AOXMIN-MV: A Heuristic Algorithm for AND-OR-XOR Minimization

E. V. Dubrova

Electronic System Design Lab
Department of Electronics
Royal Institute of Technology
KTH-Electrum, Electrum 229
S-164 40 Kista, Sweden
elena@ele.kth.se

D.M. Miller J.C. Muzio

VLSI Design and Test Group
Department of Computer Science
University of Victoria
P.O.Box 3055
Victoria, B.C., Canada, V8W 3P6
{mmiller, jmuzio}@csr.uvic.ca

Abstract

Three-level logic is shown to have a potential for reduction of the area over two-level implementations, as well as for a gain in speed over multi-level implementations. In this paper we present an heuristic algorithm, AOXMIN-MV, targeting a three-level logic expression which is an XOR of two sum-of-products. For some practical functions, such an AND-OR-XOR expression may have up to 60 times less product-terms compared to the classical sum-of-products form. Several algorithms for finding minimal AND-OR-XOR expressions were presented, but they all are time-consuming for large functions. The algorithm presented here solves this problem by (1) introducing an estimation metric, checking whether the input function is likely to have a compact AND-OR-XOR expression; (2) employing a new strategy for decomposing the input function into two sum-of-products; (3) treating the output part of a multiple-output function as a single multiple-valued variable. The experimental results show that these modification yield a faster and more efficient algorithm. Furthermore, it gives a solution to a more general problem of minimization of multiple-valued input binary-valued output logic functions.

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 [1]. The optimization problem for three-level logic is harder than that for two-level logic, but much simpler than for multi-level logic. The existing algorithms for logic optimization cannot be directly adapted to handle the three-level problem efficiently.

Many devices for implementing three-level logic are industrially offered, e.g. Xilinx's XC4000E and XC4000X series Field Programmable Gate Arrays [2], or Altera's Apex 20K, Flex 10KE and Flex 8000 Programmable Logic Devices (PLDs) families [3], [4]. A simplified logic block of these devices consists of a set of Programmable Logic Arrays, implementing the first two levels of logic, and a set of *logic expanders*, implementing the

third level. Each logic expander can be programmed to realize any function of two variables. Such a logic block implements a logic expression of the type:

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

where $P_i,\ i \in \{1, \ldots, r\}$ denotes an arbitrary product-term involving some of the variables $x_1, \ldots, x_n$ or their complements, "$\circ$" denotes a binary operation, and $1 \leq k \leq r$.

The first algorithm, addressing the optimization of such PLDs, was presented in 1991 by Malik, Harrison and Brayton in [5]. It was shown that the number of products in the three-level expression obtained by the algorithm can be significantly smaller (up to a factor of 5) than the number of products in the expression obtained by a two-level AND-OR minimizer. The algorithm [5] first determines a minimal expression (1) for the case of "$\circ$" = AND, and then applies output phase optimization to the logic expander to check suitability of the other choices of "$\circ$". Such a scheme minimizes (1) for all interesting cases except "$\circ$" = XOR, "$\circ$" = XNOR. A modified version of the algorithm [5], aiming to reduce its run-time performance, is presented in [6].

Several algorithm addressing "$\circ$" = XOR case were presented in [7], [8] and [9]. It was shown that an AND-OR-XOR expression has a smaller upper bound on the number of product-terms than the upper bound of an AND-OR or an AND-XOR expansion, implying the existence of a more economical circuit implementation for some functions [10], [11] . Furthermore, the experimental results of [8] have demonstrated that, for some functions, assigning "$\circ$" = XOR may lead to a reduction by a factor of 27 in the number of product-terms in the expression (1) compared to the number of product-terms in the sum-of-product expression obtained by a two-level AND-OR minimizer.

Unfortunately, the algorithms [7]-[9] can be time-consuming for large functions. Our experimental results show that only about 30% of practical functions have compact AND-OR-XOR forms, and for the rest the number of product-terms in their minimal AND-OR-XOR form is comparable to the number of product-terms in their minimal sum-of-products form. For a large benchmark, running an XOR-based optimization algorithm for several hours to find out that the size of the resulting AND-OR-XOR expression is very similar to the cover produced by a two-level AND-OR minimizer Espresso [12] is quite depressing.

We have found an efficient solution to this problem. We have developed an estimation metric, which gives a fast answer as to whether an input function is suitable for "$\circ$" = XOR (XNOR) optimization or not. This metric is used in the new algorithm AOXMIN-MV for finding minimal AND-OR-XOR expressions, which is reported in this paper. It evolved from the algorithm for three-level **A**ND-**O**R-**X**OR **MIN**imization AOXMIN [8]. The novel features of AOXMIN-MV are:

1. AOXMIN-MV uses the new estimation metric, checking suitability of the input function for "$\circ$" = XOR (XNOR) in the expression given by the equation (1) or not. If the function is found not suitable, the algorithm terminates. Otherwise, it continues.

2. AOXMIN-MV uses an extension of Fiduccia/Mattheyses partitioning algorithm for finding a best decomposition of the on-set of the input function into two groups [13]. In AOXMIN, this decomposition was made randomly.

3. AOXMIN was essentially a single-output minimizer. For a multiple-output problem, it minimized each function independently, and then applied Espresso to the resulting set of functions. Such an approach produces unsatisfactory results for many multiple-output functions. In our new implementation, we use the technique suggested in [15] and [16], and treat the output part as a single *multiple-valued* variable.

The AOXMIN-MV gives better solutions and is much faster compared to AOXMIN. For example, it takes just 4 minutes to find the 18 product-terms solution for t481 benchmark, while AOXMIN needed 8 hours to compute it. Furthermore, AOXMIN-MV gives a solution to a more general problem of minimization of multiple-valued input, binary-valued output logic functions.

The paper is organized as follows. Section 2 describes the basic notation and definitions which are used in the sequel. Section 3 presents the AOXMIN-MV algorithm. In Section 4 the algorithm is demonstrated on the example of a 2-bit multiplier. Section 5 includes the experimental results. In the final section, some conclusions are drawn and directions for further research are proposed.

2 Preliminaries

We use the standard definitions and notation in the area of logic synthesis ([12], [17]). The most important notions are briefly summarized in this section.

2.1 Multiple-valued input binary-valued output functions

A *multiple-valued input, binary-valued output function* $f(x_1, \ldots, x_n)$ is a mapping $f : P_1 \times P_2 \times \ldots \times P_n \rightarrow B$, where the sets $P_i = \{0, 1, \ldots, p_i - 1\}$ for $i \in \{1, \ldots, n\}$, represent the sets of p_i values the variable x_i may assume, and $B \in \{0, 1, *\}$. $*$ denotes a don't care value.

An n-input m-output switching function can be represented by a multiple-valued input binary-valued output function of $n + 1$ variables, where $p_i = 2$ for $i \in \{1, \ldots, n\}$, and $p_{n+1} = m$. It is easily proven that the boolean minimization problem for multiple-output functions is equivalent to the minimization of a multiple-valued function of this form.

A point in the domain $P_1 \times P_2 \times \ldots \times P_n$ of the function is called a *minterm*. The *on-set* F, the *don't care-set* D and the *off-set* R of f are the sets of minterms that are mapped by f to 1, $*$ and 0, respectively.

Let x_i be a variable taking values from the set P_i, and let $S_i \subseteq P_i$. Then $x_i^{S_i}$ represent the characteristic function

$$x_i^{S_i} = \begin{cases} 1 & \text{if } x_i \in S_i \\ 0 & \text{otherwise} \end{cases}$$

$x_i^{S_i}$ is called a *literal* of variable x_i. The complement of a literal $x_i^{S_i}$ (written $\overline{x}_i^{S_i}$) is the literal $x_i^{\overline{S}_i}$.

A *product-term* is a Boolean product (AND) of one or more literals. If a product-term evaluates to 1 for a given minterm, the product-term is said to *contain* the minterm. A *sum-of-products* is a Boolean sum (OR) of product-terms.

2.1.1 Operations on product-terms

Let $S = x_1^{S_1} x_2^{S_2} \ldots x_n^{S_n}$ and $T = x_1^{T_1} x_2^{T_2} \ldots x_n^{T_n}$ denote product-terms.

The *complement* of a product-term S, denoted by $\overline{S}$ is the sum-of-products $\cup_{i=1}^{n} \overline{x}_i^{S_i}$.

The *intersection* of product-terms S and T, denoted by $S \cap T$, is the product-terms $x_1^{S_1 \cap T_1} x_2^{S_2 \cap T_2} \ldots x_n^{S_n \cap T_n}$ which is the largest product-term contained in both S and T. If $S_i \cap T_i = \emptyset$ for some i, then $S \cap T = \emptyset$, and S and T are said to be *disjoint*. Otherwise, they *intersect*.

The *union* of product-terms S and T, denoted by $S \cup T$, is the product-term $x_1^{S_1 \cup T_1}$ $x_2^{S_2 \cup T_2} \ldots x_n^{S_n \cup T_n}$ which is the smallest product-term containing both S and T.

The *exclusive-or* of two S and T, denoted by $S \oplus T$, is the product-term $x_1^{R_1} x_2^{R_2} \ldots x_n^{R_n}$, where $R_i = (\overline{S}_i \cap T_i) \cup (S_i \cap \overline{T}_i)$, for $i \in \{1, \ldots, n\}$.

2.1.2 Positional cube notation

Let $x_1^{S_1} x_2^{S_2} \ldots x_n^{S_n}$ be a product-term. It can be represented by a binary vector:

$$c_1^0 c_1^1 \ldots c_1^{p_1 - 1} - c_2^0 c_2^1 \ldots c_2^{p_2 - 1} - \ldots c_n^0 c_n^1 \ldots c_n^{p_n - 1}$$

where $c_i^j = 0$ if $j \notin S_i$, and $c_i^j = 1$ otherwise. Such a notation is called a *positional cube* or simply a *cube* [18]. A cube is a convenient representation for product-terms, and we often use the terms *cube* and *product-term* interchangeably.

We use c_i to represent the binary vector $c_i^0 c_i^1 \ldots c_i^{p_i - 1}$. The notation $c_i \cup d_i$ refers to the bitwise OR of two binary vectors, $c_i \cap d_i$ refers to the bitwise AND, and $\overline{c}_i$ denotes the bitwise complement of a binary vector.

A sum-of-products is represented by a set of cubes, also called a *cover*. The *size* of a cover is the number of cubes in it. We denote the size of a cover F by $|F|$. The complement of a cover is the intersection of the complements for each cube of the cover. The intersection (union or exclusive-or) of two covers is defined as the union of the pairwise intersection (union or exclusive-or) of the cubes from each cover.

The cube provides a convenient data structure for computer implementation of the algorithm where one bit is used for each part of the cube. Boolean operations on cubes are performed as word-wide operations (e.g. the intersection of two cubes is the cube which results from the component-wise Boolean AND of two cubes), which is more efficient than manipulating the binary vectors element by element.

3 The algorithm AOXMIN-MV

In this section we describe the structure of the AOXMIN-MV algorithm. The pseudocode is shown in Figure 1.

AOXMIN-MV 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 either $g_1 \oplus g_2$ or the complement of $g_1 \oplus g_2$ is a cover for F. The objective is to minimize the total number of cubes in g_1 and g_2.

AOXMIN-MV(F, D, R)
input: on-set F, don't care set D and off-set R of f
output: sets of cubes g_1, g_2 and $i \in \{0, 1\}$ such that $(g_1 \oplus g_2) \supseteq F$, if $i = 0$, and $(\overline{g_1 \oplus g_2}) \supseteq F$, if $i = 1$, total number of cubes in g_1 and g_2.

```
/* Initialization */
F ← Cover(F, D, R);
R ← Cover(R, D, F);
if (|F| < |R|)
        initial_cost = |F|;
else
        initial_cost = |R|;
best_cost = initial_cost;

/* Checking whether f is likely to have a compact AND-OR-XOR form */
(F1, F2, NF, g1, g2) ← Check(F, R);
if (|g1| + |g2| < best_cost)
        (best_g1, best_g2, best_cost, i) = (g1, g2, |g1| + |g2|, 0);

(R1, R2, NR, g1, g2) ← Check(R, F);
if (|g1| + |g2| < best_cost)
        (best_g1, best_g2, best_cost, i) = (g1, g2, |g1| + |g2|, 1);

/* Main minimization loop */
if (best_cost < initial_cost) {
        if (NF > 2)
                (best_g1, best_g2, best_cost, i) ← GroupMigration(F1, F2, NF, R, best_cost, 0);

        if (NR > 2)
                (best_g1, best_g2, best_cost, i) ← GroupMigration(R1, R2, NR, F, best_cost, 1);

        return(best_g1, best_g2, best_cost, i);
}
else
        return("function is unlikely to have a compact AND-OR-XOR form");
```

Figure 1: Pseudocode of the **AOXMIN-MV** algorithm.

First, the covers for the on-set and off-set of the input function are computed by employing the subroutine **Cover()**. **Cover()** implements Reduce(), Expand() and Irredundant() subroutines of Espresso [12] to comprise a single pass of the minimization algorithm. If $|F| < |R|$, then the cost is initialized to $|F|$. Otherwise, it is set to $|R|$.

Check(F, R)
input: on-set F and off-set R of f
output: sets of cubes F_1 and F_2 partitioning F, number of equivalence classes N_F of F, and two sets of cubes g_1 and g_2, such that $(g_1 \oplus g_2) \supseteq F$.

$N_F \leftarrow$ **DivideEqClasses**(F)*;*
$(F_1, F_2) \leftarrow$ **RandomPartitioning**(F, N_F)*;*
$(g_1, g_2) \leftarrow$ **Obtain**(F_1, F_2, R)*;*
return(F_1, F_2, N, g_1, g_2)*;*

Figure 2: Pseudocode of the **Check()** subroutine.

The next step is to check whether the input function is likely to have a compact AND-OR-XOR form. This is done by the subroutine **Check()**, whose pseudocode is shown in Figure 2. As a preprocessing step, **Check()** performs clustering of the cubes $c_i \in F$ into equivalence classes with respect to the equivalence relation $\mathcal{R} \stackrel{df}{=} \hat{\mathcal{R}}^+$, where $\hat{\mathcal{R}}^+$ is the transitive closure of $\hat{\mathcal{R}}$, and $\hat{\mathcal{R}}$ is defined by

$$(c_i, c_j) \in \hat{\mathcal{R}} \text{ iff } (c_1 \cap c_2 \neq \emptyset)$$

Two cubes are in relation $\mathcal{R}$ either when they intersect, or when they are connected through a chain of intersecting cubes. Thus, the equivalence classes of $\mathcal{R}$ form a partition of a set of cubes into connected chains of cubes. The clustering is performed by **DivideEqClasses()** subroutine, implementing a classical algorithm for computing a transitive closure [19]. **DivideEqClasses()** takes a set of cubes as its input, computes which cubes are connected, and labels each cube with the number of the equivalence class to which it belongs. It returns the total number of N of the classes in the set (for pseudocode of **DivideEqClasses()**, see [8]).

After clustering, the equivalence classes are randomly partitioned by **RandomPartitioning()** into two groups, F_1 and F_2, such that $F_1 \cup F_2 = F$ and $F_1 \cap F_2 = \emptyset$. F_1 and F_2 are used by the the subroutine **Obtain()** to construct two different sets of cubes, g_1 and g_2, satisfying $g_1 \oplus g_2 = F$, which would possibly be of a total size smaller than $|F|$.

Obtain(F_1, F_2, R)
input: sets of cubes F_1 and F_2, partitioning F and off-set R of f.
output: sets of cubes g_1 and g_2, such that $g_1 \oplus g_2 = F$

$g_1 \leftarrow$ **Cover**(F_1, R, F_2)*;*
$g_2 \leftarrow F_2 \cup (R \cap g_1)$*;*
$g_2 \leftarrow$ **Cover**$(g_2, \emptyset, F_1)$*;*
return(g_1, g_2)

Figure 3: Pseudocode of the **Obtain()** subroutine.

Obtain() uses the following property: *If $g_1 \oplus g_2 = F$ than, for any cube $c \in R$, $(g_1 \cup c) \oplus (g_2 \cup c) = F$.* This property trivially follows from the properties of the XOR operation. The two basic steps of **Obtain()** are:

1. Obtain g_1 such that $F_1 \subseteq g_1 \subseteq F_1 \cup R,\ g_1 \cap F_2 = \emptyset$ and $|g_1| < |F_1|$.
2. Obtain g_2 such that $g_1 \oplus g_2 = F$.

The first step can be converted to the problem of finding a cover for the incompletely specified function with the on-set F_1, the don't care set R and the off-set F_2. The cover is computed by employing **Cover()**.

Next, we determine which cubes from R are specified to 1 in the obtained cover, by computing the intersection $R \cap g_1$. Finally, we invoke **Cover()** to compute a cover for the completely specified function with the on-set $F_2 \cup (R \cap g_1)$ and the off-set F_1.

Let us prove that the relation $g_1 \oplus g_2 = F$ holds:

$$
\begin{array}{lll}
1) & F_1 \oplus F_2 = F & \{\text{since } F_1 \cup F_2 = F, F_1 \cap F_2 = \emptyset\} \\
2) & (F_1 \cup (g_1 \cap R)) \oplus (F_2 \cup (g_1 \cap R)) = F & \{\text{from the above property}\} \\
3) & g_1 \oplus (F_2 \cup (g_1 \cap R)) = F & \{F_1 \subseteq g_1 \subseteq F_1 \cup R\} \\
4) & g_1 \oplus g_2 = F & \{g_2 = F_2 \cup (g_1 \cap R\}
\end{array}
$$

Clearly, $g_1 \oplus g_2 = F$ implies $(g_1 \oplus g_2) \cap R = \emptyset$.

Check() is invoked first starting from the on-set F, and then starting from the off-set R of f. Each time the best solution is updated. If neither of the costs obtained is smaller than the *initial_cost*, than it is assumed that the function is unlikely to have a compact AND-OR-XOR form, and the algorithm terminates. Otherwise, the subroutine **GroupMigration()** is called first using F_1 and F_2 as the initial partition, and then using R_1 and R_2 as the initial partition.

GroupMigration() implements a group migration algorithm [13] which is an extension of Fiduccia/Mattheyses iterative improvement algorithm [14]. Group migration algorithm repeats the following: given an initial partitioning of objects into two groups, for each object determine the decrease in cost if the object were moved to the other group. Then, move the object that produces the greatest decrease or smallest increase in cost and mark it as *moved*. After all objects have been moved once, the lowest-cost partitioning is selected. If this partitioning has a higher cost then the initial one, then the algorithm stops. Otherwise it iterates taking the new partitioning for the initial partitioning.

The pseudocode shown in Figure 4 details a group migration algorithm for improving an initial partitioning of objects into two groups. In our case, objects are equivalence classes of F, computed by **DivideEqClasses()** and partitioned between the sets F_1 and F_2. A procedure **Move**$(F_1, F_2, \text{class}_j)$ returns a new partition of classes between F_1 and F_2 obtained by moving all the cubes belonging to a given equivalence class class_j, $j \in \{1, \ldots, N_F\}$, from the set F_k, $k \in \{1, 2\}$, to the set F_l, $l \in \{1, 2\}$, $k \neq l$. Each equivalence class has flag, *moved*, which indicates whether it has been moved or not. The variable *best-move_class* is the equivalence class that, when moved, yields the best cost improvement, and *bestmove_cost* is the resulting cost. Our experimental results show that the number of times the **do**-loop repeats in usually less than three.

GroupMigration(F_1, F_2, R, *best_cost*, N)
input: sets of cubes F_1 and F_2, partitioning on-set F, the number N_F of equivalence classes in F, off-set R, current best cost and an integer i
output: sets of cubes *best_*g_1 and *best_*g_2, such that $best\_g_1 \oplus best\_g_2 = F$, updated best cost, and an integer i.

```
do {
/* Initialization */
    init_cost = best_cost;
    for (each class_j ∈ F1 ∪ F2)
        class_j.moved = false;

    /* Create a sequence of N_F moves */
    for (i from 1 to N_F) {
        bestmove_cost = ∞;
        for (each class_j not class_j.moved) {
            (F1, F2) ← Move(F1, F2, class_j);
            (g1, g2) ← Obtain(F1, F2, R);
            if (|g1| + |g2| < bestmove_cost)
                bestmove_cost = |g1| + |g2|;
                bestmove_class = class_j;
            }
            if (bestmove_cost < best_cost) {
                (bestpart_F1, bestpart_F2) = (F1, F2);
                (best_g1, best_g2, best_cost, i) = (g1, g2, bestmove_cost, i);
            }
        }
        (F1, F2) ← Move(F1, F2, bestmove_class);
        bestmove_class = bestmove_class.moved;
    }

    /* Update (F1, F2) if a better cost was found, else exit */
    if (best_cost < init_cost)
        (F1, F2) = (bestpart_F1, bestpart_F2);
    else
        return(best_g1, best_g2, best_cost, i);
} while(best_cost < init_cost);
```

Figure 4: Pseudocode of the **GroupMigration()** subroutine.

GroupMigration() is called only if the number of equivalence classes, obtained by **DivideEqClasses()**, is greater than two. Otherwise, F_1 and F_2 have at most one class each and moving it to the opposite group will cause one of F_1 and F_2 to become the empty set. If F_1 is an empty set, then g_1 is an empty set, and thus $g_2 = F$. Therefore, $|g_1| + |g_2| = |F|$ and invoking **GroupMigration()** for the case $N \leq 2$ brings no reduction in the total size of g_1 and g_2.

4 Example: a 2-bit multiplier

In this section, we demonstrate the algorithm AOXMIN-MV on the example of a 2-bit multiplier. The input of the algorithm is a specification of the 2-bit multiplier in Espresso format, shown in Figure 5.

```
.i 4
.o 4
1010    1000
1001    0100
1011    1100
0110    0100
0101    0010
0111    0110
1110    1100
1101    0110
1111    1001
.e
```

Figure 5: Specification of a 2-bit multiplier in Espresso format.

First, AOXMIN-MV applies **Cover()** to compute the minimal covers for the on-set and off-set of the function. The resulting covers F and R are shown below (in positional cube notation, with "-" separating positions in the cube). Recall, that the output part of the cube is treated as a single multiple-valued variable (4-valued variable for the 4-output 2-bit multiplier).

F	R
01-11-01-11-1000	11-11-11-10-0011
01-11-10-01-0100	11-10-11-11-0011
10-01-01-01-0110	01-11-01-11-0010
11-01-01-10-0100	01-01-01-01-0100
11-01-10-01-0010	10-11-11-11-1001
01-01-01-01-0001	10-11-10-11-1100
01-10-11-01-0100	10-10-11-11-1111
	11-10-11-10-0111
	11-11-10-11-1001
	11-11-10-10-1111

Since $|F| < |R|$, *initial_cost* = 7. Next, the subroutine **Check()** is invoked, first starting from the the cover F. The cubes in F are divided into equivalence classes in the following way:

01-11-01-11-1000	class 1
01-11-10-01-0100	class 2
10-01-01-01-0110	class 3
11-01-01-10-0100	class 4
11-01-10-01-0010	class 5
01-01-01-01-0001	class 6
01-10-11-01-0100	class 2

Thus, $N_F = 6$. The generated equivalence classes are randomly partitioned into two groups as follows:

group 1:	classes 1, 3, 4 and 5
group 2:	classes 2 and 6

The resulting sets of cubes F_1, F_2 are

F_1	F_2
01-11-01-11-1000	01-10-11-01-0100
10-01-01-01-0110	01-11-10-01-0100
11-01-01-10-0100	01-01-01-01-0001
11-01-10-01-0010	

Next, **Obtain()** is invoked to compute the cost of the resulting partitioning. First, **Cover()** is applied to compute the cover for the function with the on-set F_1, the don't care set R and the off-set F_2. The resulting cover g_1 is as follows:

g_1
11-11-11-11-1010
11-01-01-11-0100

Now, it is determined which cubes from R are specified to 1 in the above g_1, by computing the intersection $R \cap g_1$. For the positional cube representation, the intersection of two cubes can be computed as component-wise Boolean AND of the two cubes. The resulting (non-empty) cubes are shown below.

$R \cap g_1$
11-11-11-10-0010
11-10-11-11-0010
01-11-01-11-0010
01-01-01-01-0100
10-11-11-11-1000
10-11-10-11-1000
10-10-11-11-1010
11-10-11-10-0010
11-11-10-11-1000
11-11-10-10-1010

These cubes are added to F_2, and the function with the on-set $F_2 \cup (R \cap g_1)$, the don't care set $\emptyset$ and the off-set F_1 it is minimized by **Cover()**. The resulting cover is:

g_2
11-10-11-11-0010
11-11-11-10-0010
10-11-11-11-1000
11-11-10-11-1000
01-11-11-01-0100
01-01-01-01-0011

The cost of the cover is equal to the total number of cubes in g_1 and g_2, i.e. it is 8.

Next, the above steps are repeated starting from R. The cubes from R are divided into equivalence classes in the following way:

11-11-11-10-0011	class 1
11-10-11-11-0011	class 1
01-11-01-11-0010	class 1
01-01-01-01-0100	class 2
10-11-11-11-1001	class 1
10-11-10-11-1100	class 1
10-10-11-11-1111	class 1
11-10-11-10-0111	class 1
11-11-10-11-1001	class 1
11-11-10-10-1111	class 1

Since $N_R = 2$, there is a unique partitioning for them, namely:

R_1	R_2
01-01-01-01-0100	11-11-10-10-1111
	11-11-10-11-1001
	11-10-11-10-0111
	10-10-11-11-1111
	10-11-10-11-1100
	10-11-11-11-1001
	01-11-01-11-0010
	11-10-11-11-0011
	11-11-11-10-0011

The results of the computation after invoking of **Obtain()** are shown below.

g_1	$F \cap g_1$	g_2
11-01-01-11-0100	10-01-01-01-0100	11-10-11-11-0011
	11-01-01-10-0100	11-11-10-11-1001
		01-11-01-11-0010
		11-11-11-10-0111
		10-11-11-11-1101

Since $|g_1| + |g_2| = 6$, the obtained cost is smaller than the current best cost. Thus, the best cost is updated to 6 and the main minimization loop starts. Since $N_F > 2$, **GroupMigration**$(F_1, F_2, R, 6, 0)$ is called. The equivalence classes of F are subsequently moved to the opposite group by **Move**$(F_1, F_2, \text{class}_j)$ and then the cost of the move is computed by **Obtain()**. After all classes have been marked as *moved*, the resulting *best_cost* remains 6. Since $N_R = 2$, the algorithm terminates returning the above g_1 and g_2 as the best solution. It can be easily checked that $\overline{(g_1 \oplus g_2)} = F$.

Table 1: Comparison with Espresso -*Dopo* and with AOXMIN-MV -*long*.

Example function	n	m	Espresso -*Dopo* p^e	Espresso -*Dopo* t,sec	AOXMIN-MV $\|g_1\|$	AOXMIN-MV $\|g_2\|$	AOXMIN-MV t_1,sec	AOXMIN-MV -*long* $\|g_1\|$	AOXMIN-MV -*long* $\|g_2\|$	AOXMIN-MV -*long* t_2,sec	impr.	t_2/t_1
5xp1	7	10	64	0.21	8	29	40	8	29	40	0	1
alu4	14	8	359	34	9	210	243	9	210	243	0	1
b9	16	5	119	0.78	12	31	118	12	31	118	0	1
clip	9	5	95	0.99	5	77	17	5	77	17	0	1
dist	8	5	109	1.2	7	85	23	7	85	23	0	1
ex7	16	5	119	0.76	12	31	119	12	31	119	0	1
f51m	8	8	76	0.90	6	27	55	6	27	55	0	1
life	9	1	84	0.42	20	40	200	20	40	200	0	1
radd	8	5	61	0.24	6	14	15	6	14	15	0	1
rd53	5	3	19	0.01	4	13	13	4	13	13	0	1
rd73	7	3	83	0.11	13	49	494	13	49	494	0	1
rd84	8	4	192	0.46	21	109	1566	21	109	1566	0	1
root	8	5	49	0.41	1	51	3.3	1	51	3.3	0	1
t481	16	1	481	1.6	9	9	218	9	9	218	0	1
xor5	5	1	12	0.01	2	4	7.3	2	4	7.3	0	1
z4	7	4	45	0.11	4	14	9.3	4	14	9.3	0	1
in2	19	10	136	2.3	136	0	1.1	5	115	1053	0.12	957.3
t1	21	23	85	5.1	89	0	5.9	8	75	101	0.07	17.1
tial	14	8	359	18	361	0	4.2	6	296	406	0.16	96.7
x9dn	27	7	116	13	120	0	1.2	1	112	179	0.06	149.2
amd	14	24	66	2.5	68	0	1.1	68	0	361	0	328.2
b2	16	17	106	3.6	110	0	3.9	110	0	1669	0	427.9
b10	15	11	100	1.6	100	0	0.92	100	0	80	0	87.0
bc0	26	11	185	6.6	185	0	4.1	185	0	487	0	118.8
bench1	9	9	139	7.0	140	0	3.2	140	0	382	0	119.4
cordic	23	2	155	63	163	0	12	163	0	12	0	1
duke2	22	29	86	4.4	87	0	3.2	87	0	8.6	0	1.8
ex1010	10	10	279	32	324	0	17	324	0	9584	0	563.8
exam	10	10	58	7.3	59	0	2.6	59	0	11	0	4.2
gary	15	11	107	1.6	108	0	1.2	108	0	204	0	170.0
in0	15	11	106	1.4	108	0	1.1	108	0	183	0	166.4
in1	16	17	106	3.6	110	0	0.9	110	0	1670	0	1855.6
in5	24	14	62	5.4	62	0	2.2	62	0	63	0	28.6
misex3	14	14	189	59	207	0	20	207	0	1300	0	65.0
misex3c	14	14	199	19	196	0	6.2	196	0	1195	0	192.7
p1	8	18	48	2.3	48	0	1.1	48	0	11	0	10.0
ryy6	16	1	112	0.17	7	1	0.18	7	1	0.18	0	1
sao2	10	4	37	0.13	38	0	0.83	38	0	30	0	36.1
shift	19	16	100	0.34	100	0	0.29	100	0	479	0	1651.7
sym10	10	1	210	1.9	130	1	0.76	130	1	1.9	0	2.5
t2	17	16	53	1.8	53	0	0.87	53	0	63	0	72.4
table3	14	14	175	20	175	0	18	175	0	77921	0	3896.1
table5	17	15	158	30	158	0	19	158	0	59620	0	2293.1
ts10	22	16	128	0.46	128	0	0.64	128	0	1589	0	2482.8
vg2	25	8	110	9.3	110	0	1.7	110	0	81	0	47.6
x1dn	27	6	110	7.2	110	0	1.3	110	0	49	0	37.8
average	14	11	127	8.1	85	17	71.2	71	47	3520.3	0.009	345.6

5 Experimental results

We have applied the algorithm AOXMIN-MV to a set of benchmark functions. The results were compared to the solutions produced by the two-level AND-OR minimizer Espresso [12], to the results of the 3-level AND-OR-XOR minimizers reported in [7] and [9], and to the results of the previous version of the algorithm, AOXMIN [8]. AOXMIN-MV, AOXMIN and Espresso were run on a Sun Ultra 60 operating two 360 MHz CPU and 768 Mb memory.

The purpose of the first experiment was to evaluate how good the estimation subroutine **Check()** is. To do this, we have added to the AOXMIN-MV the option *-long*. If AOXMIN-MV is run with this option, when the checking **if** (*best_cost* $<$ *initial_cost*) is skipped, and **GroupMigration()** is invoked for the case *best_cost* $\geq$ *initial_cost* as well. Thus, a comparison of the runs of AOXMIN with and without *-long* option gives us an idea of whether we would get a better result if we would perform **do**-loop for the case *best_cost* $\geq$ *initial_cost* as well. Table 1 shows the results of the comparison of AOXMIN-MV and AOXMIN-MV *-long* in terms of the total number of products and in the resulting g_1 and g_2 (columns 6, 7 and 9, 10), and the time taken in seconds (columns 8 and 11). The time is user time measured using the UNIX system command *time*. The table also shows the number of products p^e in the cover obtained by Espresso with output phase optimization (*-Dopo* option) and the time t to compute it (columns 3 and 4, respectively). Columns 2 and 3 give the number of inputs n and the outputs m of the benchmark functions.

To make the comparison more clear, we have splitted the table into 3 parts: the upper part shows the benchmarks which successfully passed **if** (*best_cost* $<$ *initial_cost*) checking of AOXMIN-MV and the lower two parts show the benchmarks which did not pass. Clearly, in the first case, **GroupMigration()** is invoked for both AOXMIN-MV and AOXMIN-MV *-long*, and therefore they produce the same results in terms of products and time. The middle part shows the benchmarks for which AOXMIN-MV *-long* found the solution with a smaller number of cubes than AOXMIN-MV, i.e. these are incorrectly estimated cases. The last too columns of the table show the improvement of AOXMIN-MV *-long* over AOXMIN-MV, computed as $1 - \frac{(|g_1|+|g_2|)_{\text{AOXMIN-MV -long}}}{(|g_1|+|g_2|)_{\text{AOXMIN-MV}}}$ and the number of times AOXMIN-MV *-long* is slower than AOXMIN. The lower part of the table shows the cases when invoking **GroupMigration()** brought no improvement. One can see that AOXMIN-MV is 345 times faster on average than AOXMIN-MV *-long*, at the expense of only 0.9% more product-terms in the obtained solution.

Table 2 shows the results of the comparison of AOXMIN-MV and the previous version of the algorithm, AOXMIN [8]. AOXMIN takes as its argument an integer N_{iter}, determining the number of random partitionings to perform. For all benchmarks, we run AOXMIN for up to 50 iterations, and we show the lowest number of iterations for achieving the best result.

The upper part of the table shows the functions estimated as likely to have a compact AND-OR-XOR expression. AOXMIN is usually faster, because several random partitionings take less time than the group migration algorithm, but AOXMIN-MV produces better results. Furthermore, when running AOXMIN, we never know how many random partitionings we have to make to find the best solution. For most functions, it is too time-consuming

Table 2: Comparison with AOXMIN.

Example function	n	m	AOXMIN			AOXMIN-MV	
			$\|g_1\| + \|g_2\|$	t,sec	N_{iter}	$\|g_1\| + \|g_2\|$	t,sec
5xp1	7	10	42	2.26	10	37	40.47
clip	9	5	95	1.91	10	82	17.32
rd53	5	3	19	4.12	20	17	13.17
rd73	7	3	83	4.61	10	62	494.21
rd84	8	4	192	10.42	10	130	1566.52
t481	16	1	18	3143.31	50	18	218.16
xor5	5	1	12	2.16	20	6	7.25
alu4	14	8	447	19.38	1	219	243.03
duke2	22	29	87	4.32	1	87	3.21
ex1010	10	10	725	19.38	1	306	17.70
misex3c	14	14	197	13.07	1	196	6.2
table3	14	14	176	19.69	1	175	18.12
table5	17	15	158	20.69	1	158	19.44
bw	5	28	24	3.29	1	28	0.46

to run more than 50 partitionings. For example, 50 runs of AOXMIN takes 8 hours. Contrary, AOXMIN-MV needs just 4 mins to find the same solution.

The lower part of the table shows the functions estimated as unlikely to have a compact AND-OR-XOR expression. AOXMIN-MV is faster than a single run of AOXMIN, due to the fact that it uses **Cover()**, which is faster than Espresso used by AOXMIN. Since **Cover()** performs only a single pass of a minimization algorithm, the resulting cover may be larger, as in the case of *bw*. However, AOXMIN-MV may be much better utilization of common subterms, like in the case of *ex1010*.

Table 3: Comparison with the algorithm [7].

function	n	m	Algorithm [7]	AOXMIN-MV
			$\|g_1\| + \|g_2\|$	$\|g_1\| + \|g_2\|$
5xp1	7	10	47	37
9sym	9	1	73	73
clip	9	5	92	82
rd73	7	3	83	62
sao2	10	4	33	38
t481	16	1	364	18
adder 2-bit	4	3	7	7
adder 3-bit	6	4	26	11
adder 4-bit	8	5	37	20
adder 5-bit	10	6	79	39

Table 3 shows the results of the comparison of AOXMIN-MV and the algorithm [7]. The benchmark results for the algorithm [9] are taken from the reference [9]. Unfortunately,

the timing is not reported in [9].

The algorithm [7] first generates a minimal expansion of the function in terms of of XOR and AND operations (XOR of AND-terms). Some of the XOR's are then converted into OR's, and subsequently, a graph coloring technique is used for the minimization of XOR's in the final expansion. Our algorithm shows better results for all functions except $sao2$.

Table 4: Comparison with the algorithm [9]; t^1 is the user time on HP180, provided by the authors of [9]; t^2 is the user time measure in our experiments (on Sun Ultra 60).

function	n	m	Algorithm [9]		AOXMIN-MV	
			$\|g_1\| + \|g_2\|$	t^1,sec	$\|g_1\| + \|g_2\|$	t^2,sec
mlp4	8	8	75	1956.31	122	0.87
rd53	5	3	17	27.98	17	13.17
rd73	7	3	54	386.37	62	494.21
rd84	8	4	99	1319.94	130	1566.52
z4	7	4	22	433.91	18	9.53

Table 4 shows the results of the comparison of AOXMIN-MV and the algorithm [9]. The benchmark results and timing for the algorithm [9] were kindly provided by the authors of [9].

The algorithm [9] decomposes the function to the subfunctions of up to 5 variables and then apply an exact algorithm to find the AND-OR-XOR expression for the individual subfunctions. Finally, the AND-OR-XOR expression for the complete function is composed from these expressions. Clearly, for small functions, such an algorithm is likely produce a better solution than our algorithm. Unfortunately, it is infeasible for larger functions, because it is too time-consuming.

6 Conclusion

This paper presents a heuristic algorithm, AOXMIN-MV, for three-level AND-OR-XOR minimization. Compared to the previous version AOXMIN, reported in [8], the main novel features of AOXMIN-MV are: (1) an estimation metric, checking whether the input function is likely to have a compact AND-OR-XOR expression; (2) a different strategy for partitioning the on-set of the input function into two groups, based on an extension of Fiduccia/Mattheyses partitioning algorithm; (3) better utilization of common subterms, due to the treatment of the output part of a multiple-output function as a single multiple-valued variable. The experimental results show that, these modifications yield to a faster and more efficient algorithm.

Two major facets of our algorithm require further research. First, we are presently working on extending it to handle AND and OR cases. That would give us an algorithm targeting a minimal expression of the type shown in equation (1) for any binary operation "$\circ$". Such an algorithm could be used for multi-level logic optimization by being applied

recursively to the resulting solution until no more improvement is encountered. We are also investigating the possibility of integrating our algorithm with SIS [20].

Second, presently we use a very simple cost measure (number of products), which is might not be suitable for some FPGA architectures, like, for example, look-up table based architecture. We need to incorporate a more sophisticated cost measure, reflecting the specific of the target architecture.

Further work on the efficiency of the algorithm might also incorporate representation of multiple-valued input binary-valued output functions by Multiple-Valued Decision Diagrams (MDD) [21], and performing the basic operation of the algorithm directly on graphs. Since there is no direct correspondence between the size of the cover for the function and the size of the MDD that represents it, very large cube covers can be captured and efficiently manipulated using this representation.

Acknowledgment

The authors are grateful to Debatosh Debnath and Tsutomu Sasao from Kyushu Institute of Technology for providing us with the benchmark results and timing for the algorithm [9].

References

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

[2] ”The home page for programmable logic”, 1999. Xilinx Corporation, http://www. xilinx.com/products/products.htm.

[3] ”FLEX Devices”, 1999. Altera Corporation, http://www.altera.com/html/products/flex.html.

[4] ”APEX 20K Device Family”, 1999. Altera Corporation, http://www.altera.com/html/products/apex.html.

[5] 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.

[6] E. Dubrova, P. Ellervee ”A fast algorithm for three-level logic optimization”, *Proc. Int. Workshop on Logic Synthesis*, Lake Tahoe, May 1999, pp. 251-254.

[7] T. Sasao, ”A design method for AND-OR-EXOR three-level networks”, *Proc. Int. Workshop on Logic Synthesis*, Lake Tahoe, May 1995.

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

[9] D. Debnath, T. Sasao, "A heuristic algorithm to design AND-OR-EXOR three-level networks", *Proc. Asia and South Pacific Design Automation Conf. (ASP-DAC'98)*, Yokohama, Japan, Feb. 1998.

[10] E. V. Dubrova, D. M. Miller, J. C. Muzio, "Upper bound on number of product-terms in AND-OR-XOR expression of logic functions", *Electronics Letters*, vol. 31, 1995, pp. 541-542.

[11] D. Debnath, T. Sasao, "Exclusive-OR of two sum-of products expressions: simplification and an upper bound on the number of products", *Proc. 3rd International Workshop on the Applications of the Reed-Muller Expansion in Circuit Design* (1997), 45-60.

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

[13] D. Gajski, N. Dutt, A. Wu, S. Lin, *High Level Synthesis: Introduction to Chip and System Design*, Kluwer, 1992.

[14] C. M. Fiduccia, R. M. Mattheyses, "A linear time heuristic for improving network partitions", *Proc. 19th ACM/IEEE Design Automation Conference*, 1982, pp. 175-181.

[15] T. Sasao, "An application of multiple-valued logic to a design of programmable logic arrays", *Proc. 8th Int. Symp. Multiple-Valued Logic*, May 1978, pp. 65-72.

[16] R. Rudell, A. Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization", *IEEE Trans. on Computer-Aided Design*, vol. CAD-6, No. 5, Sept. 1987, pp. 727-749.

[17] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

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

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*, The MIT Press, Cambridge, England, 1997.

[20] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R.K. Brayton, A. Sangiovanni-Vincentelli, "Sequential circuit design using synthesis and optimization". *Proc. Int. Conf. Computer design*, 1992, pp. 328-333.

[21] D. M. Miller, "Multiple-valued logic design tools", *Proc. 23rd Int. Symp. Multiple-Valued Logic*, May 1993, pp. 2-11.