\_number - covered\_wDCs(term),$$

where *big_number* is any number higher than the number of possibly covered wDCs. This ensures that the number of literals will be minimized preferably, while the number of covered DCs will be the secondary criterion. But definitely the cost function may be modified in other ways, depending on the actual designer's demands.

Note that this approach can easily be generalized to support multiple DC weights, too. We can just compute the summary weight of covered DCs, instead of counting the number of covered wDCs. This is the way actually implemented in wBOOM.

4 Experimental Results

The synthesis tool that takes the weighted don't cares into account, wBOOM, has been applied to the ESPRESSO benchmark suite [11], running on a Pentium 1.6 GHz processor with 1 GB RAM. We used the weights $\{0, 1\}$ for don't cares derived by the overlapping synthesis problem described in [2], and briefly recalled in Section 2. Weight 1 means that the don't care has a higher priority than don't cares with the weight 0. We have tested the practical performance of wBOOM, by comparing its results to the classical BOOM minimizer [6]. Both tools have been run with the choices of 200 iterations and 20% of CD-search mutations [5].

wBOOM first optimizes the number of products and literals, and then it tries to maximize the covered don't care weighs, that is, in the considered scenario, wBOOM maximizes the number of covered don't cares with weight equal to 1 (shortly denoted as *1-wDCs*). Therefore, the discussion is based on the comparison between the number of 1-wDCs covered by wBOOM compared to the ones covered by BOOM.

We report in Table 1 a significant subset of the results. The first column reports the name of the instance considered. The following three columns refer to the experiments with wBOOM and report the number of products, the number of covered 1-wDCs, and the computational time (in seconds). The next three columns report the number of products, the number of covered 1-wDCs, and the computational time (in seconds) obtained with the BOOM minimizer. The last two columns compare the results showing the **Absolute gain** (i.e., the difference between number of 1-wDCs covered by wBOOM and BOOM) and the **Gain rate** (i.e., the difference between the number of 1-wDCs covered by wBOOM and BOOM, divided by the number of 1-wDCs covered by wBOOM).

	wBOOM			**BOOM**			**Comparison**	
	Products	**1-wDCs**	**Time**	**Products**	**1-wDCs**	**Time**	**Absolute gain**	**Gain rate**
add6	93	30	16.94	93	11	17.16	19	0.63
alu2	14	28	0.66	14	6	0.66	22	0.79
amd	1	0	0.30	1	0	0.28	0	0.00
b12	11	7	0.22	11	5	0.20	2	0.29
b9	12	0	3.19	12	0	3.17	0	0.00
bench	3	3	0.05	3	1	0.03	2	0.67
co14	13	117	1.06	13	0	1.06	117	1.00
dc2	18	162	0.30	18	0	0.34	162	0.00
ex1010	52	49	72.61	50	29	92.25	20	0.41
exep	50	148	13.77	50	1	13.78	147	0.99
f51m	14	78	0.22	14	6	0.22	72	0.92
ibm	12	0	3.73	12	0	3.77	0	0.00
in0	34	126	4.95	34	3	4.94	123	0.98
in5	25	161	9.45	26	7	9.47	154	0.96
life	35	315	2.61	35	0	2.59	315	1.00
m1	12	12	0.09	12	12	0.09	0	0.00
m2	44	148	1.06	44	1	1.08	147	0.99
max1024	86	312	5.55	87	6	5.53	306	0.98
max512	40	122	1.47	40	15	1.45	107	0.88
newcwp	5	21	0.02	5	3	0.01	18	0.86
newtpla2	3	19	0.09	3	0	0.11	19	1.00
p3	10	7	0.25	10	5	0.25	2	0.29
prom2	225	751	57.63	227	49	62.89	702	0.93
rckl	31	279	13.53	31	0	13.52	279	1.00
rd73	32	240	1.55	32	6	1.55	234	0.98
shift	50	18	0.92	50	18	0.84	0	0.00
sqn	12	49	0.25	12	4	0.16	45	0.92
t1	2	18	0.17	2	0	0.16	18	1.00
test1	36	51	2.52	36	6	2.63	45	0.88
tial	170	202	180.09	171	89	163.03	113	0.56
x1dn	15	15	36.09	15	10	36.36	5	0.33
z4	14	126	0.33	14	0	0.25	126	1.00
AVERAGE							**104.38**	**0.66**
STD DEV							**189.77**	**0.39**

Table 1: Comparison between wBOOM and BOOM.

Comparing the number of covered 1-wDCs, we can notice that even if the main synthesis objective is the minimization of the number of products, wBOOM succeeds in maximizing the number of covered weighted don't cares without loosing in minimization time. Indeed, the wBOOM covers, on average, 66% more weighted don't cares that BOOM.

We have also tested wBOOM in the particular decomposition problem described in Section 2.1. wBOOM improved the final form in about 10% of the considered benchmarks, but the gain was quite low (about 1% less products), probably because the SOP_Is were already very near to the optimum.

5 Solutions Count Analysis

Assuming the cost function from Section 3, the necessary condition for success of wBOOM is the existence of different solutions with the same number of literals. Then, wBOOM will return the one maximizing the number of covered DCs. One may wonder if this happens in practice – do there exist more solutions of equal size for practical examples? We may also ask how many different *optimum* solutions exist.

We have performed the following experiment to answer these questions: we have run BOOM (not wBOOM this time) in the same configuration as in Section 4 (200 iterations, 20% CD-Search mutations) 100-times and recorded all *different* solutions ever obtained (even in course of the iteration). Note that all the results were prime and irredundant covers. Results for some of the benchmarks coming from the decomposition process (see Subsection 2.1) are shown in Table 2. The total number of different solutions is shown in the second column, after the benchmark name. Then the number of obtained different "best" solutions is given. Then, numbers and percentages of solutions, whose quality is less than 5% (10%, 20%, respectively) worse than the best solution are shown.

We can observe that for most of the circuits only one "best" solution was obtained, which was probably the optimum one. Of course, symmetric circuits, like *max512*, *sym10*, *Z9sym* [11] have adequate numbers of different P-equivalent solutions. However, a plentiful of different near-optimum solutions can be observed. This is illustrated in Figure 3 for the *ex1010* [11] circuit. Such a behavior can be observed for most of the tested circuits.

benchmark	solutions	best solutions	≤ 5%	≤ 10%	≤ 20%
add6	2598	1	19 (1%)	218 (8%)	2580 (99%)
alu2	114	1	38 (33%)	73 (64%)	105 (92%)
alu3	1444	1	204 (14%)	495 (34%)	1133 (78%)
amd	2	1	1 (50%)	1 (50%)	1 (50%)
b12	212	1	26 (12%)	88 (42%)	212 (100%)
b9	929	1	52 (6%)	154 (17%)	319 (34%)
bench	4283	50	2599 (61%)	3803 (89%)	4213 (98%)
co14	1	1	1 (100%)	1 (100%)	1 (100%)
dc1	1	1	1 (100%)	1 (100%)	1 (100%)
dc2	16	1	1 (6%)	7 (44%)	16 (100%)
ex1010	17020	1	293 (2%)	6144 (36%)	16172 (95%)
ex7	909	2	47 (5%)	143 (16%)	315 (35%)
exep	2343	2	2139 (91%)	2334 (100%)	2343 (100%)
f51m	315	1	19 (6%)	70 (22%)	296 (94%)
ibm	1242	1	45 (4%)	169 (14%)	700 (56%)
in7	43	4	4 (9%)	18 (42%)	34 (79%)
inc	24	2	8 (33%)	11 (46%)	24 (100%)
jbp	1166	1	97 (8%)	429 (37%)	829 (71%)
life	1	1	1 (100%)	1 (100%)	1 (100%)
log8mod	37	2	19 (51%)	30 (81%)	37 (100%)
luc	4	2	4 (100%)	4 (100%)	4 (100%)
m1	9	1	2 (22%)	2 (22%)	9 (100%)
m2	141	1	87 (62%)	135 (96%)	141 (100%)
max512	1131	122	682 (60%)	947 (84%)	1129 (100%)
misj	15	1	1 (7%)	1 (7%)	1 (7%)
mlp4	338	2	25 (7%)	138 (41%)	328 (97%)
newcwp	2	1	1 (50%)	1 (50%)	2 (100%)
newtpla2	25	1	4 (16%)	7 (28%)	9 (36%)
p3	20	2	2 (10%)	2 (10%)	18 (90%)
p82	12	1	6 (50%)	9 (75%)	12 (100%)
radd	344	1	3 (1%)	11 (3%)	80 (23%)
rckl	214	1	214 (100%)	214 (100%)	214 (100%)
rd73	1	1	1 (100%)	1 (100%)	1 (100%)
risc	1	1	1 (100%)	1 (100%)	1 (100%)
root	1163	3	249 (21%)	902 (78%)	1163 (100%)
ryy6	8499	1	566 (7%)	2649 (31%)	7406 (87%)
shift	414	1	230 (56%)	313 (76%)	414 (100%)
soar	1043	1	9 (1%)	59 (6%)	506 (49%)
sqn	113	1	23 (20%)	93 (82%)	113 (100%)
sym10	875	100	147 (17%)	242 (28%)	483 (55%)
t1	8	1	1 (13%)	1 (13%)	7 (88%)
test1	2937	1	1463 (50%)	2449 (83%)	2864 (98%)
test4	9702	4	58 (1%)	1116 (12%)	9666 (100%)
vg2	2	1	1 (50%)	1 (50%)	2 (100%)
vtx1	1149	4	929 (81%)	1009 (88%)	1103 (96%)
x1dn	8	1	1 (13%)	1 (13%)	1 (13%)
x2dn	5	1	1 (20%)	4 (80%)	5 (100%)
x9dn	741	1	634 (86%)	675 (91%)	731 (99%)
z4	395	1	37 (9%)	202 (51%)	381 (96%)
Z5xp1	153	1	61 (40%)	137 (90%)	153 (100%)
Z9sym	3178	99	751 (24%)	1329 (42%)	2541 (80%)

Table 2: Numbers of solutions

We can conclude that wBOOM becomes efficient especially for symmetric functions, provided that don't cares are not symmetrical as well. Next, if the optimum solution is not required, many solution choices are available, thus don't cares can be exploited very efficiently, too.

6 Conclusion

We have introduced a new concept of weighted don't cares, and proposed a minimizer wBOOM, for a two-level (SOP) synthesis of Boolean functions with weighted don't cares.

Future work includes a more complete exploration of different synthesis scenarios that could benefit from the notion of weighted don't cares. Moreover, as the present version of wBOOM optimizes the number of products before trying to maximize the covered DC weighs, we would like to design a new version of wBOOM even more sensitive to the weights of the don't care points, by considering DC weights in the very implicant generation phase. It would be also interesting to find other applications of the concept of weighted don't cares showing more interesting gains in the size of the final circuit implementations.

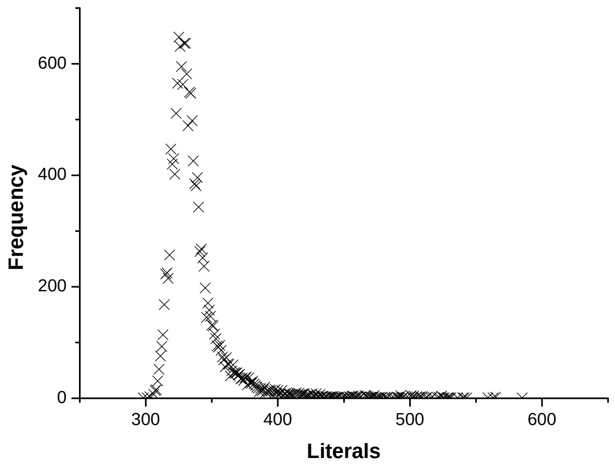

Figure 3: *Distribution of different implicants*

References

[1] A. Bernasconi, V. Ciriani, and R. Cordone. On Projecting Sums of Products. In *11th Euromicro Conference on Digital Systems Design: Architectures, Methods and Tools*, pages 787–794, 2008.

[2] A. Bernasconi, V. Ciriani, G. Trucco, and T. Villa. On Decomposing Boolean Functions via Extended Cofactoring. In *Design Automation and Test in Europe (DATE)*, pages 1464–1469, 2009.

[3] T. Czajkowski and S. Brown. Functionally Linear Decomposition and Synthesis of Logic Circuits for FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(12):2236–2249, 2008.

[4] E. Dubrova. A polynomial time algorithm for non-disjoint decomposition of multi-valued functions. In *International Symposium on Multi-Valued Logic*, pages 309–314, 2004.

[5] P. Fiser and J. Hlavicka. BOOM, A Heuristic Boolean Minimizer. *Computers and Informatics*, 22(1):19–51, 2003.

[6] P. Fiser and H. Kubatova. Flexible two-level boolean minimizer BOOM-II and its applications. In *Euromicro Conference on Digital Systems Design (DSD)*, pages 369–376, 2006.

[7] E. Goldberg, L. Carloni, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Negative thinking by incremental problem solving: application to unate covering. In *Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, pages 91–98, Washington, DC, USA, 1997.

[8] J. Hlavicka and P. Fiser. BOOM, a heuristic boolean minimizer. In *Proc. of International Conference on Computer-Aided Design,*, pages 493–442, 2001.

[9] B. Rytsar. A new approach to the decomposition of Boolean functions. 4. non-disjoint decomposition: the method of p, q-partitions. *Cybernetics and Sys. Anal.*, 45(3):340–364, 2009.

[10] T. Sasao. *Switching Theory for Logic Synthesis.* Kluwer Academic Publishers, 1999.

[11] S. Yang. Logic synthesis and optimization benchmarks user guide version 3.0. User guide, Microelectronic Center, 1991.