A Fast Algorithm for Three-Level Logic Optimization

Elena Dubrova   Peeter Ellervee
Department of Electronics
Royal Institute of Technology
S-164 40 Kista, Sweden
{elena, lrv}@ele.kth.se

Abstract

Three-level logic has been shown to have a potential for reduction in area over two-level implementations, as well as for gain in speed over multi-level implementations. In [1], Malik, Harrison and Brayton present an algorithm for three-level logic optimization, allowing fast and area-efficient implementations to be obtained for some practical functions. Unfortunately, the algorithm can be quite time-consuming for large functions. In this paper, we modify the algorithm [1] with the aim of reducing its run-time performance. This is achieved by introducing: (1) a novel strategy for pairing cubes, (2) a condition for termination of the current loop, and (3) a more efficient cost function. The experimental results show that, compared to the algorithm [1], our algorithm is 39 times faster on average at the expense of 6% larger solutions.

1. Introduction

Three-level logic is shown to be a good trade-off between the speed of two-level logic and the density of multi-level logic [2]. The optimization problem for three-level logic is harder than for two-level logic, but much simpler than for multi-level logic.

The first algorithm, addressing the optimization of three-level logic was presented in 1991 by Malik, Harrison and Brayton [1]. They consider Programmable Logic Devices (PLDs) whose simplified logic block consists of two Programmable Logic Arrays (PLAs), implementing the first two levels of logic, and a set of two-input gates, called *logic expanders*, implementing the third level. Each logic expander can be programmed to realize any function of two variables. Such a PLD implements logic expressions of the type:

$$f(x_1, \ldots, x_n) = (P_1 + \ldots + P_k) \circ (P_{k+1} + \ldots + P_l) \quad (1)$$

where P_i, $i \in \{1, \ldots, l\}$ denotes an arbitrary product-term involving some of the variables $x_1, \ldots, x_n$ or their complements, "$\circ$" denotes a binary operation, and $1 \leq k \leq l$. The authors of [1] show that the number of product-terms in the three-level expression obtained by the algorithm can be significantly smaller (up to a factor of 5) than the number of product-terms in the expression obtained by a two-level AND-OR minimizer. Unfortunately, the algorithm can be quite time-consuming for large functions. The number of iterations it performs is r^3, where r is the number of product-terms in the cover of the on-set of the input function. For an n-variable function, r may be as large as 2^{n-1}. In this paper, we modify the algorithm [1] with the aim of reducing its run-time performance. This is achieved by introducing (1) a novel strategy for pairing product-terms and (2) a condition for termination the current loop and (3) a more efficient cost function. The experimental results show that, on average, these modifications yield a 39 times reduction in the run-time, at the expense of 6% more product-terms in the obtained solutions.

2. Notation

Let $f(x_1, x_2, \ldots, x_n)$ be an incompletely specified Boolean function of type $f : \{0,1\}^n \rightarrow \{0, 1, -\}$, of the variables $x_1, \ldots, x_n$, where "$-$" denotes a don't care value.

A *product-term* is a Boolean product (AND) of one or more variables $x_1, \ldots, x_n$ or their complements. A convenient representation for a product-term is *cube* [4]. We use the terms *cube* and *product-term* interchangeably.

We use F_f, R_f and D_f to denote on-set, off-set and don't-care-set of a function f, respectively. The *size* of a set A, denoted by $|A|$, is the number of cubes in it. The *complement* of a set A, denoted by $\overline{A}$, is the intersection of the complements for each cube of A. The *intersection* of two sets A and B, denoted by $A \cap B$, is the union of the pairwise intersection of the cubes from A and B. The *union* of two sets A and B, denoted by $A \cup B$, is the union of the cubes from A and B.

A *supercube* of two cubes c_1 and c_2, denoted by $sup(c_1, c_2)$, is the smallest cube containing both c_1 and c_2.

We say that cube c_1 is *degenerate* in the variable x_i if it has don't care value in the ith position. The *dimension* of cube c_1 is the number of degenerate variables in it.

Let $N_d(c_1, c_2)$ denote the number of variables in which both c_1 and c_2 are degenerate. For example, cubes $0-1$ and $1--$ are both degenerate in the variable x_2, so $N_d(0-1, 1--) = 1$. Then, it is easy to show that the dimension of the supercube of c_1 and c_2 is given by $d(c_1, c_2) + N_d(c_1, c_2)$, where $d(c_1, c_2)$ is defined by:

$$d(c_1, c_2) \stackrel{df}{=} \sum_{i=1}^{n} (c_{1i} \neq c_{2i})$$

For example, $d(0-1, 1--) = 2$, and therefore the dimension of the supercube of $0-1$ and $1--$ is $2+1=3$.

3. Algorithm [1]

In this section we briefly describe the basic idea of the algorithm presented in [1].

First, a minimal expression of the type shown in equation (1) for the case of "$\circ$" = AND is determined. Second, output phase optimization is applied to the logic expander to check suitability of other choices of "$\circ$". Such a scheme utilizes all interesting cases except for "$\circ$" = XOR, "$\circ$" = XNOR. Several algorithms addressing "$\circ$" = XOR case are presented in [5], [6] and [7].

In order to find two sets of cubes g_1 and g_2 such that $g_1 \cap g_2$ is a cover for the on-set of the input function f, the algorithm [1] repeats the following: given an initial choice of init_g_1, for each pair of cubes $(c_1, c_2) \in$ init_g_1, the supercube of c_1 and c_2 is added to init_g_1 and the cost function is computed. After all pairs have been tried once, the pair which produces the greatest decrease or the smallest increase in cost is selected and added to init_g_1. Each such loop reduces the size of init_g_1 by at least one cube. The algorithm iterates until init_g_1 reduces to one cube.

4. Our algorithm

In this section we describe our algorithm and summarize its novel features as compared to the algorithm [1]. The target is a minimal expression of the type shown in equation (1) for the case of "$\circ$" = AND. After such an expression is found, output phase optimization is applied. We use the output phase assignment algorithm of Espresso [3]. In the final implementation, g_1 is implemented by PLA1 and g_2 is implemented by PLA2. The outputs of PLAs are combined using AND gates. However, some AND gates may have one or both inputs and possibly the output inverted, depending on the phase assignment of g_1 and g_2.

The pseudocode of our algorithm is shown in Figure 1. It receives as its input an incompletely specified multiple-output Boolean function f (in Espresso format). It returns as its output two sets of cubes g_1 and g_2, such that $g_1 \cap g_2$ is a cover for the on-set of f. The objective is to minimize the total number of cubes in g_1 and g_2.

Minimal_AND_OR_AND(F_f, D_f, R_f)
input: on-set F_f, don't care set D_f and off-set R_f of f
output: sets of cubes g_1 and g_2, such that $g_1 \cap g_2 \supseteq F_f$, and the total number of cubes in g_1 and g_2 is minimized

```
init_g1 = F_f;
best_cost = ∞;
while (init_g1 has more than one cube) {
    best_cost_local = ∞;
    for (each (c1, c2) ∈ init_g1 with d(c1, c2) ≤ 2) {
        Replace c1 and c2 by their supercube c
        g2 = Cover(F_f, D_f ∪ ¬init_g1, init_g1 ∩ R_f);
        if (|g2| < |F_f|/2) {
            g1 = Cover(F_f, D_f ∪ ¬g2, g2 ∩ R_f);
            cost = |g1| + |g2|;
        }
        else
            cost = ∞;
        if (cost < best_cost_local) {
            best_cost_local = cost;
            next_init_g1 = init_g1;
        }
        if (cost < best_cost) {
            best_cost = cost;
            best_g1 = g1;
            best_g2 = g2;
            break;
        }
        Restore init_g1;
    }
    init_g1 = next_init_g1;
}
if (F_f ⊄ best_g1 ∩ best_g2)
    return("verify error");
else
    return(best_g1, best_g2);
```

Figure 1. Pseudocode of our algorithm.

Our algorithm repeats the following basic steps (same as the basic steps of the algorithm [1]):

1. Choose an initial a set of cubes init_g_1, such that init_$g_1 \supseteq F_f$.

2. Find a smallest set of cubes g_2, satisfying the condition init_$g_1 \cap g_2 \supseteq F_f$

3. Find a smallest set of cubes g_1, satisfying the condition $g_1 \cap g_2 \supseteq F_f$

4. Repeat 1, 2 and 3 for several init_g_1 and save g_1 and g_2 with the smallest number of cubes in total.

At the first iteration of the **while**-loop, init_$g_1 = F_f$. At each iteration of the **for**-loop, a pair of cubes $(c_1, c_2) \in$ init_g_1 with $d(c_1, c_2) \leq 2$ is selected and replaced by their supercube. In this way, the size of init_g_1 is reduced by at least one cube (or by more, if the supercube contains other cubes from init_g_1). Since the supercube contains the two cubes it replaces, init_$g_1 \supseteq F_f$. The value of $d(c_1, c_2)$ is proportional to the dimension of the supercube of c_1 and c_2 (with respect to the variables which are non-degenerate in either c_1 or c_2, but not in both). Our experimantal results show that by checking only smaller supercubes we are doing much less iterations, but we are still likely to find a global (or close to a global) minimum.

Next, we look for g_2 such that init_$g_1 \cap g_2$ is a cover for F_f. It is shown in [1] that, given two sets of cubes g_1 and g_2 and a Boolean function $f = (F_f, D_f, R_f)$, $g_1 \cap g_2 \supseteq F_f$ if and only if $F_f \subseteq g_1$, $F_f \subseteq g_2$ and $R_f \cap g_1 \cap g_2 = \emptyset$. Therefore, in order init_$g_1 \cap g_2$ to be a cover for F_f, g_2 *should* contain F_f, it *may* contain any of the cubes which are not in init_g_1, and it *should not* contain any of the cubes which are in init_$g_1 \cap R_f$. If f is incompletely specified, g_2 may as well contain any of the cubes from D_f. To find a smallest set of cubes, satisfying this condition, we employ subroutine **Cover()**, which takes the function with F_f as on-set, $D_f \cup \overline{\text{init\_}g_1}$ as don't-care-set and init_$g_1 \cap R_f$ as off-set, and returns a cover for the on-set. It implements **Reduce**(), **Expand**() and **Irredundant**() subroutines of Espresso [3] to comprise a single pass of the minimization algorithm.

Once a cover for g_2 is found, we check whether its size is less than a half of the size of F_f. If this is the case, g_1 satisfying the condition init_$g_1 \cap g_2 \supseteq F_f$, is constructed in the same way as above, except that g_2 plays the role of init_g_1. Otherwise, the algorithm goes to the next iteration of the **for**-loop.

The **for**-loop is terminated as soon as the first pair of cubes with the cost smaller than the current global best cost is found. The pair of cubes which produces the greatest decrease or the smallest increase in cost is selected and added to init_g_1. This init_g_1 becomes init_g_1 for the next iteration of the **while**-loop. The algorithm iterates until init_g_1 is reduced to one cube.

Compared to the algorithm [1], the main novel features of our algorithm are:

1. In [1], the **for**-loop is iterated for all pairs of cubes in init_g_1. We reduce the number of iteration in two ways:

- The **for**-loop is entered only for the pairs of cubes c_1, c_2 with $d(c_1, c_2) \leq 2$. Our experimantal results shows that this gives an average reduction of 29% in the number of **for**-loops.

- The **for**-loop is terminated as soon as the first pair of cubes with the cost smaller than the current global best cost is found. g_1 yielding this cost becomes init_g_1 for the next iteration in the **while**-loop. This condition is more "risky" than the previous one, but it gives us the main savings in time (X% average reduction of **for**-loops).

2. In the algorithm [1], both g_1 and g_2 are constructed and minimized in each **for**-loop. The minimization step is the most time-consuming part of the algorithm, even when only a single pass of the minimization algorithm is performed, like in **Cover()**. In our algorithm, once g_2 is found, we check whether the size of g_2 is less than a half of the size of F_f. If this is the case, g_1 is constructed and minimized. Otherwise, the algorithm goes to the next iteration of the **for**-loop. This yields an average reduction of 31% in the run-time for each iteration.

5. Experimental results

We have applied our algorithm to a set of benchmark functions and have compared its results to the performance of the algorithm [1].[1] Table 1 shows the number of cubes in the resulting set of cubes g_1 and g_2 (columns 5, 6 and columns 9, 10) and the time taken in seconds (columns 8 and 12). The time is user time measured using the UNIX system command *time*. All programs were run on a Sun Ultra 60 operating with two 360 MHz CPU and with 1024 MB RAM main storage.

Columns 2 and 3 give the number of inputs n and the number of outputs m of the benchmarks functions. Column 4 refers to the number p^e of cubes in the cover computed by Espresso [3] and columns 7 and 11 show the improvement over Espresso for the algorithm [1] and our algorithm, respectively, computed as $\frac{p^e}{|g_1|+|g_2|}$.

The last two columns compare the solution computed by our algorithm to the solution obtained by the algorithm [1]. Column 13 shows the improvement in terms of the number of cubes, computed as $1 - \frac{(|g_1|+|g_2|)_{our}}{(|g_1|+|g_2|)_{[1]}}$. Negative number indicates that g_1 and g_2 generated by our algorithm have more cubes in total than g_1 and g_2 obtained by the algorithm [1]. For example, -0.06 means that $|g_1| + |g_2|$ for our algorithm is 6% larger than $|g_1| + |g_2|$ for the algorithm [1]. Column 14 shows how many times our algorithm is faster than the algorithm [1].

The experimental results demonstrate that, on average, our algorithm is 39 times faster than the algorithm [1] at the expense of 6% more cubes in g_1 and g_2 in total.

[1] The code of the algorithm [1] is not publicly available. The data we present are the results of our implementation.

Table 1. Experimental results.

Example function	n	m	Espr. p^e	Algorithm [1] $\|g_1\|$	$\|g_2\|$	$\frac{p^e}{\|g_1\|+\|g_2\|}$	t_1, sec	our Algorithm $\|g_1\|$	$\|g_2\|$	$\frac{p^e}{\|g_1\|+\|g_2\|}$	t_2, sec	impr./ wors.	$\frac{t_1}{t_2}$
5xp1	7	10	65	31	24	1.18	757	30	25	1.18	14	0	54
alu2	10	8	68	22	28	1.36	722	33	19	1.31	32	-0.040	23
alu3	10	8	66	23	21	1.50	380	28	19	1.40	24	-0.068	16
b12	15	9	43	19	9	1.54	576	19	9	1.54	30	0	19
dist	8	5	123	82	33	1.07	18117	77	31	1.14	170	0.061	107
newapla2	6	7	7	4	1	1.40	1.3	4	1	1.40	0.39	0	3.3
newbyte	5	8	8	4	1	1.60	2.4	4	1	1.60	0.52	0	4.6
newcpla1	9	16	38	27	3	1.27	611	25	2	1.41	26	0	24
newtpla	15	5	23	4	15	1.21	26	17	2	1.21	1.2	0	22
radd	8	5	75	23	14	2.03	716	19	20	1.92	14	-0.054	51
rd53	5	3	31	14	10	1.29	30	12	14	1.19	5.9	-0.083	5.1
rd73	7	3	127	20	71	1.40	1263	44	35	1.61	312	0.132	4.1
ryy6	16	1	112	2	5	16.00	3584	2	5	16.00	379	0	9.5
sqn	7	3	38	29	4	1.15	253	27	7	1.12	6.1	0	42
t2	17	16	53	26	18	1.20	2330	37	9	1.15	44	0	53
vg2	25	8	110	22	40	1.77	10154	62	26	1.25	366	-0.419	28
x1dn	27	6	110	23	40	1.75	14312	28	53	1.36	237	-0.288	47
x9dn	27	7	120	24	41	1.85	24291	53	28	1.48	307	-0.246	79
z4	7	4	59	19	10	2.03	369	16	17	1.79	2.1	-0.138	176
Z5xp1	7	10	65	34	21	1.18	961	27	29	1.16	112	-0.018	8.6
average	12	7	67	23	20	2.19	3973	28	18	2.11	104	-0.058	39

6. Conclusion

In this paper we modify the algorithm [1] to reduce its run-time. The experimental results show that, on average, our algorithm gives 39 times speed-up in the run-time at the expense of 6% larger solution.

Our ongoing research includes: (a) incorporating clustering as a preprocessing step, to further reduce run-time, and (b) extending the algorithm for XOR and XNOR cases. That would give us a tool targeting a minimal expression of the type shown in equation (1) for any binary operation “$\circ$”. Such a tool could be used for multi-level logic optimization by being applied recursively to the resulting solution until no more improvement is encountered.

References

[1] A. A. Malik, D. Harrison, R.K. Brayton, ”Three-level decomposition with application to PLDs”, *IEEE Int. Conference on Computer Design*, 1991, pp. 628-633.

[2] T. Sasao, ”On the complexity of three-level logic circuits”, *Proc. MCNC Int. Workshop on Logic Synthesis*, Research Triangle Park, North Carolina, May 1989.

[3] R.K. Brayton, G. Hachtel, C. McMullen, A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publisher, 1984.

[4] Y. H. Su, P. T. Cheung, ”Computer minimization of multi-valued switching functions”, *IEEE Trans. Comput.*, vol. C-21, 1972, pp. 995-1003.

[5] T. Sasao, ”A design method for AND-OR-EXOR three-level networks”, *Proc. Int. Workshop on Logic Synthesis*, 1995, pp. 8:11-8:20.

[6] E. V. Dubrova, D. M. Miller, J. C. Muzio, ”AOXMIN: A three-level heuristic AND-OR-XOR minimizer for Boolean functions”, *Proc. 3rd Int. Workshop on the Applications of the Reed-Muller Expansion in Circuit Design*, 1997, pp. 209-218.

[7] D. Debnath, T. Sasao, ”A heuristic algorithm to design AND-OR-EXOR three-level networks”, *Proc. Asia and South Pacific Design Automation Conf.*, 1998.