Algorithms for Technology Mapping Based on Binary Decision Diagrams and on Boolean Operations

Frédéric Mailhot, *Member, IEEE*, and Giovanni De Micheli, *Senior Member, IEEE*

***Abstract*—We describe new algorithms and a new computer-aided design tool, called *Ceres*, for technology mapping of both completely specified and incompletely specified logic networks. The algorithms are based on Boolean techniques for *matching*, i.e., for the recognition of the equivalency between a portion of a network and library cells. A novel matching algorithm, using ordered binary decision diagrams, is described. It exploits the notion of symmetry for achieving higher computational efficiency. We also describe a matching technique that takes advantage of *don't-care* conditions, by means of a *compatibility* graph. We then present a strategy for timing-driven technology mapping, based on iterative improvement. Experimental results indicate these techniques generate good-quality solutions, and require short run times and limited memory space.**

I. Introduction

Logic SYNTHESIS has been shown to be an effective means of designing logic circuits, especially for semicustom 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. Logic optimization is used to modify the structure of a logic description, such that the final structure has a lower cost than the original [1], [2]. Logic optimization has traditionally been done before technology-dependent operations, and is assumed to have already taken place in the following.

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 semicustom 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.

The technology-mapping transformation implies two distinct operations: recognizing logic equivalence between two logic functions, and finding the best set of logically equivalent gates whose interconnection represents the original circuit. The first operation, called *matching*, involves equivalence checking and input assignment. Checking for logic equivalence can be expressed as a *tautology*, which has been proven to be coNP-complete [3]. Input assignment is also computationally complex. The second operation, called *covering*, involves finding an alternate representation of a Boolean network using logic elements that have been selected from a restricted set.

The two operations intrinsic to technology mapping, matching and covering, are computationally difficult. For this reason, several approaches to technology mapping have been pursued and implemented in research and commercial mapping tools. Rule-based technology mappers [4], [5] and heuristic algorithms have been proposed [6]–[11].

In this paper, we consider an algorithmic approach to the technology-mapping problem that extends the pioneering work of Keutzer on Dagon [8] and of Detjens [7] and Rudell [12] on MIS. To put our work in perspective, we briefly summarize their approach.

Most algorithmic approaches to technology mapping attack the problem by dividing it into sub-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., fan-out). 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 a *subject graph*. Finally, each subject graph is then covered by an interconnection of library cells, to produce the final circuit.

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 [7], [12]. 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 [7] did not show substantial improvements.

In this paper, we consider an approach to technology mapping that uses network partitioning and decomposition techniques similar to those used in [8], bit with dif-

Manuscript received August 15, 1991; revised July 2, 1992. This work was supported in part by the National Science Foundation under Grant MIP-8719546; by DEC, AT&T, and NSF under a PYI award; by ARPA under Grant J-FBI-88-101; by the Natural Sciences and Engineering Council of Canada; and by the Quebec Fonds F.C.A.R. This paper was recommended by Associate Editor R. K. Brayton.

F. Mailhot is with Synopsys, Mountain View, CA 94043. This work was performed when the author was at Stanford University.

G. De Micheli is with the Center for Integrated Systems, Stanford University, Stanford, CA 94305.

IEEE Log Number 9209622.

ferent matching and covering algorithms. The covering algorithm described in this paper does not use a tree-based representation. Instead, it used Boolean matching techniques based on binary decision diagrams [13], [14] to recognize 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 also have multiple occurrences of some variables (e.g., exclusive OR's or majority functions). Boolean techniques uniformly support the description and matching of any single-output library cell, independent of cell functionality. In addition, Boolean matching techniques can take advantage of *don't-care* information.

This paper is organized as follows: We first give a brief overview of the overall approach to technology mapping. We describe partitioning, decomposition, and covering. We then present an algorithm for Boolean matching, followed by a description of the Boolean techniques used during matching. The Boolean techniques use both completely specified and incompletely specified logic functions. We then show how the *don't-care* sets are derived, and their use in technology mapping. We comment on performance-oriented mapping using iterative improvement techniques. Finally, we present implementation issues and results.

II. Technology Mapping

In this section, we present the major tasks in technology mapping. We briefly overview partitioning and decomposition, and then present our network-covering algorithm.

2.1. Partitioning

Partitioning is a heuristic step that transforms the technology-mapping problem for multiple-output networks into a sequence of sub-problems involving single-output networks. Partitioning is performed during the initial setup phase and as a part of the iterative improvement of a mapped network. We comment here briefly on the former case. The latter is described in Section IV.

The initial partitioning scheme is achieved by grouping vertices into single-output sub-networks, with the property that each outgoing edge of an internal vertex reconverges at or before the output vertex of the sub-network. Partitioning is also used to isolate the combinational portion of a network from the sequential elements and from the I/O's, where *ad hoc* techniques for mapping are used. Therefore, the circuit connections to the sequential elements are removed during the partitioning step.

After the partitioning step, the circuit is represented by a set of combinational circuits that can be modeled by *subject graphs* [8]. These graphs are single-output Boolean networks.

2.2. Decomposition

Decomposition is applied on each subject graph after partitioning. It yields an equivalent subject graph, where each vertex is a *base function*, e.g., a two-input AND/OR/NAND/NOR function. Decomposition provides a mapping solution for libraries that include the base functions (i.e., almost all libraries). We assume in the sequel that the library under consideration includes the base functions. Decomposition also increases the granularity of the network, which is beneficial to the covering step.

2.3. Network Covering

At this point, the logic circuit to be mapped has been partitioned into subject graphs $[\Gamma_1, \cdots, \Gamma_k]$, that have been decomposed. We denote by Γ_f a subject graph whose single-output vertex is v_f. We consider here the covering of a subject graph Γ_f 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_f, having only one vertex with zero out-degree v_j (i.e., a single output). It is characterized by its depth (longest directed path to v_j) and number of inputs. The associated *cluster function* is the Boolean function obtained by collapsing [1] the Boolean expressions associated with the vertices into a single Boolean function. We denote all possible clusters rooted at vertex v_j of Γ_f and their functions by $\{\kappa_{j,1}, \cdots, \kappa_{j,n}\}$.

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

$$f = j + t$$
$$j = xy$$
$$x = e + z$$
$$y = a + c$$
$$z = \bar{c} + d.$$

There are six possible *cluster functions* containing the vertex v_j of the subject graph Γ_f (Fig. 1):

$$\kappa_{j,1} = \mathrm{xy}$$
$$\kappa_{j,2} = \mathrm{x}(a + c)$$
$$\kappa_{j,3} = (e + z)y$$
$$\kappa_{j,4} = (e + z)(a + c)$$
$$\kappa_{j,5} = (e + \bar{c} + d)y$$
$$\kappa_{j,6} = (e + \bar{c} + d)(a + c).$$

The covering algorithm attempts to match each *cluster function* $\kappa_{j,k}$ to a library element. A cover is a set of clusters matched to library elements that cover the subject graph. A cover may optimize the overall area and/or timing. The area cost of a cover is computed by adding the cost of the clusters corresponding to the support variables in the *cluster function* $\kappa_{j,k}$ to the cost of the library

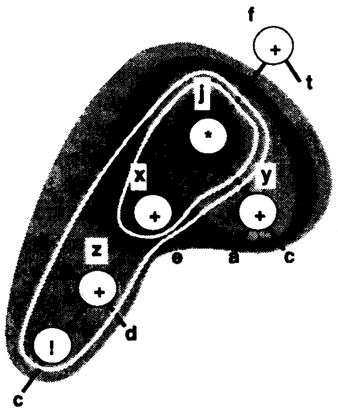

Fig. 1. Graph of all possible covers of j.

```
cover(top,equation,list,depth) {                                    /* Reached search depth: stop recursive expansion */
        if (depth = max_depth) {
        return
    }
    if (equation is empty) {                                        /* First recursion on vertex 'top' */
        if (top not yet mapped) {
            equation = get_equation_from(top)                       /* Copy equation defining top originally */
            list = get_support_from(equation)                       /* Copy original support variables of the equation defining top */
            cover(top,equation,list,1)                              /* Start expanding the equation */
            set_vertex_mapped(top)                                  /* Covering of top is done */
        }
        return }
    while list is not empty {                                       /* Expand all input variables sequentially */
        if (vertex(list) is not mapped) {                           /* Inputs need to be mapped for evaluating the best cost */
            cover(vertex(list),NULL,NULL,depth) }                   /* Map input if it's not mapped already */
        if (vertex(list) is a primary input) skip this one          /* Do not try to expand primary inputs */
        else if (any fanout(vertex(list)) does not reconverge at top) {   /* Do not expand non-reconverging multi-fanout vertices */
            cover(vertex(list),NULL,NULL,depth) }
        else {                                                      /* Expand the current input variable */
            new_list = get_support_from(equation(vertex(list)))     /* Get the support of the equation for the current variable */
            replace current list element by new_list                /* Augment the current list of support variables */
            new_equation = merge(equation,vertex(list))             /* The current variable is eliminated into the current equation */
            cover(top,new_equation,new_list,depth+1)                /* Recurse on the new (expanded) equation */
            put back list in original state }                       /* Get back the input variable list before expansion */
        list = next_element_from(list) }                            /* Get next input variable */
    check_if_in_library(top,equation,list)                          /* Verify if current equation is equivalent to a library element */
    return }
```

Fig. 2. Algorithm for network covering.

element corresponding to the cluster $\kappa_{j,k}$ under consideration. For each vertex v_j in a subject graph Γ_f, there is always at least one *cluster function* $\kappa_{j,k}$ that matches, because the *base functions* (e.g., AND/OR) exist in the library and the network as decomposed accordingly in the initial step phase. When matches exist for multiple clusters, then for any tree-like decomposition the choice of the match of minimal area cost guarantees minimality of the total area cost of the matched sub-graph [8], [7].

The cost of the required inverters is also taken into account at this stage. Each vertex v_j, when mapped, is initially annotated with two library elements: the first one, C_{ON}, gives the best cost for generating the ON set f_j, and the second one, C_{OFF}, gives the best cost for generating the OFF set $\bar{f}_j$. As soon as variable j is used as an input to a gate that is being mapped, then C_{ON} or C_{OFF} is selected according to the required polarity of j. If the same variable j is needed at a later stage with the opposite polarity, then an inverter is automatically taken into account.

The timing cost of a cover can be computed in a similar way, by considering a constant delay model. 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 v_j [12]. When matches exist for multiple clusters, then for any tree-like decomposition the choice of the match of minimal local time guarantees minimality of the total timing cost of the matched sub-graph.

The covering algorithm is implemented by procedure *cover* shown in Fig. 2.

III. BOOLEAN MATCHING

We consider now the *matching* step. Matching is used during the covering stage, to verify if a particular cluster is logically equivalent to an element of the library. Matching can be formulated as checking the tautology between a given Boolean function, the *cluster function* introduced in Section 2.3, and the set of functions representing a library element, for any permutation of its variables. We also consider the polarity problem in connection with the matching problem, because they are closely interrelated in affecting the cost of an implementation. In addition, we exploit the *don't-care* conditions of the *cluster function* in the matching step.

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

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

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

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

for each value of $(y_1, \cdots, y_n)$ and each *care* value of $(x_i^{\phi_i}, \cdots, x_j^{\phi_j}) \notin \mathfrak{DC}$; i.e., equation (1) or (2) holds for all minterms in the *care* set.

If no such ordering and polarity exist, then the element $\mathfrak{G}$ does not match the *cluster function* $\mathfrak{F}$. Furthermore, if no element in the library $\mathfrak{L}$: $\{\mathfrak{G}_1, \cdots, \mathfrak{G}_m\}$ matches $\mathfrak{F}$, then $\mathfrak{F}$ cannot be covered by the library $\mathfrak{L}$. Note that when the library contains the base function, then any vertex v of the Boolean network has always at least one associated *cluster function* that is covered by a library element: the base function into which v is initially decomposed.

Let us define the *NPN-equivalent* set of a function $\mathfrak{F}$ as the set of all the functions obtained by input-variable negation, input-variable permutation, and function negation [15]; we say that a function $\mathfrak{F}$ matches a library element $\mathfrak{G}$ when there exists an NPN-equivalent function that is tautological to $\mathfrak{G}$ modulo the *don't-care* set.

For example, any function $\mathfrak{F}(a,b)$ in the set $\{a + b, \bar{a} + b, a + \bar{b}, \bar{a} + \bar{b}, ab, \bar{a}b, a\bar{b}, \bar{a}\bar{b}\}$ can be covered by the library element $\mathfrak{G}(x_1, x_2) = x_1 + x_2$. Note that in this example, $\mathfrak{G}(x_1, x_2)$ has $n = 2$ inputs, and can match $n! \cdot 2^n = 8$ functions [11].

3.1. Use of Binary Decision Diagrams

The matching algorithms presented here are all based on Boolean operations. The advantage of using Boolean operations is that logic equivalence can be established regardless of the representation. For example, $f_1 = ab + ac + bc$ and $f_2 = a(b\bar{c} + \bar{b}c) + bc$ are logically equivalent, but structurally entirely different. Previous approaches used matching on trees or graphs representing the AND/OR (or equivalent) decomposition of a Boolean factored form (BFF). These algorithms could not detect logic equivalence, since no graph operation can transform the BFF of f_1 into f_2 without taking advantage of Boolean properties. It is important to note that different representations of Boolean functions arise because factoring is not unique, and therefore different forms (e.g., *factored forms, sum of products*) can represent the same function. Therefore a covering algorithm recognizing matches independently from the representations can yield matches of better (or at worst equal) quality than those obtained by structural matching techniques.

Our alogrithms use binary decision diagrams (BDD's) as the basis for Boolean comparisons. BDD's are based on Shannon cofactors. A logic function f is iteratively decomposed by finding the Shannon cofactors of the variables of f [13], [14]. We use BDD's in the form proposed by Bryant, where a fixed ordering of the variables is chosen during Shannon cofactoring [14]. Elsewhere, these have been called *ordered binary decision diagrams*, or OBDD's for short [16]. Bryant also introduced procedures to reduce the size of BDD's. For the purposes of technology mapping, where a BDD representation of a portion of the circuit to map is to be used only once, the computational cost of reducing BDD's is comparable to the cost of doing one single comparison between unreduced BDD's. Therefore we exploit a simple way of comparing unreduced, ordered BDD's.

3.2. A Simple Boolean Matching Algorithm

A Boolean match can be determined by verifying the existence of an assignment of the input variables such that the *cluster function* $\mathfrak{F}$ and the library element $\mathfrak{G}$ are a tautology. Tautology can be checked by using recursive Shannon cofactors [17]. 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 leaves of the recursion that are not in the *don't-care* set. This process is repeated for all possible ordering of the variables of $\mathfrak{F}$, or until a match is found.

The matching algorithm is described by the recursive procedure *simple_boolean_match* shown in Fig. 3, which returns TRUE when the arguments are a tautology for some variable ordering. At level n of the recursion, procedure simple_boolean_match is invoked repeatedly with arguments the cofactors of the nth variable of $\mathfrak{G}$ and the cofactors of all the variables of $\mathfrak{F}$ 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; in the worst case, when all variables have been cofactored. 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 the variables used to cofactor 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 Fig. 3.

Note that in the worst case all permutations and polarities of the input variables are considered. Therefore, up to $n! \cdot 2^n$ different ordered BDD's may be required for

```
simple_boolean_match(f,g,dc,var_list_f,var_list_g,which_var_g) {
    if (dc == 1) return(TRUE)                                   /* If leaf value of DC = 1, local match */
    if ( f and g are constant 0 or 1) return ( f == g )         /* If leaf value of f and g, matches if f == g */
    gvar = pick_a_variable(var_list_g,which_var_g)              /* Choose the next variable from the list of variables of g */
    remaining_var_g = get_remaining(var_list_g,which_var_g)     /* Get list of unexpanded variables of g */
    which_var_f = 1                                             /* Starting pointer for variables of f */
    while ( which_var_f ≤ size_of(var_list_f)) {                /* Try all unexpanded variables of f in turn */
        fvar = pick_a_variable(var_list_f,which_var_f)          /* Get next variable to expand */
        remaining_var_f = get_remaining(var_list_f,which_var_f) /* Update the list of unexpanded variables of f */
        f0 = shannon_decomposition(f,fvar,0)                    /* Find Shannon cofactor of f with (fvar = 0) */
        f1 = shannon_decomposition(f,fvar,1)                    /* Find Shannon cofactor of f with (fvar = 1) */
        g0 = shannon_decomposition(g,gvar,0)                    /* Find Shannon cofactor of g with (gvar = 0) */
        g1 = shannon_decomposition(g,gvar,1)                    /* Find Shannon cofactor of g with (gvar = 1) */
        dc0 = shannon_decomposition(dc,fvar,0)                  /* Find Shannon cofactor of dc with (fvar = 0) */
        dc1 = shannon_decomposition(dc,fvar,1)                  /* Find Shannon cofactor of dc with (fvar = 1) */

        if (simple_boolean_match(f0,g0,dc0,                     /* Verify that the cofactors of f and g */
                remaining_var_f,remaining_var_g,which_var_g+1)  /* are logically equivalent */
                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,                /* If the previous check failed, */
                remaining_var_f,remaining_var_g,which_var_g+1)  /* verify that f is equivalent to the complement of g */
                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) }
```

Fig. 3. Simple algorithm for Boolean matching.

each match. Furthermore, all library elements with n or less inputs need to be considered in turn, since *don't-care* information might reduce the effective number of inputs. The worst-case computational complexity of the algorithm make it practical only for small values of n. Fortunately symmetry information can be used to reduce the search space significantly. Therefore the average computational complexity is much lower than the above bound. Experimental results have shown that Boolean matching is highly efficient, as shown in Section V.

3.3. Matching Completely Specified Functions

In this section we consider the matching problem for completely specified functions; i.e., we neglect the *don't-care* set. This simplification makes possible the use of some properties of Boolean functions that otherwise would not be usable. In particular, there are invariants in completely specified functions that are not in the presence of *don't-cares*. Unateness and symmetry are two such properties. We propose to use these two properties of Boolean functions to speed up the Boolean matching operation, without hampering the accuracy or completeness of the results. In the following sections, we introduce the two properties as key elements to search-space reduction. Matching techniques with *don't-care* conditions will be dealt with in Section 3.4.

3.3.1) Search-Space Reduction

The *simple Boolean matching* algorithm presented in Section 3.2 is computationally expensive for two reasons. First $n!$ permutations of n inputs are needed before two functions can be declared non-equivalent. Second, for each permutation, all 2^n input polarities are required before logic equivalence is asserted. Since all input permutations and polarities must be tried before two logic functions are declared different, then for any arbitrary n-input cluster function, this implies that $n! \cdot 2^n$ comparisons are necessary in the worst case, i.e., whenever a match to a library element fails.

We now look into methods for reducing both the number of permutations and the number of polarities during the process of determining logic equivalence. The number of required polarities is reduced by taking the unateness property into account. The number of input permutations is reduced by using symmetry information. Note that the computational complexity is intrinsic to the Boolean matching problem; therefore, the worst-case number of comparisons is still $n! \cdot 2^n$ for any arbitrary cluster function. However, we will show that the upper bound on complexity is related to the functionality of the library elements, and the most commercially available libraries are constituted of elements that imply much-smaller upper bounds. Therefore, for most cluster functions, the worst-case bound is much less than $n! \cdot 2^n$. In addition, the average cost of Boolean matching is much lower than the worst-case bound and it is shown experimentally to be competitive with other matching techniques.

3.3.2) Unateness Property

To increase the efficiency of the Boolean matching process, we take advantage of the fact that the polarity information of unate variables is not needed to determine the logic equivalence. Therefore we define a transformation Υ that complements the input variables that are negative unate. For example, any function $\mathfrak{F}(y_1, y_2)$ in the set $\{y_1 + y_2, \overline{y_1} + y_2, y_1 + \overline{y_2}, \overline{y_1} + \overline{y_2}, y_1 y_2, \overline{y_1} y_2, y_1 \overline{y_2}, \overline{y_1}\,\overline{y_2}\}$ can be represented by the set $\{y_1 + y_2, y_1 y_2\}$. Note that the polarity information must be kept for binate variables, where both the positive and negative phases are required to express $\mathcal{F}$. By using the transformation Υ, we reduce the information required for the matching and therefore also reduce its computational cost.

As a result of using the unateness property, we redefine the matching problem as follows:

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

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

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

The unateness property is also important for another aspect of search-space reduction. Since unate and binate variables clearly represent different logic operations in Boolean functions, any input permutation must associate each unate (or binate) variable in the cluster function to a unate (or binate) variable in the function of the library element. This obviously affects the number of input-variable permutations when assigning variables of the cluster function to variables of the library element. In particular, it implies that if the cluster function has b binate variables, then only $b! \cdot (n - b)!$ permutations of the input variables are needed. Therefore, the worst-case computational cost of matching a cluster function with b binate variables is $b! \cdot (n - b)! \cdot 2^b$.

3.3.3) Logic Symmetry

One additional factor can be used to reduce the number of required input permutations. Variables or groups of variables that are interchangeable in the cluster function must be interchangeable in the function of the library element. This implies that logic symmetry can be used to simplify the search space.

Variables are symmetric if they can be interchanged without affecting the logic functionality [18]. Techniques based on using symmetry considerations to speed up algebraic matching were also presented by Morrison in [11], in a different context. Reeves also used partial logic symmetry, as a filter during verification [19]. His technique uses BDD's to extract the Chow parameters [20], which can be used to express partial logic symmetry.

Definition: Logic symmetry is represented by the binary relation $\mathcal{SR}_\mathcal{F}$ on the set of inputs $\{x_1, \cdots, x_n\}$ of $\mathcal{F}$, where $\mathcal{SR}_\mathcal{F} = \{\{x_i, x_j\} \mid \mathcal{F}(x_1, \cdots x_i, x_j, \cdots, x_n) \equiv \mathcal{F}(x_1, \cdots, x_j, \cdots, x_i, \cdots, x_n)\}$. In the following, we write $\mathcal{SR}_\mathcal{F}(x_i, x_j)$ to indicate that $\{x_i, x_j\}$ belongs to $\mathcal{SR}_\mathcal{F}$.

The symmetry property of the completely specified functions is an equivalence relation (it is reflexive, symmetric, and transitive) [21], 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. Being an equivalence relation, the symmetry property of variables in logic equations implies a partition of the variables into disjoint subsets.

A *symmetry set* of a function $\mathcal{F}$ is a set of variables of $\mathcal{F}$ that belongs to the binary relation $\mathcal{SR}_\mathcal{F}$. Two variables x_i and x_j of $\mathcal{F}$ belong to the same symmetry set if $\mathcal{SR}_\mathcal{F}(x_i, x_j)$ holds. Let us consider for example function $\mathcal{F} = x_1x_2x_3 + x_4x_5 + x_6x_7$. The input variables of $\mathcal{F}$ can be partitioned into three disjoint sets of symmetric variables: $\{x_1, x_2, x_3\}$, $\{x_4, x_5\}$, and $\{x_6, x_7\}$.

Symmetry sets are further grouped into symmetry classes. A *symmetry class* C_i, $i \in \{1, 2, \cdots\}$, is an ensemble of symmetry sets with the same cardinality i and $S_i = |C_i|$ is the cardinality of a symmetry class C_i. In the previous example, there are two symmetry classes: $C_2 = \{\{x_4, x_5\}, \{x_6, x_7\}\}$ and $C_3 = \{x_1, x_2, x_3\}$, with $S_2 = 2$, $S_3 = 1$. Note that all the other symmetry classes are empty, and therefore $\forall_{i \neq 2,3} S_i = 0$.

The symmetry properties are exploited in technology mapping as follows. Before invoking the mapping algorithm, the symmetry classes of each library element are calculated once. Symmetry classes are used in three different ways to reduce the search space during the matching phase. First, they are used as a filter to quickly find good candidates for matching. A necessary condition for matching a *cluster 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 variable ordering. Once a library element $\mathcal{G}$ that satisfies the previous requirement is found, the symmetry sets of $\mathcal{F}$ are compared to those of $\mathcal{G}$. The only assignments of 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, i.e., symmetry sets belonging to the same symmetry class C_i. Thus the number of permutations required to detect a match is $\Pi_{i=1}^{q} (S_i!)$, where q is the cardinality of the largest symmetry set, and S_i is the cardinality of a symmetry class C_i.

For example, let us enumerate the permutations for matching functions $\mathcal{F} = y_1y_2(y_3 + y_4) + y_5y_6$ and $\mathcal{G} = i_1i_2 + (i_3 + i_4)i_5i_6$. Function $\mathcal{F}$ has one non-empty symmetry class, $C_2(\mathcal{F})$, which contains three symmetry sets, $\{y_1, y_2\}$, $\{y_3, y_4\}$, and $\{y_5, y_6\}$. We associate a name, η_i, with each of the symmetry sets: $C_2(\mathcal{F}) = \{\{y_1, y_2\}, \{y_3, y_4\}, \{y_5, y_6\}\} \equiv \{\eta_1, \eta_2, \eta_3\}$; i.e., we represent the pair of symmetric variables $\{y_1, y_2\}$ by η_1, the pair $\{y_3, y_4\}$ by η_2, etc. Similarly, function $\mathcal{G}$ has only one non-empty symmetry class, C_2, with cardinality $S_2 = 3$. We associate a name, ζ_j, with the symmetry sets of $\mathcal{G}$: $C_2(\mathcal{G}) = \{\{i_1, i_2\}, \{i_3, i_4\}, \{i_5, i_6\}\} \equiv \{\xi_1, \xi_2, \xi_3\}$. We then use the labels η and ξ to represent the different permutations of symmetry sets. The cardinality of the symmetry class C_2 is $S_2 = 3$, and therefore there are $S_2! = 6$ possible assignments of symmetry sets of $\mathfrak{F}$ and $\mathcal{G}$:

$$(\eta_1, \xi_1), (\eta_2, \xi_2), (\eta_3, \xi_3)$$
$$(\eta_1, \xi_1), (\eta_2, \xi_3), (\eta_3, \xi_2)$$
$$(\eta_1, \xi_2), (\eta_2, \xi_1), (\eta_3, \xi_3)$$
$$(\eta_1, \xi_2), (\eta_2, \xi_3), (\eta_3, \xi_1)$$
$$(\eta_1, \xi_3), (\eta_2, \xi_1), (\eta_3, \xi_2)$$
$$(\eta_1, \xi_3), (\eta_2, \xi_2), (\eta_3, \xi_1).$$

Only the last assignment, where the variables of $\mathfrak{F}$ and $\mathcal{G}$ are paired as $\{\{y_1, y_2\}, \{i_5, i_6\}\}$, $\{\{y_3, y_4\}, \{i_3, i_4\}, \{i_3, i_4\}\}$, $\{\{y_5, y_6\}, \{i_1, i_2\}\}$, make functions $\mathfrak{F}$ and $\mathcal{G}$ logically equivalent.

The third use of symmetry classes is during the Boolean comparison itself. Boolean comparisons are based on iterative Shannon cofactoring. Without symmetry considerations, for an n-input function $\mathfrak{F}$, up to 2^n cofactors are needed. But since variables of a symmetry set are freely interchangeable, not all 2^n cofactors are different. For example, given $F = abc$, where $\{a, b, c\}$ are symmetrical, then the cofactor of $\{a = 0, b = 1, c = 0\}$ is equivalent to the cofactor of $\{a = 1, b = 0, c = 0\}$. In general, for a symmetry set containing m variables, only $m + 1$ cofactors are different (corresponding to $0, 1, \cdots, m$ variables set to 1). Assuming the n variables of $\mathfrak{F}$ are grouped into k symmetry sets of size $n_1, \cdots, n_k$ (where $\Sigma_{i=0}^{k} n_i = n$), then the number of needed cofactors is $\Pi_{i=0}^{k} (n_i + 1) \leq 2^n$.

Although in the worst case logic equations have no symmetry at all, our experience with commercial standard cells and (programmable) gate array libraries shows that the library elements are highly symmetric. We computed the symmetry classes C_i of every element of three available libraries (CMOS3, LSI Logic, and Actel), and established the cardinality S_i of each symmetry class C_i extracted. We found that the average cardinality $\bar{S}_i$ of all the symmetry sets of the library cells in the three libraries is less than 2, as shown in Fig. 4. Therefore, the number of permutations $\Pi_{i=1}^{q} (S_i!)$ on the average is close to 1.

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 $\Pi_{i=1}^{q} S_i^u! \cdot S_i^b! = \Pi_{i=1}^{q} S_i^u! \cdot (S_i - S_i^u)! \, \Pi_{i=1}^{q} S_i!$. Hence, when considering the polarity of the binate variables, at most $\Pi_{i=1}^{q} S_i^u! \cdot (S_i - S_i^u)! \cdot 2^{i \cdot (S_i - S_i^u)}$ Boolean comparisons have to be made in order to find a match.

As an example, in the Actel library *act1*, the worst case

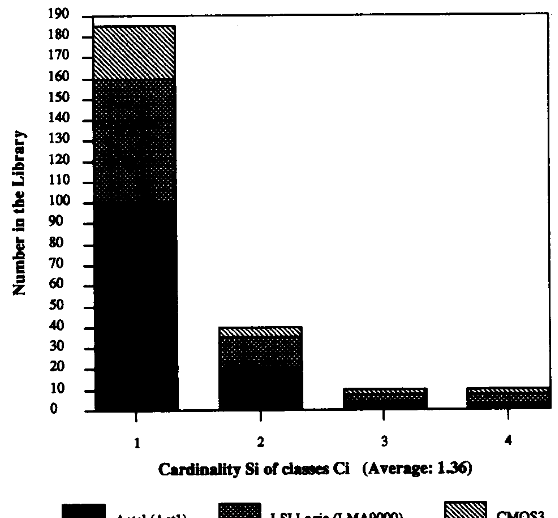

Fig. 4. Distribution of symmetry classes C_i.

occurs for the library element $MXT = d_0c_1c_3 + d_1\bar{c}_1c_3 + d_2c_2\bar{c}_3 + d_3\overline{c_2}\overline{c_3}$, where $S_1 = 7$, and $S_1^u = 4$. In that case, $4! \cdot 3! \cdot 2^3 = 1152 << 7! \cdot 2^7 = 645\,120$, where $7! \cdot 2^7$ represents the number of comparisons needed if no symmetry information is used.

Procedure *boolean_match*, a variation on procedure *simple_boolean_match*, is shown in Fig. 5. It incorporates the symmetry information to reduce the search space: permutations are done only over symmetry sets of the same size. In addition, symmetry sets of unate and binate variables are separated into distinct classes C_i^u and C_i^b. Then only symmetry sets with the same unateness property are permuted.

3.3.4) Determination of Invariant Properties

Unateness and logic symmetry are the two invariant properties we utilize for search-space reduction during Boolean matching. Since cluster functions represent arbitrary portions of Boolean networks, we preprocess every cluster function to detect possible simplification before the unateness of symmetry properties are extracted.

In particular, the preprocessing step recognizes and eliminates vacuous variables. Recall that an equation $\mathfrak{F}$ is vacuous in a variable v_i if the equation can be expressed without the use of v_i [22]. Vacuous variables are detected by checking if $\mathfrak{F}_{v_i} \equiv \mathfrak{F}_{\bar{v}_i}$ for any given variable v_i. When this condition is true, variable v_i is vacuous, and therefore does not influence the value of $\mathfrak{F}$. In that case, we arbitrarily set variable v_i to 0 or to 1, to simplify the expression of function $\mathfrak{F}$.

Unateness is the first property to be extracted from Boolean functions. For efficiency reasons, the unateness determination is done in two successive steps. The first step consists of considering a decomposition of the function $\mathfrak{F}$ into base functions represented by a leaf-DAG and detecting the phase of each variable of $\mathfrak{F}$. The phase de-

```
boolean_match(f,g,f_symmetry_sets,g_symmetry_sets) {
    if ( f and g are constant 0 or 1) {                          /* If leaf value of f and g, matches if f == g */
        return ( f = g ) }
    if ( f_symmetry_sets is empty) {                             /* All variables of current symm set are assigned */
        f_symmetry_sets = get_next_f_symmetry_set()              /* Get next symm set in the list */
        symmetry_size = size_of(f_symmetry_sets)
        while ( symmetry sets of g with                          /* Try all symm sets of g with the same size (recursion) */
            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)
            else return(FALSE) } }
    fvar = pick_a_variable(f_symmetry_sets)                      /* Select variables from compatible symm sets for decomp */
    gvar = pick_a_variable(g_symmetry_sets)

    f0 = shannon_decomposition(f,fvar,0)                         /* Find Shannon cofactor of f with (fvar = 0) */
    f1 = shannon_decomposition(f,fvar,1)                         /* Find Shannon cofactor of f with (fvar = 1) */
    g0 = shannon_decomposition(g,gvar,0)                         /* Find Shannon cofactor of g with (gvar = 0) */
    g1 = shannon_decomposition(g,gvar,1)                         /* Find Shannon cofactor of g with (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) }                                           /* Verify that the cofactors of f and g are equivalent */
    else if ((boolean_match(f0,g1,f_symmetry_sets,g_symmetry_sets)
        and (boolean_match(f1,g0,f_symmetry_sets,g_symmetry_sets) {
        return(TRUE) }                                           /* Verify that f and is equivalent to the complement of g */
    else return(FALSE) {
```

Fig. 5. Algorithm for fast Boolean matching.

tection proceeds as follows. Starting at the root of the leaf-DAG representing function $\mathfrak{F}$, a token representing a positive phase is propagated depth first toward the leaves of the DAG. When a vertex corresponding to a negative unate function is traversed, the phase of the token passed down is complemented. Each variable reached during the graph traversal is annotated with the phase of the current token. The traversal of the leaf-DAG takes at most $2n - 1$ steps, where n is the number of leaves: since the network is decomposed into 2-input gates, then each level in the levelized DAG has at most half the number of vertices of the previous level. Therefore, such a DAG with n inputs has at most $n + n/2 + n/4 + \cdots + 1 = \Sigma_{i=0}^{\infty} n/2^i - \Sigma_{i=0}^{\infty} 1/2^i + 1 = 2n - 2 + 1 = 2n - 1$ vertices.

All variables used in only one phase are necessarily unate. However, the first operation can falsely indicate binate variables, because the algorithm relies on structure, not on Boolean operations. In the second step, the unateness property of the remaining variables (those the first step labeled as binate) is detected verifying implications between cofactors [22]. The unateness property of these *possibly* binate variables is detected by verifying if $\mathfrak{F}_{v_i} \Rightarrow \mathfrak{F}_{\bar{v}_i}$ (negative unate variable) or if $\mathfrak{F}_{\bar{v}_i} \Rightarrow \mathfrak{F}_{v_i}$ (positive unate variable). If neither implication is true, then variable v_i is binate.

Once the unateness information has been determined, symmetry properties are extracted. The transformation Υ, presented in Section 3.3.2, is applied to ensure that symmetry will be detected between unate variables regardless of phase. We detect that two variables are symmetric simply by verifying that $\mathcal{SR}_{\mathfrak{F}}(x_i, x_j)$ is true for that pair of variables. Since the symmetry property is transitive, when variable v_i is symmetric to v_j, and v_j to v_k, the symmetry of v_i and v_k is established without further verification. Similarly, if v_i is symmetric to v_j, and v_j is not symmetric to v_k, then v_i is not symmetric to v_k. As a result, when two variables are symmetric, the symmetry relations to the second variable are identical to those of the first variable, and do not need to be established through additional verification. This implies that it is not always necessary to verify all pairs of variables for symmetry. All pairs of variables must be processed (by verifying that $\mathcal{SR}_{\mathfrak{F}}(x_i, x_j)$ is true) only when there is no symmetry. This is the worst case, and $n(n - 1)/2$ swaps must be done, where n is the number of equal inputs to the equation. When a function is completely symmetric, i.e., when all inputs to a function $\mathfrak{F}$ are symmetric, then only $n - 1$ swaps are needed.

The unateness property is used to reduce the number of swaps needed. Assuming b out of the n input variables are binate, then at worst $b(b - 1)/2 + (n - b)(n - b - 1)/2$ swaps are required. At best, $n - 2$ swaps are needed, when both binate and unate variables are maximally symmetric. In order to verify that swapping two variables $\{v, v_j\}$ leaves $\mathfrak{F}$ unchanged, it is sufficient to verify that $\mathfrak{F}_{v_i\bar{v}_j} \equiv \mathfrak{F}_{\bar{v}_i v_j}$. As in the first step, this is done using Shannon cofactors, and a single ordering (and polarity) of the variables is sufficient.

Since the polarity information is relevant to binate variables, two swaps have to be done for each pair of binate variables, one swap for each polarity of one of the two variables. Again, Shannon cofactors are used to check if the two instances of the equation are the same, and, as in the first step, only variable ordering is used.

From an implementation standpoint, symmetry classes are established once for each library element. Each library element is then inserted into a database, with its symmetry sets used as the key. Library elements with the same symmetry sets are further grouped by functionality (e.g.,

$\mathcal{G}_1 = y_1, y_2$ and $\mathcal{G}_2 = y_1 + y_2$ are grouped together in a new entry $\mathcal{H}$ of the database corresponding to functions of two equivalent, unate inputs).

3.4. Matching Incompletely Specified Functions

The importance of the use of *don't-care* conditions in multiple-level logic synthesis is well recognized [23]. We consider here *don't-care* conditions that are specified at the network boundary and that arise from the network interconnection itself [24]. Since the topology of the network changes during the covering stage, *don't-care* conditions are dynamically computed.

Therefore a technology-mapping algorithm that exploits *don't-care* sets must involve two tasks: 1) computing and updating local *don't-care* sets and 2) using the *don't-care* information to improve the equality of the mapped circuit. We present the use of *don't-care* sets first and we defer their computation to Section 3.4.2.

We have considered two approaches to using *don't-care* conditions in technology mapping. The former uses Boolean simplification before matching a function to a library element. The latter merges simplification and matching in a single step and it is motivated by the following rationale: *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 [25] (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 fewer literals. A pass-transistor-based multiplexer is such a gate. Assuming a function is defined by its ON set $\mathfrak{F}$ and its *don't-care* set $\mathfrak{DC}$:

$$\mathfrak{F} = (a + b)c$$
$$\mathfrak{DC} = \bar{b}\bar{c}$$

then $(a + b)c$ is the representation that requires the least number of literals (3), and the corresponding logic gate is implemented in CMOS pass-transistor logic by 6 transistors. On the other hand, $a\bar{b} + bc$ requires one more literal (4), but it is implemented by only 4 pass transistors, and it is likely to be faster.

A second example, taken from the MCNC benchmark *majority*, is also representative of the uses of *don't-cares* during matching (Fig. 6). In that example, using *don't-cares* yields better matches, and gives an overall lower cost for the resulting circuit: the cluster function OUT $= \bar{T}a + \bar{T}c + \bar{T}d$ has an associated *don't-care* set $\mathfrak{DC} \supset Td + Tc$, and can be reexpressed as OUT $= (\overline{\bar{c}\bar{d}(\bar{a} + T)}$. The two expressions have the same number of literals (4), and are therefore equally likely to be chosen by a technology-independent simplify operation (which relies on literal count). But only one of the two exists in the library, and that match is essential in finding the best overall cost.

These examples show that applying Boolean simplification before matching may lead to inferior results, as compared to merging the two steps into a single task. For this reason, we directly use *don't-care* sets in the Boolean matching step to search for the best implementation in terms of area (or timing).

3.4.1) Compatibility Graph

Boolean matching that incorporates the *don't-care* information can be done using the simple matching algorithm presented in Section 3.2. Unfortunately, when *don't care* conditions are considered, the *cluster function* $\mathfrak{F}$ cannot be uniquely characterized by a symmetry set. Therefore the straightforward techniques based on symmetry sets presented in the previous section no longer apply. The simple matching algorithm would require in the worst case $n! \cdot 2^n$ variable orderings, each ordering requiring up to 2^n Shannon cofactorings. Therefore the algorithm is likely to be inefficient.

Another straightforward approach is considering all the completely specified functions $\mathcal{H}$ that can be derived from $\mathfrak{F}$ and its *don't care* set $\mathfrak{DC}$, by adding to $\mathfrak{F}$ all subsets of $\mathfrak{DC}$. In this case, the symmetry sets can be used to speed up matching. Unfortunately, there are 2^N possible subsets of $\mathfrak{DC}$, where N is the number of minterms in $\mathfrak{DC}$. Therefore this approach can be used only for small *don't-care* sets. For large *don't-care* sets, a pruning mechanism must be used to limit the search space.

We consider in this section a formalism that allows us to efficiently use *don't-care* sets in matching. We first introduce a representation of n-variable functions that exploits the notion of symmetry sets and NPN equivalence and that can be used to determine matches 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 set V is in one-to-one correspondence with the ensemble of all different NPN-equivalent classes of functions. The edge set $E = \{(v_i, v_j)\}$ of the graph $G(V, E)$ denotes the vertex pairs such that adding a minterm to a function included in the NPN class represented by v_i leads to a new function belonging to the NPN class represented by v_j. Such a graph $G(V, E)$ for $n = 3$ is shown in Fig. 7.

Each vertex v_i in the graph is annotated with one function θ_i belonging to the corresponding NPN-equivalent class of v_i. The function θ_i is chosen arbitrarily among the members of the NPN-equivalent class that have the least number of minterms. For example, vertex 4 in Fig. 7 corresponds to functions $\{abc + \bar{a}\bar{b}\bar{c}, \bar{a}bc + a\bar{b}\bar{c}, \bar{a}\bar{b}c + ab\bar{c}, a\bar{b}c + \bar{a}b\bar{c}\}$ and their complements. The representative function θ_4 for vertex 4 is $\{abc + \bar{a}\bar{b}\bar{c}\}$, but could be any of the four functions just enumerated. The set of functions θ_i is used as the basis for establishing relations between vertices v_i. Each vertex v_i is also annotated with the library elements that match the corresponding function θ_i.

The graph $G(V, E)$ is called a *matching compatibility* graph, because it shows which matches are *compatible*

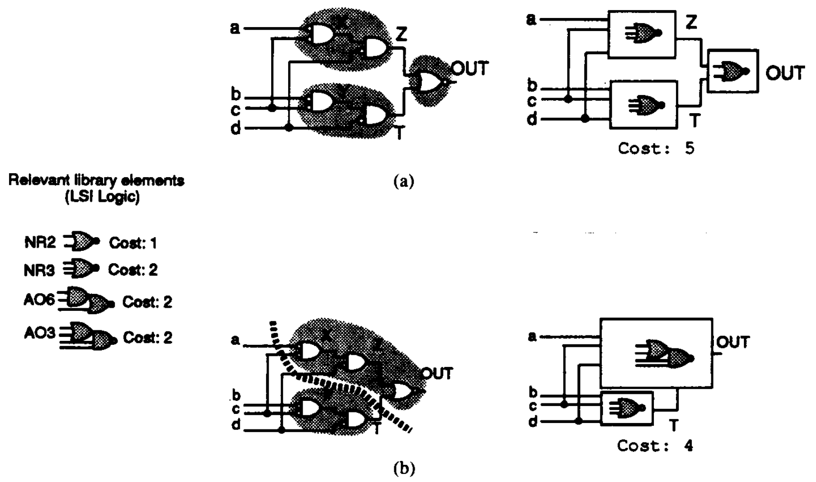

Fig. 6. Mapping the majority MCNC benchmark with LSI Logic library elements. (a) Mapping without DC. (b) Mapping with DC.

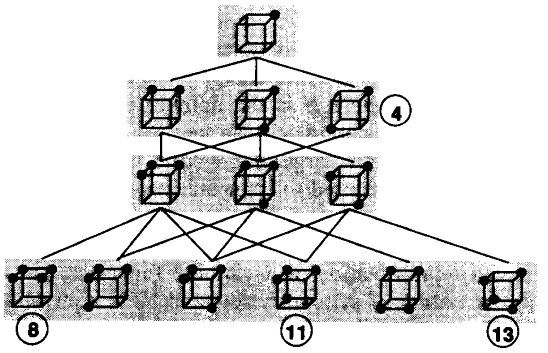

Fig. 7. Matching compatibility graph for 3-variable Boolean space. Note that no path exists between vertex 4 and vertices 8, 11, and 13.

TABLE I
NUMBER OF k-INPUT CELLS INCLUDED IN THE FULL ACT1 AND THE LSI LOGIC LIBRARIES

	Number of Cells in the Library		Percentage of Total	
Number of Inputs	act1	LSI	act1	LSI
1	1	2	0.1%	2.5%
2	8	12	1.1%	14.8%
3	47	22	6.7%	27.2%
4	210	20	30.0%	24.7%
5	285	6	40.7%	7.4%
6	128	8	18.3%	9.9%
7	21	0	3.0%	0.0%
8	1	9	0.1%	11.1%
11	0	2	0.0%	2.5%

with the given function and a *don't-care* set. Note that the size of the compatibility graph is small for functions of 3 and 4 variables, where there are 14 and 222 different NPN-equivalent functions respectively [15], representing the 256 and 65 536 possible functions of 3 and 4 variables. Unfortunately, for functions of more than 4 variables, the number of NPN-equivalent functions grows very quickly (functions of 5 and 6 variables have 616 126 and $\simeq 2 \times 10^{14}$ respectively [20]), although it is very sparse in terms of the vertices corresponding to library elements. At present, we have implemented techniques for technology mapping using *don't-care* conditions for *cluster functions* of at most 4 variables. Although libraries contain cells with more than 4 inputs (see Table I), we found from experimental results of mapped networks that the majority of the library elements used have 4 or less variables (see Table II for the distribution of the number of inputs of cells used for mapping 30 benchmarks). Therefore, it is a reasonable implementation decision to use *don't-cares* only for cluster functions whose fan-in is less than or equal to 4.

TABLE II
NUMBER OF k-INPUT CELLS USED IN MAPPING 30 BENCHMARKS WITH THE FULL ACT1 AND THE LSI LOGIC LIBRARIES. A DEPTH OF 5 WAS USED DURING COVERING

	Number of Cells Used		Percentage of Total	
Number of Inputs	act1	LSI	act1	LSI
1	305	2110	2.9%	12.9%
2	3176	6816	30.7%	41.8%
3	2998	2705	29.0%	16.6%
4	3685	4583	35.6%	28.1%
5	182	103	1.8%	0.6%
6	0	3	0.0%	0.0%

For functions of 4 variables and less, the compatibility graph is constructed once and annotated with the library elements. Each vertex v_i in the graph is also annotated with the paths p_{ij} from the vertex v_i to a vertex v_j corresponding to library element $\mathcal{G}_k \in \mathcal{L}$. The set of paths $P_{i\mathcal{L}} = \{p_{i0}, p_{i1}, \cdots, p_{im}\}$ represents all the paths from vertex v_i to the vertices corresponding to library elements.

Each path represents the set of additional minterms differencing the function θ_i corresponding to v_i from the function θ_j of v_j, where v_j corresponds to a library element. Therefore, checking if a function $\mathcal{F}$ is logically equivalent (modulo the *don't-care* set $\mathcal{DC}$) to a library element $\mathcal{G}_k \in \mathcal{L}$ is the same as verifying that vertex v_i (corresponding to function $\mathcal{F}$) has some path p_{ik} to vertex v_k (corresponding to library element $\mathcal{G}_k$), such that the corresponding minterms are in the *don't-care* set $\mathcal{DC}$.

Let us define $\mathcal{M}(v_i)$ as the number of minterms of the representative function θ_i of vertex v_i in a given $n-$dimensional Boolean space, and the distance between two vertices v_i and v_j as $\mathcal{D}(v_i, v_j) = |\mathcal{M}(v_i) - \mathcal{M}(v_j)|$.

Then the number of paths from vertex v_i to any other vertex (including itself) of the compatibility graph is $2^{2^n - \mathcal{M}(v_i)}$. For a 4-variable compatibility graph, the total number of paths for the entire network is 375 522. This is reasonable from an implementation point of view, since each path is represented by 16 bits, and thus the entire set of paths occupies approximately 750 kilobytes. Note that in general not all paths need to be stored, since the elements of the library usually represent only a subset of all possible NPN-equivalent classes.

If we consider all possible combinations of minterms, the maximum number of paths $|P_{ik}|$ between a vertex v_i and a library element v_k is

$$|P_{ik}| = \binom{2^n - \mathcal{M}(v_i)}{\mathcal{D}(v_i, v_k)} + \binom{2^n - \mathcal{M}(v_i)}{2^n - \mathcal{M}(v_i) - \mathcal{M}(v_k)}.$$

The first term of the expression for $|P_{ik}|$ represents all the combinations of minterms that can make a function of $\mathcal{M}(v_i)$ minterms into a function of $\mathcal{M}(v_k)$ minterms, in an n-dimensional Boolean space. The second term of the expression represents the combinations of minterms that yield a function of $2^n - \mathcal{M}(v_k)$, i.e., the complement of the functions computed for in the first part. Although this upper bound function grows exponentially, experimental results show that the actual number of paths between any pair of vertices is much smaller. For the 4-variable compatibility graph, the maximum number of paths between any two vertices is 384, corresponding to vertices $v_i = abcd$ and $v_j = abcd + \bar{a}(b\bar{d} + \bar{c}d)$. Given that $\mathcal{M}(v_i) = 1$ and $\mathcal{M}(v_j) = 5$, it is clear that the actual number of paths is much smaller than the worst case of 4004 calculated with the above formula. This is due to the fact that not all combinations of added minterms will make function θ_i logically equivalent to θ_k. In some cases, it is even impossible to reach some library element v_k from vertex v_i. For example, in Fig. 7, vertex v_4 cannot reach vertices v_8, v_{11}, v_{13}. In addition, some paths do not need to be recorded, because their head vertex does not correspond to a library cell.

The matching of a *cluster function* $\mathcal{F}$ to a library element is redefined in terms of the compatibility graph as follows. For *cluster functions* with no applicable *don't-care* set, only procedure *boolean_match* is used to find the vertex $v_{\mathcal{F}} \in G(V, E)$ corresponding to the NPN-equivalent class of *cluster function* $\mathcal{F}$ (without using $\mathcal{DC}$). Since the graph represents all possible functions of 4 or less variables, then there exists a vertex in the graph which is NPN-equivalent to $\mathcal{F}$. At the same time vertex $v_{\mathcal{F}}$ is found, the algorithm computes the transformation $\mathcal{J}$ representing the input ordering and polarity on the inputs and output such that $\mathcal{J}(\mathcal{F}) = \theta_{\mathcal{F}}$. The transformation $\mathcal{J}$ is applied to the *don't-care* set $\mathcal{DC}$, to generate a new expression, $\mathcal{J}(\mathcal{DC})$, consistent with the representative function $\theta_{\mathcal{F}}$ of $v_{\mathcal{F}}$. There exists a match to the library 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 are included in the image $\mathcal{J}(\mathcal{DC})$ of the *don't-care* set $\mathcal{DC}$ of $\mathcal{F}$. It is necessary that *don't care* sets are transformed by the operator $\mathcal{J}$ before the path inclusion is checked, because paths in the *compatibility* graph are computed between representative functions θ_i.

The algorithm for graph traversal is shown in Fig. 8. It is invoked with the vertex found by algorithm *boolean_matching* and the image $\mathcal{J}(\mathcal{DC})$ to the corresponding *don't-care* set as parameters. When finished, the algorithm returns the list of all the matching library elements, among which the minimum-cost one is chosen to cover $\mathcal{F}$.

3.4.2) Computation of Relevant Don't-Care Sets

Don't-care sets are classified into two major categories: external DC's, and internal DC's [23]. External DC's are assumed to be provided by the user along the network specification. They represent conditions that never occur on the primary inputs of the circuit and conditions that are never observed on the primary outputs. Internal DC's occur because of the Boolean network structure. They are further classified into *controllability don't-cares* and *observability don't-cares. Controllability don't-cares* represent impossible logic relations between internal variables. *Observability don't-cares* represent conditions under which an internal vertex does not influence any primary output.

The existence of *controllability* and *observability don't-care* sets represent two different (but complementary) aspects of a network. *Controllability don't-care* sets are related to the logic structures in the transitive fan-in of a vertex, whereas *observability don't-care* sets are related to the logic structures in the transitive fan-out of a vertex in the Boolean network. The dynamic programming formulation of technology mapping implies the network to map is modified starting at the primary inputs, and is completed when all primary outputs are processed. The technology-mapping operation modifies the logic structure of the network, and potentially modifies the internal *don't-care* sets. Therefore, *don't-care* sets should be calculated dynamically, as the boundary of the mapped network moves from primary inputs to primary outputs (Fig. 9).

Controllability don't-care sets are conceptually easily computed: a vertex is being mapped only when all its predecessors are mapped. Then all the logic functions expressing a vertex are known, and it is straightforward to extract the *controllability don't-care* sets from them. For

```
dc_match(f,dc) {
    vertex = get_vertex(f)                                  /* Find starting point in the compatibility graph */
    NPN_orientation = cast_to_same_NPN_class(vertex,f)      /* Find how f and the vertex are related */
    NPN_dc = change_NPN(dc,NPN_orientation)                 /* Transform dc as f was transformed into the vertex function */
    for (all paths p_i in vertex) {                         /* Find which paths to library elements are covered by dc */
        if (included(p_i,NPN_dc) {
            update_cover_list(vertex,cover_list) } }
    return(cover_list) }
```

Fig. 8. Algorithm for compatibility graph traversal using don't-cares.

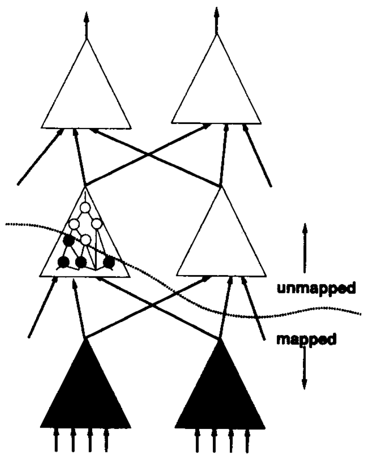

Fig. 9. Example of a partially mapped network.

a subset of variables Y that are inputs to a gate or a subnetwork, the *controllability don't-care* sets (*CDC*) represent the impossible patterns for the variables Y. The *CDC* sets can be derived from the *satisfiability don't-care* sets (*SDC*) by taking the iterated consensus of *SDC* on the variables different from Y, where the *satisfiability don't-care* set[1] is defined as $SDC = \Sigma i \oplus \mathfrak{F}_i$ [23]. For example, for $x = ab$, the *satisfiability don't-care* is $SDC_x = x \oplus (ab) = \bar{x}ab + x(\bar{a} + \bar{b})$. *Controllability don't-care* sets can be computed in a straightforward manner from *SDC* for a particular subset of variables $\{a, b, c, \cdots\}$. Given the satisfiability don't-care set $SDC = \Sigma i \oplus \mathfrak{F}_i$, each variable j of *SDC* not in the cutset $\{a, b, c, \cdots\}$ is eliminated by intersecting $SDC\,|_j$ with $SDC\,|_{\bar{j}}$ i.e., $CDC_{\{a,b,c,\cdots\}}(f) = \Pi_{i \notin \{a,b,c,\cdots\}} \Pi_{i=0}^{1} (\Sigma j \oplus \mathfrak{F}_j)$. For example, let the subset of variables be $\{x, a\}$ and $SDC = \bar{x}ab + x(\bar{a} + \bar{b})$. The $CDC_{\{x,a\}} = SDC\,|_b\ SDC\,|_{\bar{b}} = x\bar{a}$. The *controllability* don't-care sets are computed dynamically as the mapping operation proceeds through the network (Fig. 10). Alternatively, the *CDC* can be computed using images and a range computation algorithm [26], [27], [28], [29].

Observability don't-care sets deal with the successors of vertices. They denote conditions under which a variable does not influence any primary output. For example, in the network $x = at$, $t = b + c$, t is unobservable when $a = 0$ (in that case, $x = 0$ regardless of the value of t). By the very nature of dynamic programming techniques, when a vertex is being processed, its successors are not yet mapped. This implies that the exact *observability* of a vertex is known only after the mapping is completed. Note that unless the *observability don't-cares* are recomputed each time a vertex is modified, it is not possible to use the full *ODC* set for all the vertices [30]. Therefore, *compatible* subsets of the *ODC* need be used, as described in [31]. Although good algorithms have been proposed to compute compatible *observability don't-care* sets [32], [33], efficient implementations are far from trivial, and we decided not to include them at present. The results reported in Section V therefore represent only the use of *controllability don't-care* sets.

Note that when $(\mathfrak{DC} \cup \mathfrak{F} = 1)$, or $\mathfrak{DC} \cup \overline{\mathfrak{F}} = 1$, then the algorithm finds a match with the constant value 1 (0 in the second case). This is always preferred to any other match, since it has a cost of 0. As a result, for every cell mapped to a library element, there exists at least two *controllable* input patterns (i.e., it is possible to generate these patterns from the primary inputs), such that the output of the cell is 0 for one pattern and 1 for the other. This is a necessary condition to make a network testable. Assume that the library cells consist of testable gates (i.e., such that internal nodes are controllable and observable from the cell input/output pins). Then our method guarantees that the mapped circuit is 100% testable for stuck-at faults with the recently proposed I_{DDQ} testing method [34], [35]. However, cell controllability is not sufficient for achieving 100% testability by using standard testing methods. Indeed, it is possible that the output of a cell is not observable at the primary outputs when the *controllable* input patterns are applied to that cell. But by using a post-processing step involving standard ATPG and redundancy removal techniques [36], the mapped network can be made 100% testable for single stuck-at faults. The post-processing step could in principle be eliminated by computing *observability don't-care* conditions. In practice this goal is hard to achieve, since the network is mapped from primary inputs to primary outputs and the *observability* of a vertex being mapped depends on portions of the network yet to be mapped.

IV. Performance-Driven Mapping

We now introduce an ensemble of operations to deal specifically with the optimization of delays through the network, in conjunction with technology mapping. We consider a delay model of the network with

[1]Recall that a Boolean network is defined by a set of equations $i = \mathfrak{F}_i$. Therefore the condition $(i \neq \mathfrak{F}_i) = i \oplus \mathfrak{F}_i$ can never occur.

```
update_cdc(vertex,current_cdc) {
        sdc = recursive_xor(vertex,support(current_cdc))                          /* Add all SDCs between vertex and current CDC cut set */
        unused_support = unused_support_recursive(vertex,support(current_cdc)) /* Get list of cut set vertices now covered by vertex */
        new_cdc = sdc ∪ current_cdc                                              /* Update CDC with SDC previously calculated */
        new_cut_set = vertex ∪ {support(current_cdc) \ unused_support}           /* Update CDC cut set */
        for (v ∈ support(new_cdc), but ∉ new_cut_set)
                new_cdc = consensus(new_cdc, v)                                  /* Eliminate unused variables */
        return(new_cdc)


recursive_xor(eq,vertex,list)
        eq = equation(vertex)
        sdc = xor(eq,vertex)
        for (v ∈ support(eq), but not in list)
                sdc = sdc ∪ recursive_xor(v,list)
        return(sdc)


unused_support_recursive(vertex,list)
        eq = equation(vertex)
        for (v ∈ support(eq) and in list)
                if (last_fanout(v) = vertex)
                        unused_support = unused_support ∪ v
        for (v ∈ support(eq) and not in list)
                unused_support = unused_support ∪ unused_support_recursive(v,list)
```

Fig. 10. Algorithm for dynamic controllability don't-cares calculation.

δ	intrinsic (unloaded) gate delay;
C	load capacitance at a gate output;
$\delta + \alpha \cdot C$	total gate delay;
$a_j =$	$(\delta + \alpha \cdot C + \max a_i)$ arrival time at the output of gate v_j, where a_i is the arrival time at a gate input, with gate $v_i \in$ fan-in(v_j).

In our formulation of the problem, we are given the set of arrival times $\{a_i\}$ of the primary inputs, together with the set of required times $\{r_o\}$ of the primary outputs. For synchronous circuits with period T, we assume the input arrival times to be 0, and the required times at the outputs (i.e., register inputs) to be $T - t_{\text{setup}}$. We use the concept of slack [37], [38], [39], where the slack s_j at a certain vertex v_j corresponds to the difference between the required time at that vertex r_j and the arrival time a_j, i.e., $s_j = r_j - a_j$. Therefore, time-critical nets are those with negative slacks.

We already mentioned in Section 2.3 that dynamic programming techniques can be used to optimize timing as well as area. But there is an important difference between the two optimizations: evaluating the area cost of a particular vertex mapping involves only vertices already mapped (predecessors of the vertex), whereas evaluating the timing cost involves also the successors of the vertex being mapped. Successors are needed because the capacitive load on the output of a gate influences its rise and fall times. Since the dynamic programming technique implies that the successors of a vertex being processed are not yet mapped, then the capacitive load on its output is not known. Therefore specific methods to deal with delay have to be introduced. *Binning* has been proposed by Rudell [40], where each vertex is (possibly) mapped for all the possible capacitive loads on its output. We propose a different heuristic solution, involving iterative mapping of the network. The first mapping of the network includes only the optimization of area. Then, the portions of the network that do not meet the timing constraints are iteratively remapped. This method has the advantage that the entire environment of a vertex is known when it is remapped. In particular, the capacitive load driven by the vertex is known exactly.

It is important to remark that a solution under given timing constraints may not exist. Therefore our strategy is to perform a set of transformations leading to a mapped network that either satisfies the constraints or cannot be further improved by the transformations themselves.

In order to be efficient, iterative remapping has to be powerful enough to modify substantially the portions of the network that do not meet the timing constraints, i.e., the vertices with negative slack. To converge to a good solution in a finite number of steps, it must also be monotonic. We propose an ensemble of three techniques to achieve this goal:

- *Repartitioning* modifies the original partition of multi-fan-out vertices.
- *Redecomposition* changes the two-input decomposition, taking into account delay information.
- *Buffering* adds buffer to large fan-out gates on the critical paths.

The three operations are described next, and a description of their integration in the iterative process follows.

4.1. Repartitioning

Repartitioning takes place after a first mapping has been done, using the traditional partitioning technique outlined in Section 2.1. Repartitioning targets multiple-fan-out vertices that do not meet the timing constraints. The goal is to change partition-block boundaries, by merging subject graphs, to have other (and possibly more) choices when matching (and redecomposing) the vertices along the critical paths. Merging multiple-fan-out subject graphs means the merged portions have to be duplicated for the other fan-outs to achieve the original functionality.

Consider, for example, the subcircuit in Fig. 11, where a gate j is a multiple-fan-out vertex on the critical path. The original arrival time a_j at its output is $a_j = \max(a_i) + \delta_j + \alpha_j \cdot C_j$, where a_i is the arrival time on the inputs, δ_j is the intrinsic delay of gate j, α_j is the fan-out-dependent delay factor, and C_j is the fan-out. Assuming a_l is the latest arriving input, we can reexpress a_j as $a_j = \delta_l + \alpha_l \cdot C_l + \delta_j + \alpha_j \cdot C_j$. Assuming one ot the output fan-outs is on the critical path, duplicating vertex j, with the new gate driving the critical path only, we get (Fig. 12) $a_j' = \delta_l + \alpha_l \cdot (C_l + C_{lj}) + \delta_j + \alpha_j \cdot C_{jk}$, and $a_j'' = \delta_l + \alpha_l \cdot (C_l + C_{lj}) + \delta_j + \alpha_j \cdot (C_j - C_{jk})$, where

a_j'	new arrival time for the critical path,
a_j''	new arrival time for the other fan-outs of j,
C_{lj}	input capacitance of j at input l,
C_{jk}	input capacitance of gate k, which is the gate corresponding to the fan-out of j on the critical path.

Then, the difference in delay is $\Delta a_j' = a_j' - a_j = \alpha_l \cdot C_{lj} + \alpha_j \cdot (C_{jk} - C_j)$, and $\Delta a_j'' = a_j'' - a_j = \alpha_l \cdot C_{lj} - \alpha_j \cdot C_{jk}$. The arrival time of the fan-in vertices of v_j are also modified by the duplication process. The difference in delay is $\Delta a_l' = a_l' - a_l = \alpha_l \cdot C_{lj}$.

This example shows some important properties for vertices with multiple fan-outs:

- Duplicating gates *per se* reduces delay along the critical paths, when $(\alpha_{i0} \cdot C_{ij} - \alpha_j \cdot C_{jk} \leq 0)$. This is usually the case, and it can be verified on a case-by-case basis.
- The fan-ins of the duplicated vertex are slowed down by the addition of one gate load $(\alpha_{ik} \cdot C_{ij})$.

For a particular vertex that does not meet the timing constraints, it is therfore simple to verify how much can be gained by duplication, and whether or not the duplication affects other critical nets. In particular, if all the inputs of the vertex to duplicate have a single fan-out, then duplication is always a good solution. In addition, the duplicated vertex can now be merged forward into the next partition (it is now a single fan-out vertex). Mapping can be redone at this point on the modified partition, possibly improving delay even more.

4.2. Redecomposition

Redecomposition is used both alone or in combination with repartitioning. The goal is to bring late-arriving signals of a partition closer to its output. Redecomposition has (like decomposition) two important side effects:

- It influences the list of library elements that may cover a subject graph.
- It influences the critical path through the Boolean network.

The first point is related to the fact that different decom-

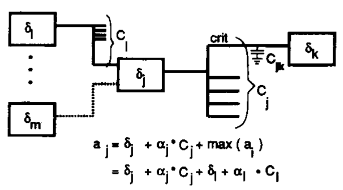
δ_l
δ_m
C_l
δ_j
crit
C_{jk}
δ_k
C_j
$a_j = \delta_j + \alpha_j \cdot C_j + \max(a_i)$
$= \delta_j + \alpha_j \cdot C_j + \delta_l + \alpha_l \cdot C_l$

Fig. 11. Delays in a subcircuit.

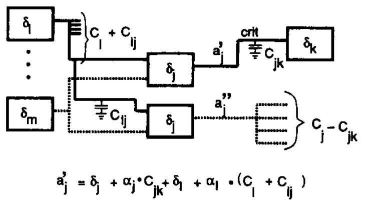

Fig. 12. Delays in a subcircuit after gate duplication.

positions might give rise to different *possible* covers. For example, given $f = a + b\bar{c} + \bar{b}c$, the following decompositions imply very different covers:

$$\begin{array}{ll} f_1 = a + x_1 & f_2 = x_2 + z_2 \\ x_1 = y_1 + z_1 & x_2 = a + y_2 \\ y_1 = b\bar{c} & y_2 = b\bar{c} \\ z_1 = \bar{b}c & z_2 = \bar{b}c. \end{array}$$

In particular, the decomposition f_1 allows the XOR $x_1 = b\bar{c} + \bar{b}c$ to be mapped, whereas in decomposition f_2, the XOR gate cannot be found (because variable a appears as an input to the same gate as $y_2 = b\bar{c}$). We address the first point by heuristically trying to keep repeated literals together during the decomposition. The second point is important because decomposition can be used to push late-arriving signals closer or further from the output, *possibly* reducing or lengthening the critical path. This problem has been addressed by Singh [41] and Paulin [42].

Redecomposition implies unmapping a portion of mapped network, changing its *base function* decomposition, and then remapping the modified block. It is a tentative process, in that mapping the redecomposed partition does not necessarily give better results. We therefore isolate subgraphs being redecomposed, and use the new decomposition only when it produces better results. The evaluation of the value of a redecomposed (and remapped) partition is fairly simple and involves two steps. First, since the subcircuit under consideration has a single output, we can just compare the arrival times of the original and redecomposed partition blocks. Second, we check if the input loads have increased, and, if so, if any other critical net was created.

The redecomposition algorithm we are using follows the same principle that Singh [41] proposed. One significant difference is that we use BDD's instead of kernel extraction for the decomposition. After a subgraph Γ is isolated for redecomposition, its inputs are ordered in de-

```
redecomp(eq) {
    eq_list = get_input_list(eq)                        /* Get support of equation */
    order_input_list(eq_list)                           /* Order support by arrival times (latest first) */
    bdd = create_reduced_bdd(eq,eq_list)                /* Create BDD with previous order */
    get_network_from_bdd(bdd) }                         /* Transform BDD into Boolean Factored form */


get_network_from_bdd(bdd) {
    if (low(bdd) == ZERO) {
        if (high(bdd) == ONE)                           /* f = x */
            return(create_equation(LITERAL,control_var(bdd))
            else {                                      /* f = x h */
            eq = create_product_of(control_var(bdd),
                get_network_from_bdd(high(bdd))
            return(eq) } }
            else if (low(bdd) == ONE) {
    if (high(bdd) == ZERO)                              /* f = x' */
        eq = create_complement(control_var(bdd))
            return(eq)
            else {                                      /* f = x' + h */
            eq = create_sum_of(complement(control_var(bdd)),
                get_network_from_bdd(low(bdd))
            return(eq) } }
            else {
if (high(bdd) == ZERO) {                                /* f = x' l */
        eq = create_product_of(complement(control_var(bdd)),
            get_network_from_bdd(high(bdd))
        return(eq) }
        else if (high(bdd) == ONE) {                    /* f = x + l */
        eq = create_sum_of(control_var(bdd),
            get_network_from_bdd(low(bdd))
        return(eq) } }
        else {                                          /* f = x' l + x h */
    s1 = create_product_of(complement(control_var(bdd)),
        get_network_from_bdd(low(bdd))
    s2 = create_product_of(control_var(bdd),
        get_network_from_bdd(high(bdd)))
    eq = create_sum_of(s1,s2)
    return(eq) } }
```

Fig. 13. Algorithm for BDD-to-Boolean-network conversion.

creasing order of their arrival times. That input list then specifies the order in which the variables are processed during the BDD extraction. After the BDD is reduced, it is then retransformed into a standard Boolean network, which is finally mapped. Procedure *redecompose* transforms a Boolean network into one where the latest arriving inputs are closer to the output (Fig. 13).

For example, let us reconsider the Boolean network described in Section 2.3:

$$f = j + t$$

$$j = xy$$

$$x = e + z$$

$$y = a + c$$

$$z = \bar{c} + d.$$

Assume vertex v_j does not meet its timing requirement, and that the arrival times of the inputs to the partition block rooted by v_j are $\{a_a = 10.0, a_c = 12.0, a_d = 5.0, a_e = 7.0\}$. The variable ordering used for creating the BDD representing j would be $\{c, a, e, d\}$. The resulting BDD is shown in Fig. 14, together with the two-input gate decomposition derived from the BDD.

4.3. *Buffering/Repowering*

Buffering and repowering are used as last resorts, when the two other methods failed. Repowering is used first, using gates with more drive capability for vertices with high fan-out that are on critical paths. After repowering, buffering is used to speed up nets with large fan-outs,

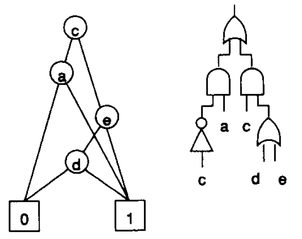

Fig. 14. BDD and corresponding Boolean network.

when neither redecomposition nor repartitioning can be applied, or they would modify other critical nets.

4.4. *Iterative Mapping*

The three techniques outlined above are integrated in an iterative procedure. After a first area-oriented mapping, arrival times and required times are computed for each gate in the network. The required times on the outputs are assumed to be given, and so are the arrival times on the inputs. The difference between arrival time and required time, or slack, is computed for each gate. The gates that have negative slacks are then operated upon in reverse topological order, where primary output gates appear first, and primary input gates appear last.

Redecomposition and repartitioning are used iteratively until the constraints are satisfied (i.e., no negative slack) or no more improvement is possible. Since each step is accepted only if it speeds up the target gate without affecting negatively the slacks on surrounding gates, this process is guaranteed to complete in a finite number of

TABLE III
MAPPING RESULTS FOR AREA (ACTEL ACT1 LIBRARY). "RTIME" INDICATES THE RUN TIME IN SECONDS ON A DECSTATION 5000

					Ceres			
	MIS2.2		Depth 3		Depth 4		Depth 5	
Circuit	cost	rtime	cost	rtime	cost	rtime	cost	rtime
C6288	1649	139.0	1425	85.6	1425	86.5	1425	87.1
k2	1308	4408.0	1209	131.9	1209	170.5	1209	253.3
C7552	1250	501.3	1117	97.4	1100	123.6	1062	178.3
C5315	957	264.8	899	79.2	887	107.2	831	151.2
frg2	941	869.3	1049	103.9	1045	141.6	844	193.1
pair	823	179.9	872	69.2	813	90.3	716	120.7
x1	761	630.1	827	95.4	825	150.6	807	328.5
C3540	658	295.0	641	45.3	628	62.0	608	101.9
vda	651	3103.5	543	46.1	543	61.4	543	91.7
x3	637	288.1	639	54.8	620	74.9	527	108.1
rot	570	254.7	583	48.4	582	73.8	551	137.5
alu4	521	1702.5	561	44.9	559	72.0	550	147.7
C2670	431	177.3	379	28.4	359	39.0	317	64.4
apex6	378	65.2	399	23.2	400	30.5	399	45.6
C1355	371	53.2	176	12.5	178	15.1	178	22.2
term1	360	368.0	380	29.3	365	44.6	302	72.0
x4	346	160.9	433	32.5	428	49.0	368	79.5
alu2	297	623.8	325	24.3	329	41.2	320	90.1
frg1	286	83.9	277	30.1	277	51.6	271	120.6
C1908	283	87.2	266	15.9	265	21.5	263	34.0
ttt2	217	167.9	283	23.4	283	37.5	249	69.0
C880	193	47.0	191	11.0	182	14.9	178	23.0
C499	178	51.8	176	9.9	176	14.1	168	21.1
example2	175	50.8	179	11.1	179	14.5	175	21.5
apex7	147	44.6	149	9.8	150	13.3	142	19.3
my_adder	128	38.6	112	8.8	96	11.8	64	14.4
C432	125	35.9	93	7.5	93	10.4	93	17.2
f5 lm	124	120.0	132	12.2	131	21.2	129	44.7
z4ml	106	96.8	113	10.2	113	17.4	91	33.7
c8	103	45.9	143	11.1	136	15.9	116	22.7
Total	14 974	14 955.0	14 571	1213.4	14 376	1677.9	13 496	2715.1
	100%	1.0	97.3%	0.08	96.0%	0.11	90.1%	0.18

steps. If this process fails in meeting the constraints, the gates with negative slacks are first repowered, and if necessary buffered.

V. IMPLEMENTATION AND RESULTS

The algorithms presented in this paper have been implemented in a program called *Ceres*. *Ceres* consists of approximately 55 000 lines of C code, of which 30 000 lines were encapsulated as the *SLIF Tools*, and reused by other logic optimization programs within the *Olympus* synthesis system [43]. *Ceres* can be used either interactively or in batch mode. It reads a structured, sequential logic description as its input in the SLIF format [39]. The input format allows for hierarchy, combinational logic, sequential elements, and tristate buffers. Annotations can be attached to gates or signals, to specify additional information (capacitance, delay, power requirements, etc.). Various output formats are supported (such as LSI Logic's, Actel's ADL, Mentor's M, Berkeley's EQN), for compatibility with other synthesis tools. A Unix-shell-like interface handles interactive operations, with built-in alias, history, pipes, and help facilities.

We compared *Ceres* to the technology mapper included in UC Berkeley's *mis2.2*. The two programs were run on the MCNC benchmarks, on a DECstation 5000. For area optimization, the technology mapper in *mis2.2* was run with the default options (which allow matches across multi-fanout vertices[2]). For delay optimization, the technology mapper in *mis2.2* was run using the `map -n 1` command, which indicates delay is to be chosen as the principal cost metric. Note that all benchmarks were first technology-decomposed in *mis2.2*, in order to have an identical starting point. The rationale for this style of comparison is to show the effectiveness of Boolean matching techniques compared to other matching techniques. Therefore, there was a need to establish an identical starting point. For all experiments, we used the LSI Logic lsi_10K and Actel act1 libraries as target technologies.

Tables III and IV show mapping results for area. These results reflect the use of *Ceres* and *mis2.2* on unoptimized MCNC benchmarks. Ceres was run using various covering depths, which allow trade-offs between run times and quality of results. Table III represents circuits bound to the Actel library, which has a large number of elements

[2]Note that although using Boolean techniques does not preclude such matches, the current version of Ceres does not recognize them. Therefore, results could be further improved by incorporating that method.

TABLE IV
MAPPING RESULTS FOR AREA (NO DON'T-CARES, LSI LOGIC lsi_10K LIBRARY). "RTIME" INDICATES THE RUN TIME IN SECONDS ON A DECSTATION 5000

	MIS2.2		Ceres Depth 3		Ceres Depth 4		Ceres Depth 5	
Circuit	cost	rtime	cost	rtime	cost	rtime	cost	rtime
C6288	2429	73.7	2241	53.6	2241	52.4	2241	55.9
k2	2182	149.0	2302	110.5	2302	185.3	2302	340.5
C7552	2699	127.1	2797	95.0	2780	139.0	2749	234.2
C5315	1959	86.1	2103	49.0	2002	114.7	1987	192.3
frg2	2045	122.7	1915	90.5	1869	149.1	1712	254.7
pair	1505	68.7	1529	61.4	1525	96.2	1483	165.7
x1	1410	103.1	1498	85.6	1498	175.7	1491	459.4
C3540	1192	67.6	1208	48.1	1201	78.5	1186	155.8
vda	1039	71.6	1090	45.6	1090	71.7	1090	119.5
x3	1285	68.0	1250	52.0	1254	81.2	1095	130.4
rot	1108	70.7	1120	50.7	1121	91.1	1122	107.1
alu4	1009	78.8	1015	48.5	1015	94.6	999	229.5
C2670	862	45.8	909	22.0	887	49.8	893	93.2
apex6	714	33.6	680	24.5	680	35.2	680	59.3
C1355	561	27.9	404	15.5	404	20.4	402	32.9
term1	721	44.9	706	33.6	687	57.3	573	107.2
x4	690	43.6	734	33.2	734	56.9	643	96.8
alu2	565	44.3	588	29.8	587	56.7	576	140.2
frg1	579	45.3	595	37.2	595	77.2	592	210.0
C1908	592	39.4	600	21.6	600	30.9	587	56.5
ttt2	429	33.5	425	23.1	421	39.2	377	73.3
C880	342	22.5	314	16.8	314	25.0	309	44.7
C499	421	24.6	406	14.0	406	18.7	404	29.2
example2	354	23.6	371	14.7	371	20.9	352	33.1
apex7	283	20.0	285	14.0	285	19.0	268	28.7
my_adder	242	17.8	224	13.4	223	18.1	216	24.4
C432	221	17.8	215	12.4	215	17.9	215	32.9
f51m	234	21.1	197	14.5	197	25.0	191	55.0
c8	237	19.1	221	12.6	218	16.7	215	23.7
cht	211	16.6	204	11.5	204	14.5	188	18.8
Total	28 120	1628.5	28 146	1154.9	27 926	1928.9	27 138	3604.9
	100%	1.0	100.0%	0.7	99.2%	1.2	96.3%	2.2

with repeated literals. In this case, results show that using Boolean operations for matching leads to both better implementation and faster run times when comparing to pattern matching based comparisons: results range from 3% better area for more than 12-times-faster run times, to 10% better area for 4-times-faster run times. Table IV shows results using the LSI Logic 10K library. This table shows again how Ceres can trade off quality of the results for run time. In particular, results comparable to those of *mis2.2* are obtained with 30% faster run times, or 4% better results at a cost of a doubling in run time.

Tables V and VI also show mapping results for area, but taking *don't-care* information into account. The results are shown both for circuits that were optimized before mapping and for circuits that were not optimized before mapping. For both the Actel and LSI Logic technologies, the results show that using *don't-cares* during the technology-binding operation improves the quality of the results when operating on both optimized and unoptimized circuits. Circuits in the first category were optimized using UC Berkeley's *mis2.2* with the standard script, which involves technology-independent operations. It is worth noticing that the results of operating on nonoptimized circuits using *don't-care* information sometimes are better than the ones of optimized circuits mapped without using *don't-cares*. This indicates that the use of *don't-care* during the technology-mapping phase effectively does some limited logic synthesis, which is traditionally thought of as a technology-independent operation. Table V, for which the Actel library was used, shows the use of *don't-care* information leads to area improvements of 9% and 8% on unoptimized and optimized circuits respectively. Table VI, for which the LSI Logic library was used, shows area improvements of 8% and 6% in corresponding cases. Calculation of *don't-care* is computationally intensive, requiring large amounts of memory and CPU time. As a result, run times increase by a factor of 10 to 20 when using *don't-care* information. For some examples we use subsets of the full *don't-care* set, to keep reasonable run times. In some cases, even the calculation of subsets of the *don't-cares* involves a very large amount of memory, and the program runs out of space. In tables V and VI, the top portion represents results using subsets of the *don't-care* sets, and the bottom portion represents results using the full *don't-care* sets.

Fig. 15 shows the area/delay trade-off curve for different circuits. Note that all the points on these curves are obtained as successive results during the delay optimiza-

TABLE V
MAPPING RESULTS FOR AREA (USING DON'T-CARES, ACTEL ACT1 LIBRARY). "RTIME" INDICATES THE RUN TIME IN SECONDS ON A DECSTATION 5000

	Original circuits				Optimized circuits			
	No DC		With DC		No DC		With DC	
Circuit	area	rtime	area	rtime	area	rtime	area	rtime
frg2	844	194.7	774	2960.7	369	33.4	361	463.3
x1	807	327.5	761	2979.2	172	35.3	166	171.9
alu4	550	147.5	443	1356.9	457	111.3	410	1151.3
apex6	399	45.9	387	431.8	451	77.0	420	682.2
i6	348	25.0	346	607.7	150	18.0	150	199.8
C1908	263	34.1	248	7687.3	195	26.2	193	7279.5
x4	368	79.4	329	517.4	182	13.9	179	140.1
term1	302	73.2	237	721.0	104	27.2	102	146.7
frg1	271	120.0	249	715.2	106	37.4	106	181.2
x3	527	108.5	549	1388.4	426	64.7	409	587.6
vda	543	92.5	542	1250.6	278	27.0	266	326.7
x1	807	327.5	761	2938.9	172	35.1	166	172.2
C1355	176	12.5	170	4040.4	172	9.4	170	3650.3
C499	168	20.6	170	3321.8	172	19.8	170	3157.1
C432	93	18.1	92	305.7	390	135.0	232	1481.1
f51m	128	47.5	121	146.9	71	21.5	72	110.8
alu2	320	90.1	254	369.2	253	65.0	212	303.7
ttt2	249	70.9	147	262.2	92	14.3	89	74.3
i5	178	18.7	178	67.8	66	4.1	66	9.2
example2	175	22.0	150	92.5	160	17.9	149	105.9
c8	116	10.8	74	92.5	66	5.6	54	57.2
apex7	142	19.4	122	105.6	129	18.9	102	166.9
cht	124	22.2	111	116.5	84	5.0	84	32.9
9symml	94	33.2	94	114.8	101	36.8	103	116.4
z4ml	91	34.6	35	119.5	33	8.5	25	34.5
sct	83	17.2	80	53.3	35	6.1	33	65.2
lal	77	12.7	75	35.3	39	5.9	37	50.8
Total	8243	2026.3	7499	32 799.1	4925	880.3	4526	20 919.1
	1.0	1.0	0.91	16.2	0.60	0.43	0.55	10.3
					1.0	1.0	0.92	23.8

TABLE VI
MAPPING RESULTS FOR AREA (USING DON'T-CARES, LSI LOGIC LSI_10K LIBRARY). "RTIME" INDICATES THE RUN TIME IN SECONDS ON A DECSTATION 5000

	Original circuits				Optimized circuits			
	No DC		With DC		No DC		With DC	
Circuit	area	rtime	area	rtime	area	rtime	area	rtime
frg2	1754	229.4	1525	2928.9	594	40.9	578	408.2
x1	1695	382.2	1560	2939.7	327	43.8	317	169.2
alu4	999	174.3	894	1302.2	836	130.8	785	1060.7
apex6	680	52.2	679	335.8	803	93.0	799	646.3
i6	581	62.3	579	400.0	318	24.8	318	213.7
C1908	596	40.7	588	7701.7	462	32.6	464	7322.0
x4	670	88.7	595	481.9	317	18.7	313	125.7
term1	598	81.7	450	651.3	219	34.0	217	138.6
frg1	583	144.1	506	705.4	227	46.6	226	182.3
x3	1125	115.7	1117	1265.6	750	75.3	720	526.2
vda	1078	113.4	1077	1046.6	520	33.7	511	265.1
x1	1695	382.2	1560	2942.0	327	43.5	317	169.2
C1355	404	24.4	403	4025.9	410	23.4	410	3624.7
C499	406	23.3	425	3068.1	410	23.3	411	3123.2
C432	202	22.1	166	267.8	779	153.1	437	1381.6
f51m	244	52.9	241	172.3	146	28.1	147	100.1
alu2	570	102.6	431	328.1	470	75.7	425	264.2
ttt2	453	73.7	302	188.1	193	18.5	186	63.8
i5	356	23.5	356	67.4	198	7.3	198	12.0
c8	249	27.2	182	77.9	128	14.0	123	59.5
apex7	269	23.4	244	192.2	265	23.2	227	139.7
cht	231	26.6	200	83.7	127	7.9	127	22.8

TABLE VI (*Continued*)

	Original circuits				Optimized circuits			
	No DC		With DC		No DC		With DC	
Circuit	area	rtime	area	rtime	area	rtime	area	rtime
9symml	214	37.6	215	100.3	216	43.5	216	116.5
z4ml	181	42.7	180	127.6	68	12.3	65	39.6
sct	144	20.3	142	47.0	86	9.6	83	61.4
lal	156	16.2	157	36.9	94	9.6	96	48.8
Total	16 133	2383.4	14 774	31 484.4	9290	1067.2	8716	20 285.1
	1.0	1.0	0.92	13.2	0.58	0.45	0.54	8.5
					1.0	1.0	0.94	19.0

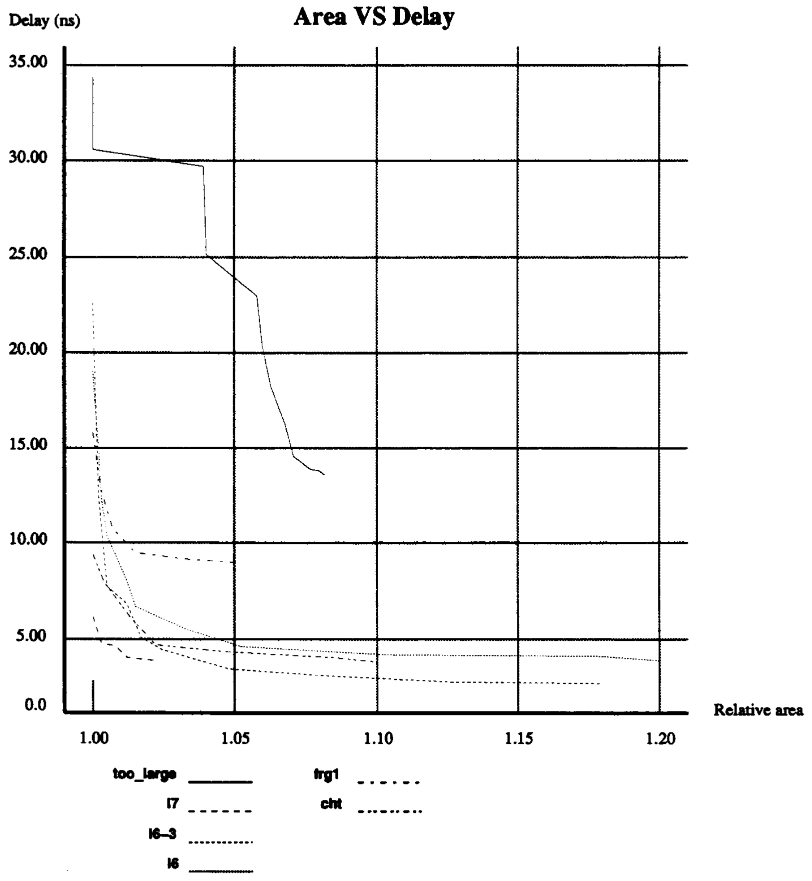

Fig. 15. Example of area/delay tradeoffs.

tion iterations. It is possible for the user to choose between any of these implementation during a single run of the program. In Tables VII and VIII we show the results for both the smallest (area optimization) and the fastest (delay optimization), as an indication of the range of possibilities from which a solution can be selected. Comparative results of *Ceres* and *mis2.2* in Table VII show that for the Actel library, the fastest implementations are 20% faster and 5% smaller when using *Ceres*. Table VIII shows results using the LSI Logic library, where the fastest implementations are 7% faster and 24% smaller when comparing *Ceres* to *mis2.2*.

VI. Conclusion

This paper has presented an ensemble of new methods for technology mapping. Efficient ways of using Boolean techniques were described, among which was the extraction of symmetries as a means of reducing the number of computations during logid-equivalence recognition. Results have shown that for libraries containing many gates with repeated literals, Boolean methods are very efficient in terms of both execution time and quality of the results. For example, *Ceres* finds very good solutions for libraries of field-programmable gate arrays, which fall into that category.

TABLE VII
MAPPING RESULTS FOR DELAY (NO DON'T-CARES, ACTEL ACT1 LIBRARY). "RTIME" INDICATES THE RUN TIME IN SECONDS ON A DECSTATION 5000

	MIS2.2			Ceres				
				Smallest		Fastest		
Circuit	delay	area	rtime	delay	area	delay	area	rtime
C6288	1365.6	2458	422.1	891.9	1425	835.7	2028	608.7
C7552	192.2	1880	945.0	172.7	1062	148.8	1567	433.1
C5315	222.6	1401	499.8	197.8	831	186.3	960	230.5
frg2	164.4	1243	146.5	272.2	844	110.9	918	370.2
pair	169.4	1518	138.8	150.9	716	130.2	1074	271.8
x1	51.6	250	107.0	102.3	807	93.0	851	349.2
C3540	306.4	1022	273.0	290.8	608	265.8	682	155.6
vda	75.4	931	574.3	95.2	543	79.4	669	127.4
x3	82.8	700	282.4	117.2	527	87.0	668	140.0
rot	205.4	504	128.2	318.5	551	192.9	1133	398.1
alu4	246.2	615	179.3	312.0	550	258.9	1183	188.0
C2670	182.4	567	185.6	215.2	317	179.8	342	83.6
apex6	107.0	611	132.5	104.6	399	83.7	583	47.7
C1355	220.4	606	108.6	126.4	176	116.4	253	40.4
term1	83.8	333	135.6	128.1	302	124.8	313	85.8
x4	59.6	373	149.2	101.4	368	65.1	397	104.1
alu2	208.6	355	104.0	287.2	320	199.8	1214	208.5
frg1	71.8	85	38.7	79.4	271	74.4	299	133.4
C1908	207.8	429	131.9	195.6	263	170.6	457	50.8
ttt2	58.6	190	81.7	115.2	249	83.7	306	85.2
C880	175.0	296	79.1	171.1	178	145.1	318	50.9
C499	129.0	334	88.4	128.3	168	114.3	233	27.2
example2	90.6	296	38.8	109.9	175	65.1	251	38.7
apex7	95.4	203	65.4	117.8	142	107.3	177	39.3
my_adder	264.8	159	45.1	157.8	64	148.8	129	29.6
C432	199.6	231	62.5	200.6	93	182.1	168	56.2
f5 lm	68.0	105	52.8	56.5	129	56.5	129	47.9
c8	46.2	132	53.9	49.4	116	46.5	173	66.4
i10	293.8	2142	688.0	518.1	1232	305.2	2002	348.9
dalu	188.2	1490	803.7	213.0	909	169.2	1224	344.7
i7	62.2	637	620.4	55.8	398	32.9	402	97.1
i6	54.4	460	387.1	55.8	320	23.6	328	79.9
count	254.6	127	44.0	166.5	63	83.7	119	12.7
comp	116.4	121	41.1	111.4	60	95.7	250	46.6
i4	41.4	158	60.0	57.8	122	57.8	122	36.1
cht	50.0	160	62.1	32.9	124	32.9	124	22.5
cc	34.6	63	32.0	27.9	31	27.9	31	4.6
Total	6447.2	23 185	7988.6	6505.2	15 453	5181.8	22 077	5461.4
	1.0	1.0	1.0	1.01	0.67	0.80	0.95	0.68

TABLE VIII
MAPPING RESULTS FOR DELAY (NO DON'T-CARES, LSI LOGIC LSI_10K LIBRARY). "RTIME" INDICATES THE RUN TIME IN SECONDS ON A DECSTATION 5000

	MIS2.2			Ceres				
				Smallest		Fastest		
Circuit	delay	area	rtime	delay	area	delay	area	rtime
C6288	84.2	5963	92.0	106.9	2263	106.9	2263	206.9
C7552	43.0	5210	133.0	34.1	2611	25.5	3072	438.8
C5315	22.7	4113	107.1	30.7	1981	28.6	2208	169.5
x1	7.9	764	29.0	11.4	1695	9.0	1709	462.8
C3540	32.9	2423	69.7	42.1	1214	32.3	2951	462.5
vda	14.3	2260	141.6	19.9	1078	10.7	1432	214.7
x3	11.6	2013	56.7	15.8	1125	8.5	1423	211.9
rot	18.6	1345	39.8	37.4	1117	20.1	1866	341.6
alu4	29.7	1434	40.5	45.5	993	27.8	2399	332.8
C2670	23.8	1705	54.7	32.9	864	22.6	1225	90.0
apex6	9.7	1339	38.3	12.7	674	10.8	721	62.8
C1355	19.6	1373	32.0	15.9	404	15.9	404	21.8
term1	11.0	861	32.0	15.0	598	12.7	745	114.4
x4	10.1	1014	34.2	14.0	670	6.7	765	142.2

TABLE VIII (*Continued*)

	MIS2.2			Ceres				
				Smallest		Fastest		
Circuit	delay	area	rtime	delay	area	delay	area	rtime
alu2	27.5	747	26.4	36.8	568	22.1	1261	250.8
frg1	9.3	223	15.9	15.9	583	9.0	613	237.8
C1908	20.8	1234	39.4	29.3	596	23.9	1348	106.4
ttt2	7.4	528	22.4	15.0	452	8.2	572	113.3
C880	17.1	737	26.0	22.3	309	12.7	1444	124.0
C499	13.6	935	26.8	17.1	406	16.4	884	93.6
apex7	9.6	485	20.1	13.5	270	8.1	516	74.3
my_adder	21.0	348	16.3	39.1	256	17.8	682	97.2
C432	20.1	524	19.8	28.3	202	25.6	256	36.3
f51m	7.6	257	15.4	7.3	244	6.5	248	62.9
c8	6.05	321	18.3	7.3	249	6.1	338	45.6
i10	40.6	5529	143.6	81.3	2638	58.4	2731	283.0
dalu	36.4	3855	94.7	31.9	2090	23.8	3015	535.8
i7	20.8	1742	60.8	6.2	762	3.9	779	107.4
i6	17.3	1474	46.2	22.6	580	2.7	684	131.8
count	18.0	243	15.4	28.5	112	7.6	279	43.2
comp	11.8	251	15.5	11.6	151	11.3	219	19.3
i4	7.2	500	24.8	5.7	208	5.0	300	22.9
cht	6.4	507	18.4	9.5	231	3.8	254	55.0
cc	4.6	163	13.3	5.5	74	3.4	87	10.6
Total	662.3	52 420	1580.1	869.0	28 268	614.4	39 693	5723.9
	1.0	1.0	1.0	1.3	0.54	0.93	0.76	3.6

The introduction of *don't-cares* during the technology-binding phase allows for better logic simplification, since the cost metric can be more closely tied to the final realization of the circuit. It was shown that for both optimized and unoptimized circuits, the use of *don't-care* information leads to better results.

Finally, iterative mapping was introduced as a way to improve delays that do not meet user-defined timing constraints. Various techniques are applied in sequence, which monotonically improves the critical paths. It is therefore possible to trade off between area and delay in the final implementation.

The use of *don't-care* sets during technology mapping opens new ways of conceiving logic synthesis, with less separation between technology-independent and technology-dependent transformations. Current results are positive, and indicate the usefulness of migrating traditionally technology-independent synthesis techniques into the technology-dependent domain. However, it is important to remark that the technology-mapping techniques shown here are not a substitute for technology-independent logic synthesis. Moreover, the techniques presented can be further extended by considering observability *don't care* sets and by relying on a sparse compatibility graph to detect logic equivalence of functions of more than 4 inputs in the presence of *don't-cares*.

Acknowledgment

The authors wish to thank Jerry Yang, David Filo, Michiel Ligthart, Polly Siegel, and Hiok Nam Tay for their comments and contributions to the SLIF Tools. The contribution of Thierry Klein, David Ku, and Phil Johnson to the definition of the SLIF format is also acknowledged. The many discussions with Maurizio Damiani about the calculation of *don't cares* and their usage are greatly appreciated.

References

[1] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "Mis: A multiple-level logic optimization system," *IEEE Trans. Computer-Aided Design*, vol. 6, pp. 1062–1081, Nov. 1987.

[2] D. Bostick, G. D. Hachtel, R. Jacoby, M. R. Lightner, P. Moceyunas, C. R. Morrison, and D. Ravenscroft, "The Boulder optimal logic design system," in *Int. Conf. Computer-Aided Design*, IEEE, pp. 62–69, Nov. 1987.

[3] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 1979.

[4] J. Darringer, D. Brand, W. Joyner, and L. Trevillyan, "Lss: A system for production logic synthesis," *IBM J. Res. Develop.*, Sept. 1984.

[5] D. Gregory, K. Bartlett, A. de Geus, and G. Hachtel, "Socrates: a system for automatically synthesizing and optimizing combinational logic," in *23rd Design Automation Conf.*, IEEE/ACM, pp. 79–85, 1986.

[6] M. R. C. M. Berkelaar and J. A. G. Jess, "Technology mapping for standard-cell generators," in *Int. Conf. Computer-Aided Design*, IEEE, pp. 470–473, Nov. 1988.

[7] E. Detjens, G. Gannot, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "Technology mapping in mis," in *Int. Conf. Computer-Aided Design*, IEEE, pp. 116–119, Nov. 1987.

[8] K. Keutzer, "Dagon: Technology binding and local optimization by dag matching," in *24th Design Automation Conf.*, IEEE/ACM, pp. 341–347, 1987.

[9] M. C. Lega, "Mapping properties of multi-level logic synthesis operations," in *Int. Conf. Computer Design*, IEEE, pp. 257–261, Oct. 1988.

[10] R. Lisanke, F. Brglez, and G. Kedem, "Mcmap: A fast technology mapping procedure for multi-level logic synthesis," in *Int. Conf. Computer Design*, IEEE, pp. 252–256, Oct. 1988.

[11] 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, Eds. New York: North-Holland, 1989, pp. 53–64.

[12] R. Rudell, "Logic synthesis for VLSI design," Ph.D. dissertation, U. C. Berkeley (Memorandum UCB/ERL M89/49), Apr. 1989.

[13] S. B. Akers, "Binary decision diagrams," *IEEE Trans. Computers*, vol. 27, pp. 509-516, June 1978.

[14] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Computers*, vol. 35, pp. 677-691, Aug. 1986.

[15] S. Muroga, *Threshold Logic and its Applications*. New York: John Wiley, 1971.

[16] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a bdd package," in *Design Automation Conf.*, IEEE, pp. 40-45, June 1990.

[17] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Norwell, MA: Kluwer Academic Publishers, 1984.

[18] E. J. McCluskey, "Detection of group invariance or total symmetry of a Boolean function," *Bell Syst. Tech J.*, vol. 35, pp. 1445-1453, Nov. 1956.

[19] D. S. Reeves and G. Kedem, "Variable matching for verification," MCNC Tech. Rep. TR90-15, Apr. 1990.

[20] S. L. Hurst, D. M. Miller, and J. C. Muzio, *Spectral Techniques in Digital Logic*. San Diego, CA: Academic Press, 1985.

[21] C. L. Liu, *Elements of Discrete Mathematics*. Neww York: Mc-Graw-Hill, 1977.

[22] E. J. McCluskey, *Logic Design Principles with Emphasis on Testable Semicustom Circuits*. New York: Prentice-Hall, 1986.

[23] 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 Trans. Computer-Aided Design*, vol. 7, pp. 723-740, June 1988.

[24] P. McGeer and R. K. Brayton, "Consistency and observability invariance in multi-level logic synthesis," in *Int. Conf. Computer-Aided Design*, IEEE, pp. 426-429, Nov. 1989.

[25] R. K. Brayton, R. Camposano, G. De Micheli, R. H. J. M. Otten, and J. T. J. an Eijindhoven, "The Yorktown silicon compiler system," in *Silicon Compilation*, D. Gajski, Ed. Reading, MA: Addison-Wesley, 1988.

[26] O. Coudert, C. Berthet, and J. Madre, "Verification of sequential machines based on symbolic execution," in *Lecture Notes in Computer Science No. 407: International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, 12-14 Jun. 1989*, J. Sifakis, Ed. New York: Springer-Verlag, 1990.

[27] H. Savoj, R.K. Brayton, and H. Touati, "The use of image computation techniques in extracting local don't cares and network optimization," in *Int. Conf. Computer-Aided Design*, IEEE, Nov. 1991.

[28] H. Savoj, Private communication, Apr. 1991.

[29] F. Mailhot, "Technology mapping for VLSI circuits exploiting Boolean properties and operations," Ph.D. dissertation, Stanford University, Dec. 1991.

[30] K. C. Chen and S. Muroga, "Sylon-dream: A multi-level network synthesizer," in *Int. Conf. Computer-Aided Design*, IEEE, pp. 552-555, Nov. 1989.

[31] H. Savoj and R. K. Brayton, "The use of observability and external don't cares for the simplification of multi-level networks," in *Design Automation Conf.*, ACM/IEEE, pp. 297-301, June 1990.

[32] M. Damiani and G. D. Micheli, "Derivation of don't care conditions by perturbation analysis of combinational multiple-level logic circuits," in *Int. Workshop Logic Synth.*, MCNC, p. 6.1a, May 1991.

[33] M. Damiani and G. D. Micheli, "Observability don't care sets and Boolean relations," in *Int. Conf. Computer-Aided Design*, IEEE, pp. 502-505, Nov. 1990.

[34] R. R. Fritzemeir, J. Soden, R. K. Treece, and C. Hawkins, "Increased CMOS IC stuck-at fault coverage with reduced I_{DDQ} test sets," in *Int. Test Conf.*, IEEE, pp. 427-435, 1960.

[35] T. J. Chakraborty, S. Bhawmik, and C. J. Lin, "Enhanced controllability for I_{DDQ} test sets using partial scan," in *Design Automation Conf.*, ACM/IEEE, pp. 278-281, June 1991.

[36] D. Bryan, F. Brglez, and R. Lisanke, "Redundancy identification and removal," in *Int. Workshop Logic Synth.*, MCNC, p. 8.3, May 1989.

[37] R. Hitchcock, G. Smith, and D. Cheng, "Timing analysis of computer hardware," *IBM J. Res. Develop.*, vol. 26, pp. 100-105, Jan. 1982.

[38] G. De Micheli, "Performance-oriented synthesis in the Yorktown silicon compiler," *IEEE Trans. Computer-Aided Design*, pp. 751-765, Sept. 1987.

[39] G. De Micheli, "Synchronous logic synthesis," in *Int. Workshop Logic Synth.*, MCNC, p. 5.2, May 1989.

[40] R. Rudell, "Technology mapping for delay," in *Int. Workshop Logic Synth.*, MCNC, p. 8.2, May 1989.

[41] K. J. Singh, A. R. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Timing optimization of combinational logic," in *Int. Conf. Computer-Aided Design*, IEEE, pp. 282-285, Nov. 1988.

[42] P. G. Paulin and F. J. Poirot, "Logic decomposition algorithms for the timing optimization of multi-level logic," in *26th Design Automation Conf.*, IEEE/ACM, pp. 329-333, June 1989.

[43] G. De Micheli, D. Ku, F. Mailhot, and T. Truong, "the Olympus synthesis system," *IEEE Design and Test*, pp. 37-53, Oct. 1990.

Frédéric Mailhot (S'82-M'83) received the B.S. degree in physics engineering from Ecole Polytechnique de Montreal, P.Q., Canada, in 1980, the M.S.E.E. degree from the Universite de Sherbrooke, P.Q., Canada, in 1981, the D.E.A. degree in microelectronics from the Universite de Grenoble, France, in 1982, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 1983.

He is a senior CAD engineer at Synopsys, Mountain View, CA, where he is involved in research and development of logic synthesis tools. Previously, when he was at Stanford, he was involved in the design and development of the Olympus Synthesis System. His research interests include logic synthesis, test, and the interaction between behavioral and logic synthesis.

Giovanni De Micheli (S'79-M'80-SM'89), for a photograph and biography please see page 46 of the January 1993 issue of this TRANSACTIONS.