Technology Mapping Using Boolean Matching and Don't Care Sets

Frédéric Mailhot *Giovanni De Micheli*

Center for Integrated Systems
Stanford University
Stanford, CA 94305

Abstract

We describe a new approach to technology mapping where matchings are recognized by means of Boolean operations. The matching algorithm uses tautology checking based on Shannon decompositions. We show how to use the symmetry and unateness properties to speed-up the Boolean matching algorithm. We examine how *don't care* information can be used during Boolean matching. The algorithms have been implemented in program Ceres and tested on the 1989 MCNC benchmark circuits.

1 Introduction

Logic synthesis has been shown to be an effective means of designing logic circuits, especially for semi-custom designs. The computer-aided synthesis of a logic circuit involves two major steps: the optimization of a technology independent logic representation, using Boolean and/or algebraic techniques, and technology mapping. Technology mapping is the task of transforming an arbitrary multiple-level logic representation into an interconnection of logic elements from a given library of elements. Technology mapping is a very crucial step in the synthesis of semi-custom circuits for different technologies, such as sea-of-gates, gate-arrays or standard cells. The quality of the synthesized circuits, both in terms of area and performance, depends heavily on this step. For this reason, several approaches to technology mapping have been pursued and implemented in research and commercial mapping tools. Unfortunately, the mapping problem is a difficult one from a computational complexity stand-point. Therefore, rule-based technology mappers [6, 12] and heuristic algorithms have been proposed [2, 8, 11, 13, 14, 17].

In this paper, we consider an algorithmic approach to technology mapping problem that extends the pioneering work of Keutzer on Dagon [11] and of Detjens [8] and Rudell [19] on MIS. To put our work in perspective, we briefly summarize their approach.

Technology mapping consists of three major tasks. First Boolean networks are partitioned into an interconnection of single-output sub-networks, with the property that each internal vertex has unit outdegree (i.e. fanout). Then each sub-network is decomposed into an interconnection of two-input functions (e.g. AND,OR,NAND or NOR). Each sub-network is modeled by a directed acyclic graph (DAG), called *subject graph*. Finally each subject graph is covered by an interconnection of library cells.

Finding a cover of a subject graph that optimizes area or timing is a difficult problem. Keutzer proposed to represent library functions by trees and to use a dynamic programming technique for optimal covering, based on fast tree matching algorithms. A similar approach was used by Rudell and Detjens [8, 19]. Note that the overall area (and timing) of a mapped network depends on the partitioning, decomposition and covering tasks. However, good results were achieved by this approach and extensions based on DAG matching presented by Detjens [8] did not show substantial improvements.

We consider in this paper an approach to the technology mapping that uses network partitioning and decomposition techniques similar to those used in [11], with an improved covering algorithm. The covering algorithm described in this paper does not use the tree-based representation. Instead, it uses Boolean matching techniques based on Shannon decomposition [4] for recognizing whether a logic function can be implemented by a library cell. The rationale for this choice is that the representation of single-output networks by trees makes cumbersome (and in some cases impossible) the efficient mapping of logic functions that have multiple occurrences of some variables into networks of gates that have also multiple occurrences of some variables (e.g. exclusive ORs or majority functions). Boolean techniques support uniformly the description and the matching of any single-output library cell, independently of its functionality. In addition, Boolean matching techniques can take advantage of *don't care* information.

The importance of the use of *don't care* conditions in multiple-level logic synthesis is well recognized [1]. In this paper we consider *don't care* conditions that are specified at the network boundary and that arise from the network interconnection itself [16]. *Don't care* conditions are usually exploited to minimize the number of literals (or terms) of each expression in a Boolean network. While such a minimization leads to a smaller (and faster) implementation in the case of pluri-cell design style [5] (or PLA-based design), it may not improve the local area and timing performance in a cell-based design. For example, cell libraries exploiting pass-transistors might be faster and/or smaller than other gates having less literals. A pass-transistor based multiplexer is such a gate: assuming a function is defined by its *on* set $\mathcal{F}$ and its *don't care set* $\mathcal{DC}$:

$$\begin{aligned}\mathcal{F} &= (a+b)c \\ \mathcal{DC} &= \overline{b}\overline{c}\end{aligned}$$

then $(a+b)c$ is the representation that requires the least number of literals (3), and the corresponding logic gate is implemented by 6 transistors. On the other hand, $a\overline{b}+bc$ requires one more literal (4), but is implemented by only 4 pass-transistors, and is likely to be faster. Since Boolean minimization may lead to sub-optimal results, we propose to use directly the *don't care* conditions during Boolean matching in the search for the best implementation in terms of area (or timing).

This paper is organized as follows. We first show how to determine Boolean matchings and we show techniques for speeding-up the matching operation. We then examine how *don't care* conditions can be used in conjunction with Boolean matching. Eventually we present a covering algorithm based on Boolean matching and we conclude by showing the effectiveness of this technique on benchmark circuits.

2 Boolean Matching

Matching is the key operation of the technology mapping process. It identifies whether an element of the library can be used to implement a part of a given Boolean function. Matching can be formulated as checking the tautology between a given Boolean function, called the *target function*, and the set of functions representing a library element, for any permutation of its variables. We also consider the phase-assignment problem in connection with the matching problem, because they are closely interrelated in affecting the cost of an implementation. Finally, we include the *don't care set* of the target function during the matching operation.

We denote the target function by: $\mathcal{F}(x_1, \dots, x_n)$. It has n inputs and one output. We denote the phase of variable x_i by: $\phi_i \in \{0, 1\}$, where $x_i^{\phi_i} = x_i$ for $\phi_i = 1$, $x_i^{\phi_i} = \overline{x_i}$ for $\phi_i = 0$. We denote the *don't care set* of the target function by: $\mathcal{DC}(x_1, \dots, x_n)$. We denote the library by: $\mathcal{L} : \{\mathcal{G}_1, \dots, \mathcal{G}_m\}$. Its elements $\mathcal{G}$ are multiple-input single-output functions. We define the matching problem as follows:

Given a target function $\mathcal{F}(x_1, \dots, x_n)$, its *don't care set* $\mathcal{DC}(x_1, \dots, x_n)$, and a library element $\mathcal{G}(y_1, \dots, y_n)$, find an ordering $\{i, \dots, j\}$ and a phase assignment $\{\phi_1, \dots, \phi_n\}$, of the input variables of $\mathcal{F}$, such that either equation (1) or (2) is true:

$$\mathcal{F}(x_i^{\phi_i}, \dots, x_j^{\phi_j}) = \mathcal{G}(y_1, \dots, y_n) \tag{1}$$

$$\overline{\mathcal{F}}(x_i^{\phi_i}, \dots, x_j^{\phi_j}) = \mathcal{G}(y_1, \dots, y_n) \tag{2}$$

for each value of $(y_1, \dots, y_n)$ and each *care* value of $(x_i^{\phi_i}, \dots, x_j^{\phi_j}) \notin \mathcal{DC}$, i.e. equation (1) or (2) is a tautology for all minterms not in the *don't care set*.

If no such ordering and phase assignment exist, then the element $\mathcal{G}$ does not match the target function $\mathcal{F}$. Furthermore, if no element in the library $\mathcal{L} : \{\mathcal{G}_1, \dots, \mathcal{G}_m\}$ matches $\mathcal{F}$, then $\mathcal{F}$ cannot be covered by the library $\mathcal{L}$.

In other words, if we define the *NPN-equivalent* set of a function $\mathcal{F}$ as the set of all the functions obtained by input variable Negation, input variable Permutation and function Negation [18], we say that a function $\mathcal{F}$ matches a library element $\mathcal{G}$ when there exist a NPN-equivalent function which is tautological to $\mathcal{G}$ modulo the *don't care set.*

For example, any function $\mathcal{F}(y_1, y_2)$ in the set: $\{y_1 + y_2, \overline{y_1} + y_2, y_1 + \overline{y_2}, \overline{y_1} + \overline{y_2}, y_1 y_2, \overline{y_1} y_2, y_1 \overline{y_2}, \overline{y_1 y_2}\}$ can be covered by the library element: $\mathcal{G}(x_1, x_2) = x_1 + x_2$. Note that in this example $\mathcal{G}(x_1, x_2)$ has $n = 2$ inputs, and can match $n! \cdot 2^n = 8$ functions [10, 17].

2.1 A Simple Boolean Matching Algorithm

A Boolean match can be determined by verifying that there exists a matching of the input variables such that the target function $\mathcal{F}$ and the library function $\mathcal{G}$ are a tautology. Tautology can be checked by recursive Shannon decomposition [4]. The two Boolean expressions are recursively cofactored generating two decomposition trees. The two expressions are a tautology if they have the same logic value for all the leaves of the recursion that are not in the *don't care set*. This process is repeated for all possible orderings of the variables of $\mathcal{F}$, or until a match is found.

The matching algorithm is described by the recursive procedure *simple_boolean_match* shown in figure 1, which returns TRUE when the arguments are a tautology for some ordering. At level n of the recursion, procedure *simple_boolean_match* is invoked repeatedly with arguments the cofactors of the n^{th} variable of $\mathcal{G}$ and the cofactors of all the variables of $\mathcal{F}$ until a match is found, in which case the procedure returns TRUE. If no match is found, the procedure returns FALSE. The recursion stops when the arguments are constants, i.e. when all variables have been cofactored, in which case the procedure returns TRUE when the corresponding values match (modulo the *don't care* condition). Note that when a match is found, the sequence of variables used to cofactor $\mathcal{F}$ in the recursion levels 1 to N represents the order in which they are to appear in the corresponding library element. The algorithm is shown in figure 1.

Note that in the worst-case all permutations and phase assignments of the input variables are considered. Therefore, up to $n! \cdot 2^n$ different Shannon decompositions may be required for each match. The worst-case computational complexity of the algorithm make it practical only for small values of n.

```
simple_boolean_match(f,g,dc,var_list_f,var_list_g,which_var_g) {
    if (dc == 1) return(TRUE)
    if ( f and g are constant 0 or 1) return ( f == g )
    which_var_f = 1
    gvar = pick_a_variable(var_list_g,which_var_g)
    remaining_var_g = get_remaining(var_list_g,which_var_g)
    while ( which_var_f ≤ size_of(var_list_f)) {
        fvar = pick_a_variable(var_list_f,which_var_f)
        remaining_var_f = get_remaining(var_list_f,which_var_f)
        f0 = shannon_decomposition(f,fvar,0)
        f1 = shannon_decomposition(f,fvar,1)
        g0 = shannon_decomposition(g,gvar,0)
        g1 = shannon_decomposition(g,gvar,1)
        dc0 = shannon_decomposition(dc,fvar,0)
        dc1 = shannon_decomposition(dc,fvar,1)

        if (simple_boolean_match(f0,g0,dc0,
                remaining_var_f,remaining_var_g,which_var_g+1)
                and simple_boolean_match(f1,g1,dc1,
                remaining_var_f,remaining_var_g,which_var_g+1)) {
                return(TRUE) }
        else if (simple_boolean_match(f1,g0,dc0,
                remaining_var_f,remaining_var_g,which_var_g+1)
                and simple_boolean_match(f0,g1,dc1,
                remaining_var_f,remaining_var_g,which_var_g+1)) {
                return(TRUE) }
        which_var_f = which_var_f + 1 }
    return(FALSE) }
```

Figure 1: Simple Algorithm for Boolean Matching

2.2 Speeding Up Boolean Matching

We present in this section some techniques to speed-up the matching of completely specified functions. Uncompletely specified functions are dealt with in the following sections.

To increase the efficiency of the Boolean matching process, it is important to remark that the phase information of the unate variables is irrelevant to determine the matching. Therefore we define a transformation Υ that complements the input variables that are negative unate. Note that the phase information cannot be taken away from binate variables, where both the positive and negative phases are required to express $\mathcal{F}$. By using this transformation, we reduce the information required for the matching and therefore reduce also its computational cost. For example, any function $\mathcal{F}(y_1, y_2)$ in the set: $\{y_1 + y_2, \overline{y_1} + y_2, y_1 + \overline{y_2}, \overline{y_1} + \overline{y_2}, y_1 y_2, \overline{y_1} y_2, y_1 \overline{y_2}, \overline{y_1 y_2}\}$ can be represented by the set: $\{y_1 + y_2, y_1 y_2\}$.

As a result, we redefine the matching problem as follows:

Given a target function $\mathcal{F}(x_1, \dots, x_n)$ and a library element $\mathcal{G}(y_1, \dots, y_n)$, find an ordering $\{i, \dots, j\}$ and a phase assignment $\{\phi_k, \dots, \phi_l\}$ of the binate variables $\{k, \dots, l\}$ of $\mathcal{F}$, such that either (3) or (4) is true:

$$\Upsilon(\mathcal{F}(x_i, \dots, x_k^{\phi_k}, \dots, x_l^{\phi_l}, \dots, x_j)) \equiv \Upsilon(\mathcal{G}(y_1, \dots, y_n)) \tag{3}$$

$$\Upsilon(\overline{\mathcal{F}}(x_i, \dots, x_k^{\phi_k}, \dots, x_l^{\phi_l}, \dots, x_j)) \equiv \Upsilon(\mathcal{G}(y_1, \dots, y_n)) \tag{4}$$

The following considerations are also important in reducing the computational complexity:

- *Any input permutation must associate each unate (binate) variable in the target function to a unate (binate) variable in the function of the library element.*

- *Variables or groups of variables that are interchangeable in the target function must be interchangeable in the function of the library element.*

The first point implies that if the target function has m binate variables, then only $m! \cdot (n - m)!$ permutations of the input variables are needed.

The second point implies that symmetry classes can be used to simplify the search. A symmetry class is a set of variables that are interchangeable without affecting the logic functionality [15]. Techniques based on symmetry considerations to speed-up algebraic matching were also presented by Morrison in [17].

For a given function $\mathcal{F}(x_1, \dots, x_i, \dots, x_j, \dots, x_n)$, x_i and x_j belong to the same symmetry class if

$$\mathcal{F}(x_1, \dots, x_i, \dots, x_j, \dots, x_n) \equiv \mathcal{F}(x_1, \dots, x_j, \dots, x_i, \dots, x_n)$$

The symmetry property is an equivalence relation (it is reflexive, symmetric and transitive), hence if $\{x_i, x_j\}$ and $\{x_i, x_k\}$ are two symmetry sets, then $\{x_j, x_k\}$ is also a symmetry set.

Symmetry classes are used in two different ways to reduce the search space. First, they are used as a filter to quickly find good candidates for matching. A necessary condition for matching a target function $\mathcal{F}$ by library element $\mathcal{G}$ is that both have exactly the same symmetry classes. Hence only a small fraction of the library elements need be checked by the computationally intensive Boolean comparison to see if they match the logic equation. The symmetry classes for each library element are calculated once before invoking the mapping algorithm.

Second, symmetry classes are used during the Boolean comparison itself. Once a library element $\mathcal{G}$ that satisfies the previous requirement is found, the symmetry sets of $\mathcal{F}$ are compared to those of $\mathcal{G}$. Then only variables belonging to symmetry sets of the same size can possibly produce a match. Since all variables from a given symmetry set are equivalent, the ordering of the variables within the set is irrelevant. This implies that the permutations need only be done over symmetry sets of the same size. Thus the number of permutations required to detect a match is: $\prod_{i=1}^{q}(S_i!)$, where S_i is the number of sets of cardinality i, and q is the size of the largest symmetry set. Although in the worst case logic equations might have no symmetry at all, our experience with commercial standard cells and programmable logic devices libraries (such as CMOS3, LSI Logic or Actel) is that the elements are highly symmetrical, the average S_i being less than 2, as shown in Table 1. For example, the gate MX $(a\overline{x}\overline{y} + b\overline{x}y + cx\overline{y} + dxy)$ in the Actel library has 4 sets of single unate variables $(\{a\}, \{b\}, \{c\}, \{d\})$, and 2 sets of single binate variables $(\{x\}, \{y\})$.

Library	Average S_i	Maximum S_i
Actel	1.29	4
CMOS3	1.27	4
LSI Logic	1.32	4

Table 1: Average number of symmetry sets for different libraries

Unateness information and symmetry classes are used together to further reduce the search space. Unate and binate symmetry sets are distinguished, since both unateness and symmetry properties have to be the same for two variables to be interchangeable. Thus $S_i = S_i^u + S_i^b$, where S_i^u is the number of sets of cardinality i made of unate variables, S_i^b is the number of sets of cardinality i made of binate variables. This further reduces the number of permutations to $\prod_{i=1}^{q} S_i^u! \cdot S_i^b! = \prod_{i=1}^{q} S_i^u! \cdot (S_i - S_i^u)! < \prod_{i=1}^{q} S_i!$. Hence, when considering the phase assignment of the binate variables, at most $\prod_{i=1}^{q} S_i^u! \cdot (S_i - S_i^u)! \cdot 2^{i \cdot (S_i - S_i^u)}$ trials have to be made in order to find a match. In the Actel library, the worst case occurs for the library element *MXT* $= d_0 c_1 c_3 + d_1 \overline{c_1} c_3 + d_2 c_2 \overline{c_3} + d_3 \overline{c_2 c_3}$, where $S_1 = 7$, and $S_1^u = 4$. In that case, $4! \cdot 3! \cdot 2^3 = 1152 \ll 7! \cdot 2^7 = 645,120$, where $7! \cdot 2^7$ represents the number of trials needed if no symmetry information is used.

Procedure *boolean_match*, a variation on procedure *simple_boolean_match* is shown in figure 2. It incorporates the symmetry information to reduce the search space: permutations are done only over symmetry sets of the same size.

2.3 Use of the Don't Care Sets

When *don't care* conditions are considered, the target function $\mathcal{F}$ cannot be uniquely characterized by a symmetry set. Therefore the techniques based on symmetry sets presented in the previous section no longer apply.

A straight-forward approach is to consider all the functions $\mathcal{H}$ that can be derived from $\mathcal{F}$ and its *don't care* set $\mathcal{DC}$. Unfortunately, there are 2^N possible combinations, where N is the number of minterms in $\mathcal{DC}$. Therefore this approach can be used only for small *don't care* sets. For large *don't care* sets, a pruning mechanism has to be used to limit the search space.

We introduce now a representation of n-variable functions that exploits the notion of symmetry sets and NPN-equivalence and that can be used to determine matchings while exploiting the notion of *don't care* conditions. For a given number of input variables n, let $G(V, E)$ be a graph whose vertex

```
boolean_match(f,g,f_symmetry_sets,g_symmetry_sets) {
        if ( f and g are constant 0 or 1) {
                return ( f = g ) }
        if ( f_symmetry_sets is empty) {
                f_symmetry_sets = get_next_f_symmetry_set()
                symmetry_size = size_of(f_symmetry_sets)
                while ( symmetry sets of g with
                        size symmetry_size have still to be tried) {

                        g_symmetry_sets = get_next_available_set(g,symmetry_size)
                        boolean_match(f,g,f_symmetry_sets,g_symmetry_sets)
                        if ( it is a match) {
                                return(TRUE) } } }
        fvar = pick_a_variable(f_symmetry_sets)
        gvar = pick_a_variable(g_symmetry_sets)

        f0 = shannon_decomposition(f,fvar,0)
        f1 = shannon_decomposition(f,fvar,1)
        g0 = shannon_decomposition(g,gvar,0)
        g1 = shannon_decomposition(g,gvar,1)

        if ((boolean_match(f0,g0,f_symmetry_sets,g_symmetry_sets)
                and (boolean_match(f1,g1,f_symmetry_sets,g_symmetry_sets) {
                return(TRUE) }
        else return(FALSE) {
```

Figure 2: Algorithm for Fast Boolean Matching

set is in one-to-one correspondence with the ensemble of all different NPN equivalent functions, and $E = \{(v_i, v_j)\}$ such that the function represented by v_i and v_j differ in one minterm. Such a graph $G(V, E)$ for $n = 3$ is shown in figure 3.

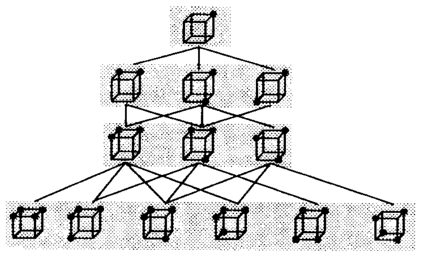

Figure 3: Matching compatibility graph for 3-variable Boolean space

Each vertex V in the graph is annotated with the library elements that matches the corresponding function, when a match exists. Then, for a given target function $\mathcal{F}$ the corresponding vertex $v_{\mathcal{F}}$ can be determined. There exists a matching to the cell $\mathcal{G}$ if there is a path in the graph $G(V, E)$ from $v_{\mathcal{F}}$ to $v_{\mathcal{G}}$ (possibly of zero length) whose edges corresponds to minterms in the *don't care set* of $\mathcal{F}$.

The graph $G(V, E)$ is called *matching compatibility* graph, because it shows which matching are *compatible* with the given function. Note that the size of the compatibility graph is small for functions of 3 or 4 variable, where there are 14 and 222 different NPN-equivalent functions respectively [18], representing the 256 and 65536 possible functions of 3 and 4 variables.

The compatibility graph is constructed automatically once, for a given value of n, and annotated with the library elements. The algorithm for graph traversal is shown in figure 4, and it is invoked with the vertex found by *boolean_matching* as parameter. The algorithm returns the list of all the matching library elements, among which the minimum-cost one is chosen to cover $\mathcal{F}$. The algorithm has shown to be practical for values of $n \leq 4$, because of the size of the compatibility graph.

3 A Covering Algorithm

The logic circuit to be mapped is partitioned into subject graphs $\{\Gamma_1, \ldots, \Gamma_K\}$, that are decomposed into an interconnection of two-input

```
traverse_graph(node,f,dc) {
        mark_visited(node)
        for (all minterms $\mu_i$ in dc) {
                remaining_dc = subtract(dc,$\mu_i$)
                new_f = add(node->equation,$\mu_i$)
                compatible_node = node_compatible(node,$\mu_i$)
                if (compatible_node is not visited) {
                        traverse_graph(compatible_node,new_f,remaining_dc) } }
        if (node is a library element) {
                if (node->library represents the best cost) {
                        update_best_cover(node->library) } } }
```

Figure 4: Algorithm for Reachability Graph traversal using *Don't Cares*

gates. We consider here the covering of a subjet graph Γ_i that optimizes some cost criteria (e.g. area or timing). For this purpose we use the notions of *cluster* and *cluster function*.

A *cluster* is a connected sub-graph of the subject graph Γ_i, having only one vertex with zero out-degree ν_j. It is characterized by its depth (longest directed path from ν_j) and its number of inputs. The associated *cluster function* is the Boolean function obtained by collapsing [3] the Boolean expressions associated to the vertices into a single Boolean function. We denote all possible clusters containing the vertex ν_j of Γ_i by $\{\kappa_{j,1}, \ldots, \kappa_{j,N}\}$.

As an example, consider the Boolean network (after an AND/OR decomposition):

$$\begin{aligned} f &= bx_1 \\ b &= e + x_2 \\ x_2 &= \bar{c} + d \\ x_1 &= a + c \end{aligned}$$

There are six possible *cluster functions* for the subject graph Γ_i containing the sink vertex $\nu_1 = f$:

$$\begin{aligned} \kappa_{1,1} &= bx_1 \\ \kappa_{1,2} &= b(a + c) \\ \kappa_{1,3} &= (e + x_2)x_1 \\ \kappa_{1,4} &= (e + x_2)(a + c) \\ \kappa_{1,5} &= (e + \bar{c} + d)x_1 \\ \kappa_{1,6} &= (e + \bar{c} + d)(a + c) \end{aligned}$$

The algorithm attempts to match each *cluster function* $\kappa_{j,k}$ to a library element. The area cost of a cover is computed by adding to the cost of the matching of the cluster $\kappa_{j,k}$ under consideration the cost of the clusters corresponding to the variables in the Boolean function for $\kappa_{j,k}$. For any cluster, there is always a match, because the network was decomposed into AND/ORs in the initial setup phase. When matchings exist for multiple clusters, then the choice of the matching of minimal area cost guarantees minimality of the total area cost of the matched sub-graph [11, 8], for the particular AND/OR decomposition under consideration. The cost of the required inverters is also taken into account at this stage.

The timing cost of a cover can be computed in a similar way. The propagation delay through a cluster is added to the maximum of the arrival times at its inputs, to compute the local time at the vertex ν_j [19]. When matching exist for multiple clusters, then the choice of the matching of minimal local time guarantees minimality of the total timing cost of the matched sub-graph, again for the particular AND/OR decomposition under consideration.

The covering algorithm is implemented by procedure *get_bigger_function* shown in figure 5.

As the subject graph is being mapped, the *don't care* set changes accordingly. For example, let us assume that the chosen matchings for b, x_1 and x_2 are $b = \text{OR3}(\bar{c}, d, e)$, $x_1 = \text{OR2}(a, c)$ and $x_2 = \text{OR2}(\bar{c}, d)$. Then the satisfiability *don't care* sets associated with these variables are:

$$\begin{aligned} \mathcal{DC}_b &= bc\bar{d}\bar{e} + \bar{b}(\bar{c} + d + e) \\ \mathcal{DC}_{x_1} &= x_1\bar{a}\bar{c} + \overline{x_1}(a + c) \\ \mathcal{DC}_{x_2} &= x_2c\bar{d} + \overline{x_2}(\bar{c} + d) \end{aligned}$$

These *don't care* sets can be used for Boolean matching in the rest of the circuit. For example, let's assume the cluster κ_2 of f is being processed. Then, its associated *cluster function* is $b(a + c)$. However, $\bar{b}\bar{c}$ is part of the *don't care* set $\mathcal{DC}_b$, and can be used during the matching. In this particular case, the functions $b(a + c)$ and $a\bar{b} + bc$ are both valid matches. For some technologies, the second option may be preferred, (i.e. a multiplexer may be better than an AND-OR gate).

```
get_bigger_function(top,equation,list,depth) {
                if (depth  = max_depth) {
                return
        }
        if (equation is empty) {
                if (top not yet mapped) {
                        equation = get_equation_from(top)
                        list = get_support_from(equation)
                        get_bigger_function(top,equation,list,1)
                        set_node_mapped(top)
                }
                return }
        while list is not empty {
                if (node(list) is not mapped) {
                        get_bigger_function(node(list),NULL,NULL,depth) }
                if (node(list) is a global input) skip this one
                else if (fanout(node(list)) > 1) {
                        get_bigger_function(node(list),NULL,NULL,depth) }
                else {
                        new_list = get_support_from(equation(node(list))
                        replace current list element by new_list
                        new_equation = merge(equation,node(list))
                        get_bigger_function(top,new_equation,new_list,depth+1)
                        put back list in original state }
                list = next_element_from(list) }
        check_if_in_library(top,equation,list)
        return }
```

Figure 5: Algorithm for network covering

Function	*Don't Care*	Library element
$a(b + c)$	$\bar{a}b$	AO32A,MX2
$a(b + c)$	$\bar{a}b$	OA32,MAJ3
$a(b + cd)$	$b(\bar{c}d + c\bar{d})$	OA32,AO1C,AO3
abc	$\bar{a}\bar{b}c$	AND3,XA1A
$ab(c + d) + \bar{b}\bar{c}\bar{d}$	$b\bar{c}\bar{d}$	OA2

Table 3: Compatible library elements (Actel)

4 Implementation and Results

The technology mapping algorithms have been implemented in a program called Ceres. Ceres reads the logic description of the circuits and of the library in a description language called SLIF (Structure and Logic Interchange Format) [7]. SLIF allows for the description of sequential networks, which are also mapped by program Ceres. Search limiting heuristics, under the control of the user, are also available to speed up the execution times. Ceres has been tested on DECstations 3100 and 3200, as well as on SUN workstations. We used the benchmarks circuits provided for the 1989 MCNC Logic Synthesis Workshop. Starting from the original description of the benchmarks, we compared the results of our technology mapping algorithm to the ones of the technology mapper built in MIS-II, release 2.1. We used three different libraries: Actel, LSI Logic and CMOS3. Area was chosen as the metric for the final implementation, counting inverters as well as other logic gates in the total cost.

Procedures *simple_boolean_match*, *boolean_match*, *traverse_graph* and *get_bigger_function* have been coded and tested in Ceres. However,

Circuit	Actel				LSI Logic				CMOS3			
	MISII		Ceres		MISII		Ceres		MISII		Ceres	
	cost	run time	cost	run time	cost	run time	cost	run time	cost	run time	cost	run time
9sym	209	854.4	204	84.7	421	51.5	415	44.1	7656	17.5	7512	20.8
misex1	44	57.5	44	8.4	89	23.9	105	8.3	1744	6.0	2024	6.2
misex2	72	55.0	73	56.1	146	28.4	147	15.0	2792	7.8	2800	12.2
rd53	62	69.4	60	15.2	106	25.9	122	21.1	2120	6.5	2272	18.6
xor5	35	46.6	34	13.6	63	22.4	68	10.5	1256	4.8	1280	15.9
clip	355	4938.7	358	108.5	670	81.7	726	75.8	12864	30.4	13624	69.8
bw	126	198.7	119	24.3	210	34.4	237	17.8	4104	14.2	4624	47.5
e64	781	278.4	745	174.2	1552	198.1	1566	161.1	30136	60.0	31312	35.3
vg2	320	291.1	356	62.7	600	72.2	635	66.5	11760	25.6	12480	61.6
sao2	197	125.3	191	55.6	339	47.0	362	50.9	6816	14.8	7152	45.7
o64	70	52.7	70	21.0	88	28.9	87	9.5	1752	7.9	1736	7.1
rd73	303	2027.5	360	123.1	598	72.5	630	37.0	11576	26.3	12072	65.1
con1	9	37.2	8	6.1	20	18.8	21	4.8	392	3.3	376	3.5
misex3c	542	3238.7	588	116.1	1015	120.8	1080	106.4	19624	47.5	21040	44.8
cm163a	20	39.5	21	18.0	42	20.1	34	15.7	728	4.0	600	13.5
decod	19	40.7	22	9.1	38	20.9	51	7.0	744	4.2	936	2.5
pcle	30	40.8	31	5.9	63	20.9	51	7.3	1136	4.6	936	4.9
cm82a	9	36.8	8	6.0	22	18.7	19	6.8	368	3.3	328	2.3
cmb	26	40.8	24	24.7	47	21.0	50	7.8	912	4.2	968	3.6
majority	5	34.8	5	29.5	10	16.6	16	5.4	176	3.3	272	3.2

Table 2: Mapping Results (total cost of mapped circuit is shown)

only procedure *boolean_match* is currently integrated within procedure *get_bigger_function*. Hence, results of technology mapping reported in table 2 do not reflect the use of *don't cares*. Examples of running procedure *traverse_graph* are reported separately in Table 3, and show how functions can match multiple library elements.

The run times on a DECstation 3100 ranged from a few seconds for *majority* to less than 3 minutes for *e64*. Table 2 summarizes our results. It is worth noting that the run times from program Ceres were consistently small, for the three libraries used. In particular, the run times using the Actel library, which has a wealth of XORs, multiplexers and majority functions [9], were comparable to the run times using the other two libraries, indicating that the Boolean matching algorithm used in Ceres handles these types of gates very efficiently.

5 Conclusions and Future Work

We have presented a different algorithmic approach to technology mapping. Boolean operations are used during the matching step, making it possible to recognize quickly such functions as exclusive ORs and majority functions. Different filters and simplifications have been proposed to reduce the search space that would otherwise be very large. Results have shown that this approach, in its present implementation, is competitive with other algorithmic methods based on tree or DAG covering.

We have studied the use of *don't cares* in connection with technology mapping. Our experiments have shown that the present techniques are limited to library cells of at most four variables. However, most library elements fit into this class. The experiments have also shown that the use of *don't cares* leads to a wider choice for matching library elements. Future work will include the full integration of this technique into Ceres.

Acknowledgements

The authors wish to thank David C. Ku for numerous and fruitful discussions. This work is supported in part by the National Science Foundation under grant MIP-8719546, by DEC, ATT and NSF under a PYI award, by a Stanford CIS seed grant, by the Natural Sciences and Engineering Council of Canada, and by the Quebec Fonds F.C.A.R.

References

[1] K.A. Bartlett, R.K. Brayton, G.D. Hachtel, R.M. Jacoby, C.R. Morrison, R.L. Rudell, A. Sangiovanni-Vincentelli and A.R. Wang, *Multilevel Logic Minimization Using Implicit Don't Cares*, IEEE Transactions on CAD, Vol. 7, No.6, pp.723-740, June 1988.

[2] M.R.C.M. Berkelaar and J.A.G. Jess, *Technology Mapping for Standard-Cell Generators*, Proceedings of the ICCAD, pp. 470-473, November 1988.

[3] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A.R. Wang, *MIS: A Multiple-Level Logic Optimization System*, IEEE Transactions on CAD, Vol. CAD-6, Nu. 6, pp.1062-1081, November 1987.

[4] R. K. Brayton, G. D. Hachtel, C. T. McMullen, A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 193 p. 1984.

[5] R. K. Brayton, R. Camposano, G. De Micheli, R. H. J. M. Otten and J. T. J. van Eijndhoven, *The Yorktown Silicon Compiler System*, in D.Gajski, *Silicon Compilation*, Addison Weseley, 1988.

[6] J. Darringer, D. Brand, J. Gerbi, W. Joyner and L. Trevillyan, *LSS: a System for Production Logic Synthesis*, IBM J. Res. Develop., September 1984.

[7] G.De Micheli "Algorithms for Synchronous Logic Synthesis" *International Workshop on Logic Synthesis* North Carolina, May 1989.

[8] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli and A. Wang, *Technology Mapping in MIS*, Proceedings of the ICCAD, pp.116-119, November 1987.

[9] A. El Gamal, J. Green, J. Reyneri, E. Rogoyski, K. A. El-Ayat and A. Mohsen, *An architecture for Electrically Configurable Gate Arrays*, IEEE Journal of Solid-State Circuits, Vol. 24, No. 2, pp.394-398, April 1989.

[10] G. Hachtel and M. Lightner, *Tutorial: Multi-Level Logic Synthesis*, ICCAD 1988.

[11] K. Keutzer, *DAGON: Technology Binding and Local Optimization by DAG Matching*, Proceedings of the 24^{th} ACM/IEEE Design Automation Conference, 1987.

[12] D. Gregory, K. Bartlett, A. deGeus, and G. Hachtel, *SOCRATES: A System for Automatically Synthesizing and Optimizing Combinational Logic*. Proceedings of the 23^{rd} ACM/IEEE Design Automation Conference, pp. 79-85, June 1986.

[13] M. C. Lega, *Mapping Properties of Multi-Level Logic Synthesis Operations*, Proceedings of the ICCD, pp.257-261, October 1988.

[14] R. Lisanke, F. Brglez, G. Kedem, *McMAP: A Fast Technology Mapping Procedure for Multi-Level Logic Synthesis*, Proceedings of the ICCD, pp. 252-256, October 1988.

[15] E. J. McCluskey, *Detection of Group Invariance or Total Symmetry of a Boolean Function*, Bell Syst. tech. J., V.35, pp. 1445-1453, November 1956.

[16] P. McGeer and R.K. Brayton, *Consistency and Observability Invariance in Multi-Level Logic Synthesis*, Proceedings of the International Workshop on Logic Synthesis, North Carolina, May 23-26 1989.

[17] C.R. Morrison R.M. Jacoby and G.D. Hachtel, *TECHMAP: Technology Mapping with Delay and Area Optimization*, in *Logic and Architecture Synthesis for Silicon Compilers*, G. Saucier and P.M. McLellan editors, North-Holland, pp.53-64, 1989.

[18] S. Muroga, *Threshold Logic and its Applications*, John Wiley, 1971, 478 p.

[19] R. Rudell, *Logic Synthesis for VLSI Design*, Ph.D. Dissertation, U.C. Berkeley, Memorandum UCB/ERL M89/49, 26 April 1989, 223 p.