Technology Mapping of Digital Circuits

Giovanni De Micheli

Center for Integrated Systems
Stanford University
Stanford, CA 94305

Abstract

Technology mapping is an important task of logic synthesis of digital circuits. It consists of transforming a multiple-level Boolean network into an interconnection of primitive gates that belong to a pre-specified library. Therefore technology mapping is of utmost importance for designing in array-based and cell-based methodologies.

Technology mapping aims at achieving minimal area or minimal delay circuits. The problem is computationally hard. Rule-based methods and heuristic algorithms have been applied. The algorithms rely on two important tasks, namely *matching* and *covering*. Matching detects if a portion of a network can be implemented by a library element. Matching can be based on structural or on Boolean operations. Covering consists of choosing an appropriate sets of matched elements, that implement the original network and that optimize the overall area and/or delay.

The major approaches to technology mapping are reviewed, with an emphasis on the recent results achieved by Boolean matching methods that can exploit the *don't care* conditions of a network. Specialized technology mappers are described, that deal with functional-cell and *programmable gate array* libraries.

1 Introduction

Today, most digital circuits are designed by means of computer-aided logic synthesis and optimization techniques. Efficient experimental and commercial synthesis tools have successfully been used for designing commercial products. Semi-custom circuit implementations, such as sea-of-gates, gate-arrays or standard cells, require conforming the digital circuit to the available cell primitives, described by a library. This step, called *technology mapping*, is extremely critical for achieving high-performance and/or minimal area implementations. For this reason, several approaches to technology mapping have been pursued and implemented in research and commercial design tools.

Technology mapping can be seen as a logic synthesis task. Most systems perform technology mapping after having optimized the logic circuit independently of the available circuit primitives. This approach is justified by the overall complexity of the synthesis task.

While most digital circuits are sequential and hierarchical in nature, the most studied technology mapping problems deal with their combinational components, because the choice of implementation of registers, I/O circuits and drivers in a given library is often done by direct replacement. Therefore we consider here the technology mapping problem for combinational circuits. Combinational circuits can be defined by a set of Boolean equations, or equivalently, by an interconnection of uncommitted gates, each one implementing a combinational logic function. The technology mapping problem consists of transforming such a circuit into an I/O equivalent one, where each expression, or gate, is an instance of an element in a given library. Multiple choices in mapping often arise. Since library elements are characterized in terms of area and propagation delay, then some optimal technology mapping problems can be defined, namely: i) minimize the overall area cost (possibly under timing constraints); ii) minimize the maximum I/O propagation delay (possibly under area constraints).

Technology mapping methods depend on the nature of the library elements. While the most common elements are combinational single-output functions, some libraries contain multiple-output cells, such as full-adders and encoders/decoders. Some mappers support only single-output cell libraries. In general, libraries can be described by enumerating its components with their properties. However, emerging technologies, such as *programmable gate-arrays* (PGAs), do not require full enumeration of the library, because this one can be described in a functional way. For example, the library of some memory-based PGAs contains all logic functions of up to a given number of inputs. Therefore, when a functional library representation is possible, specialized technology mapping methods can be used.

The technology mapping problem is a difficult one from a computational complexity stand-point. For this reason, the solution methods that have been proposed fall into two major categories: rule-based technology mappers [5, 10, 11] and heuristic algorithms [2, 3, 6, 13, 14, 15, 16, 17]. In this paper we review the existing approaches in these two classes, with particular emphasis on the novel Boolean techniques developed at Stanford University. We consider then techniques that perform logic optimization concurrently with technology mapping. Eventually, we consider the specialized technology mapping algorithms for functional-cell and *programmable-gate array* libraries.

2 Rule-based technology mapping

Rule-based technology mapping is widely used, because of the flexibility of the method. Some of the early logic synthesis systems, such as LSS [5] and Socrates [10], used a rule-based approach.

In a rule based system, a network is mapped by a step-wise refinement approach. The network undergoes local transformations that preserve its functionality. Each transformation can be seen as the replacement of a subcircuit by an equivalent one that satisfies the technology requirements.

A database contains a family of circuit patterns, and for each one the corresponding replacement patterns according to the target library and the overall goal (such as optimizing area or delay). Several rules may match a pattern, and a priority scheme is used to choose the replacement. The rules may be simple or complex. An example of a simple rule is the replacement of a 4-input OR gate by a tree of 2-inputs OR gates, in the case of a library that does not support 4-input gates. More complex rules can handle, for example, multiple-output combinational and/or sequential logic elements.

The strength of this approach is that rules can be added to the database to cover all thinkable replacement and particular design styles. This is also its weakness. Databases are library specific, and adding or removing elements from a library is not straight-forward. Compiling a library into a database is a time-consuming task and, even though it is automated, it is best done by experts.

Another problem with rule-based systems is the order in which rules should be applied and the possibility of look-ahead and backtracking. Rules must select first the subcircuit to be replaced and then decide on the new pattern. Some rule-based systems, such as LSS [5] and Lores/ex [11], used a greedy strategy. The local "best" replacement according to some metric is chosen at each step.

Other systems, such as Socrates [10], use a more complex search for choosing the transformations. The major concern is to explore the different choices and their consequences before applying a replacement. Technology mapping is an iterative technique that traverses a set of circuit configurations. A search strategy is used to to select the "best" replacement. The breadth of a search is the number of configurations that are taken into account and that can be reached by applying one rule. The depth of a search is the number of consecutive transformations that are considered. For a given value of the breadth and the depth, a set of configurations can be evaluated in the search for the "best" one. The larger the breadth and the depth, the better the look-ahead capability is. This affects substantially the quality of the solution but also the computing time. A set of *meta-rules* is used to decide which portion of the circuit should be replaced and it controls the breadth and depth values. Meta-rules perform also trade-offs between area and delays and between the quality of the solution and the computing time.

3 Algorithms for technology mapping

Algorithms for technology mapping have been developed for libraries of combinational single-output cells. Even though this assumption may seem restrictive, practical approaches to technology mapping may involve several techniques. Multiple-output gates, as well as registers, may be identified, and mapped to the corresponding cells, using simple replacement rules. We assume that the library enumerates the cells, by specifying their equivalent (single-output combinational) logic function and their area and timing properties. Cell timing is generally specified as input/output propagation delay as a function of fanout. Some PGA libraries (or subsets) can be cast in this format (e.g. the Actel libraries) and therefore the algorithms described in this section can also be used for these PGAs. Alternatively, specialized algorithm that exploit the structure of these libraries can be used, as shown in Section 5.

In the sequel, we represent unmapped circuits by Boolean networks, i.e. by directed acyclic graphs where vertices correspond to Boolean functions (represented also by uncommitted gates) and edges to dependencies. Edges are directed from inputs to outputs. Such networks may have been previously optimized by technology independent procedures.

Algorithms for technology mapping were pioneered by Keutzer at AT&T Bell Laboratories [13]. Keutzer realized the similarity between the technology mapping problem and the code generation task in a software compiler. In both cases, a matching problem relates the identification of the possible substitutions, which are chosen on the basis of an optimality criterion.

The technology mapping problem can be divided into a *matching* and a *covering* tasks. Matching consists of identifying whether a subnetwork (i.e. a subset of the original network) can be replaced by a library cell. There are different approaches to solving the matching problem, that relate to the representation being used for the network and the library. Both can be described by Boolean functions, or by graphs representing the algebraic AND/OR decomposition of the expressions. We call the former approach Boolean and the latter structural. Thus expression pattern matching approaches can be classified as structural techniques. The graph-based structural approach was proposed by Keutzer [13], and used later by Rudell [3] and Detjens [6] in program *mis*. An expression pattern matching approach was introduced by Morrison [17]. The Boolean matching technique was was proposed by Mailhot [16] and implemented in program *ceres*.

Two Boolean functions $f_1(x_1, x_2, \ldots, x_n)$ and $f_2(y_1, y_2, \ldots, y_n)$ have a *Boolean match*, if there exists a variable permutation ψ such that $f_1(x_1, x_2, \ldots, x_n) = f_2(\psi(x_1, x_2, \ldots, x_n))$ is a tautology, i.e. if the functions yield equivalent outputs for any input pattern. Given a structural representation of two functions by two graphs in a pre-defined format (e.g. Boolean networks of two-input NOR gates and inverters), there is a *structural match* if the graphs are isomorphic. Clearly a structural match implies a Boolean match, but the converse is not true. Consider for example the following two functions: $f_1 = xy + \overline{x}\,\overline{y} + \overline{y}z$ and $f_2 = xy + \overline{x}\,\overline{y} + xz$, that are logically equivalent, but entirely different in the expression pattern and in its structural representation. Note that different structures for a given function are due to the fact that there exist different possible factoring and that there are even different *sum of products* representations of the same function.

Matching algorithms are described in Sections 3.1.1 and 3.2.1. Efficient algorithms are important, but the major difficulty in solving the technology mapping problem lies in selecting appropriate matches.

The *covering* task consists of choosing an adequate number of cells, that covers the network and that satisfies some optimality properties. The optimum cover problem is intractable [9]. Even though some covering problems have been solved efficiently by branch and bound techniques [18], technology mapping requires solving a particular covering problem, which Rudell has well characterized and named *binate covering* problem [18] and others called *covering with closures*. The problem can be described as follows. Given a network, consider all possible matches and their cost, that can be, for example, the area costs of the corresponding library cells. An optimum covering requires that we select a set of matches covering the network and with minimum total cost. Covering the network implies that there must be a cell matched to the network outputs and recursively a cell matched to any vertex that is an input to a matched cell. Therefore, the selection of a match implies the selection of other matches. If we denote by m_1 a match and by M the set of possible matches at the inputs of the cell corresponding to m_1, then m_1 implies M or equivalently $(\overline{m}_1 + M)$ is a clause of a conjunctive expression representing the feasible covers. Binate covering takes its name from the fact that this expression is binate. There are no efficient algorithms to solve the binate covering problems for networks of interesting size.

To render the problem tractable, most heuristic approaches to technology apply a *partitioning* and a *decomposition* step before *covering*. Partitioning consists of splitting the Boolean network into a collection of subgraphs, each modeling a single-output network, and called *subject graphs*. Then each subject graph is decomposed into an interconnection of *base* functions (e.g. 2-input AND,OR,NAND or NOR and inverters). Note that the two steps are interchangeable in principle, but that performing decomposition after partitioning is preferable in practice, because the latter is applied to circuits of smaller size. These steps are exemplified by Figures 1, 2 and 3.

It is important to remark that the partitioning and decomposition steps are heuristics that help in reducing the problem complexity, but that can hurt the quality of the solution. Partitioning is also used to isolate the combinational portion of a network from the sequential elements and from the I/Os, where *ad hoc* techniques for mapping are used. Partitioning may be achieved by detecting the multiple fanout points that identify partition blocks. Decomposition into two-input functions can then be applied recursively to the vertices of the Boolean network. Decomposition is beneficial in increasing the granularity of the network, and therefore in easing the mapping process.

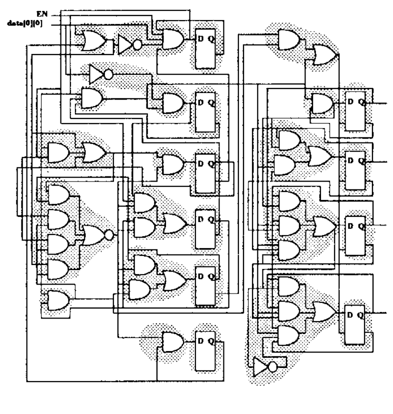


Figure 1: Network partitioning.

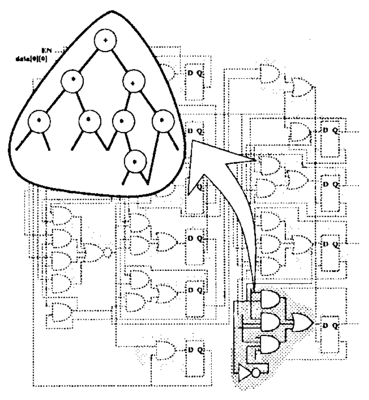


Figure 2: Network decomposition.

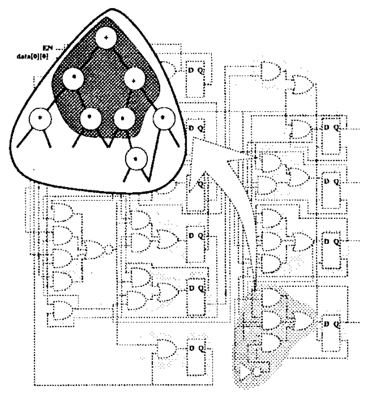


Figure 3: Network covering.

Eventually each subject graph is covered by an interconnection of library cells. For selected portions of the subject graph, all the cells in the library are tried for a match and, when one exists, that portion is labeled with the area and timing cost of the matching cell. The selection of a match is done according to different covering schemes, as described in detail in the following section.

3.1 Covering algorithms based on structural matching

Without loss of generality, we consider here a subject graph that has been decomposed using 2-input NANDs and inverters as the base function. We consider also a representation of the library cells in terms of graphs, called *target graphs*, with a similar decomposition.

Structural matching can be tested by checking the isomorphism between two rooted graphs. Even though the complexity of this problem has not been assessed [9], experimental results have shown that the computation time is negligible for problem of practical size [6]. However, to further simplify the problem and speed-up its solution, Keutzer considered that most cells in any library have corresponding functions where literals are used only once, i.e. whose representations are trees. Notable exceptions are the exclusive (N)OR gates. He proposed then to approximate the graph representation by trees, and to use tree matching algorithms to detect the isomorphism. An example is shown in Figures 4 and 5.

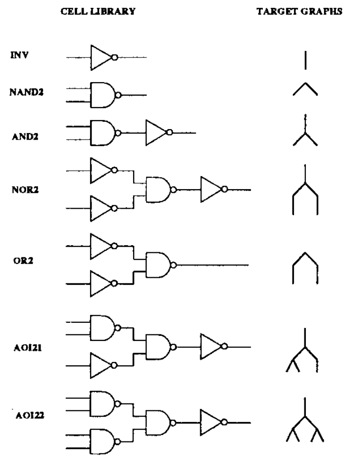


Figure 4: a) Simple library; b) target trees.

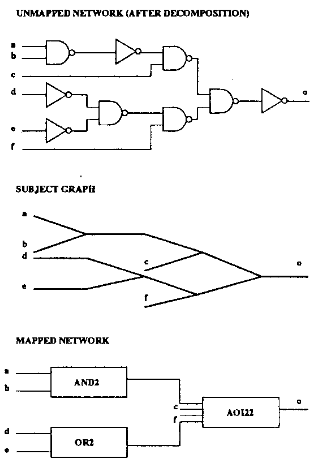


Figure 5: Unmapped network: $o = (abc + (d + e)f)'$; subject graph and mapped network.

3.1.1 Tree-based matching

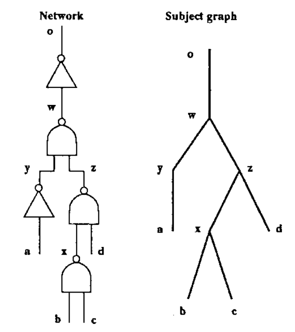
Let us assume then that the subject graph can be represented by a tree, by splitting the terminal vertices, and that the library is also represented by a family of trees. A library cell matches a vertex of the subject graph, if there is an isomorphic subgraph. The problem can be solved in linear time. Several algorithms have been presented for tree matching [12]. Some transform trees into patterns, and use pattern matching techniques. A simple but efficient method is to compare the in-degree of pairs of vertices in both the subject and the target trees, starting from the roots and proceeding top-down until the leaves of the target tree are reached. If there is a mismatch, the algorithm terminates with an unsuccessful match. Else, the adjacent vertices are recursively visited.

3.1.2 Tree-based covering

Keutzer combined the tree matching algorithm with a dynamic programming procedure to perform covering. To be specific, let us consider the case in which an optimal area implementation is sought for. The algorithm traverses the subject graph in a bottom-up fashion. At any internal vertex, it attempts to match the rooted subtree at that vertex with the trees corresponding to all cell libraries. There are three possibilities for any given cell.

- The cell tree and the rooted subtree match. Then, the vertex is labeled with the cell cost.
- The cell tree is isomorphic to a connected subtree of the rooted subtree with the same root and a set of leaves L. Then, the vertex is labeled with the cell cost plus the labels of the vertices L.
- There is no match.

If we assume that the library contains the gate implementing the base function then, for any vertex, there exists at least one cell for which one of the first two cases applies, and we can therefore label that vertex. Therefore, it is possible to choose at each vertex of the subject graph the best labeling among all possible matches. At the end of the graph traversal, the vertex labeling corresponds to an optimum covering. An example of this procedure is given in Figure 6.



Vertex	Match	Cost
x	NAND2(b,c)	NAND2
y	INV(a)	INV
z	NAND2(x,d)	2 NAND2
w	NAND2(y,z)	3 NAND2 + INV
o	INV(w)	3 NAND2 + 2INV
	AOI21(x,d,a)	NAND2 + AOI21

Figure 6: Example of structural covering: a) network; b) subject graph ; c) possible matching at each vertex and corresponding cost.

Note that the overall optimality is weakened by the fact that the total area of a mapped network depends also on the partitioning and decomposition steps.

In the case that a minimum delay covering is sought for, the algorithm can still be applied with the following considerations. When the propagation delay of each cell is insensitive to fanout, then the overall input/output delay of the subject graph can be computed by adding the cell propagation delay to the maximum arrival time at the cell inputs. Therefore, when a cell tree is isomorphic to a connected subtree of the rooted subtree with the same root, then the vertex is labeled with the cell cost (propagation delay) plus the maximum of the labels in the set L. This guarantees the construction of a minimum delay mapping for this delay model.

An accurate delay modeling of most libraries requires a fixed term plus the product of a fanout coefficient and the capacitive load. Libraries have multiple gates for the same logic function, according to the required drive. The higher the fanout drive, the lower the fanout coefficient but the higher the input capacitance, because larger devices are employed. The problem of selecting a cell in a bottom-up traversal of the subject graph is difficult due to the fact that the input capacitance of the following stages are unknown when matching, because the following stages correspond to vertices closer to the root and therefore yet to be mapped.

Rudell realized that for most libraries the values of input capacitances are a finite and small set [18]. Therefore he proposed to use binning techniques to label with integers the possible total capacitive loads at each vertex. This can be done as a pre-processing step. The tree-matching algorithm is still used, but for each vertex an array of solutions is kept, corresponding to the possible loads. For each match, the arrival time is computed for each load value. For each input to the matching cell, the best match for driving the cell (for any load) is selected and the corresponding arrival time is used. If enough labels are used to cover all possible loads, then the algorithm guarantees an optimum solution.

The computational complexity of the tree-covering approach can be evaluated as follows. Let us consider the optimum area covering problem. Attempts for matches are done at every vertex of the subject tree and for each library element. Since the library size is a constant, the complexity is linear in the size of the subject graph. Similar considerations apply to minimum delay covering, where now there is an additional linear dependence on the number of labels used in the discretization of the load values.

While the tree matching algorithm is very efficient, there are three pitfalls. First, there are multiple non-isomorphic representation for some cells, because the decomposition into a given base functions is not necessarily unique. Therefore, a library cell may correspond to more than one target graphs. As a result, each vertex of the subject graph must be tested for matching against a larger number of target graphs, increasing the computational burden of the algorithm.

Second, cells with multiple use of literals, such an exclusive (N)OR gates, cannot be represented by trees. The tree matching and covering algorithm can be extended to subject and target graphs, where the input vertices (i.e. vertices with null in-degree) can have multiple outdegree, i.e. where the corresponding subgraphs obtained by deleting the input vertices are trees. In this case, such target graphs can match vertices of the subject graphs as long as the corresponding input vertices match input vertices. This provides a limited use of cells with multiple use of literals.

Lastly, structural matching can detect a subset of the possible matches and it does not permit the use of the *don't care* information in the mapping process. This can lead to solutions of inferior quality.

3.2 Covering algorithms based on Boolean matching

Boolean matching can overcome the pitfalls of structural matching, but it is in principle a computationally more intensive task, because it requires to check the tautology between a function (representing a portion of the network) and the set of functions representing each library element, for all variable permutations. Mailhot showed [16] that the combination of Boolean matching techniques with an effective reduction of the search space based on symmetry considerations can lead to efficient implementations that are competitive with the structural approach in both computing time and quality of results.

We denote the function representing a portion of the network by $\mathcal{F}(x_1, x_2, \cdots, x_n)$, and we call it *cluster function*. We represent by $\mathcal{G}(x_1, x_2, \cdots, x_n)$ the *target function*, i.e. the function representing a library cell. The covering algorithm tries to match selected cluster functions to all the target functions in the library. In practice, filtering techniques, based on necessary conditions for matching, can be used to screen the library and to improve the computational efficiency. We defer the description of the covering procedure until Section 3.2.2 and we describe first the Boolean matching algorithm.

3.2.1 Boolean matching

Boolean matching questions the existence of a variable order ψ, such that $\mathcal{F}(x_1, x_2, \cdots, x_n) = \mathcal{G}(\psi(x_1, x_2, \cdots, x_n))$ is a tautology. Different methods can be used for tautology checking. In particular, for a given ordering, binary decision diagrams (BDDs) can be used as the basis for Boolean comparisons [4]. The two logic function are represented by two trees, obtained by recursive Shannon decompositions about their variables in the given order.

Equating the two functions is a tautology when the two trees have the same value at the corresponding leaves. An example is shown in Figure 7. If *don't care* conditions are taken into account, then it is sufficient to check that only the values at the *care* leaves match. This test needs to be repeated for all possible orderings ψ or until a match is found.

Boolean matching can be made practical, by considering filters that reduce drastically the number of permutations to be considered. Filters check necessary conditions for matching. For example:

- *Any input permutation must associate each unate (binate) variable in the cluster function to a unate (binate) variable in the function of the target function.*
- *Variables or groups of variables that are interchangeable in the cluster function must be interchangeable in the target function.*

The first point implies that if the cluster function has m binate variables, then only $m! \cdot (n-m)!$ permutations of the input variables are needed. The second point implies that symmetry classes can be used to simplify the search. A symmetry class is a set of variables that are interchangeable without affecting the logic functionality.For a given function $\mathcal{F}(x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n)$, x_i and x_j belong to the same symmetry class if

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

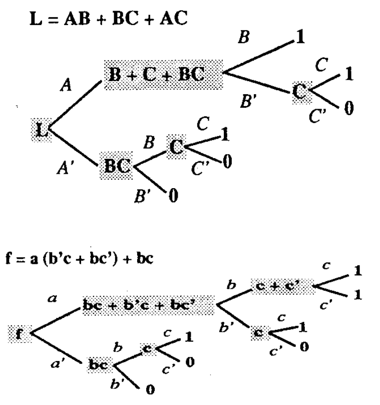


Figure 7: Boolean matching with Binary Decision Diagrams.

The symmetry property of completely specified functions is an equivalence relation (it is reflexive, symmetric and transitive), hence if $\{x_i, x_j\}$ and $\{x_i, x_k\}$ are two symmetry sets, then $\{x_j, x_k\}$ is also a symmetry set. Being an equivalence relation, the symmetry property of variables in logic equations implies a partition of the variables into disjoint subsets.

Symmetry classes are used in three different ways to reduce the search space. First, they are used as a filter to quickly find good candidates for matching. A necessary condition for matching a cluster function $\mathcal{F}$ by a target function $\mathcal{G}$ 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. The symmetry classes for each library element are calculated once before invoking the mapping algorithm.

Second, symmetry classes are used during the ordering of the variables. Once a library element $\mathcal{G}$ that satisfies the previous requirement is found, the symmetry sets of $\mathcal{F}$ are compared to those of $\mathcal{G}$. Then only variables belonging to symmetry sets of the same size can possibly produce a match. Since all variables from a given symmetry set are equivalent, the ordering of the variables within the set is irrelevant. This implies that the permutations need only to be computed over symmetry sets of the same size. Thus the number of permutations required to detect a match is: $\prod_{i=1}^{q}(S_i!)$, where S_i is the number of sets of cardinality i, and q is the size of the largest symmetry set. For example, the gate AOI11 $= \overline{(ab + cd + ef)}$ in the LSI Logic library has 3 sets of coupled unate variables $(\{a, b\}, \{c, d\}, \{e, f\})$, and thus for that library element $S_2 = 3$.

Third, symmetry classes are used to simplify the generation of the BDDS. Indeed, interchangeable variables have interchangeable cofactors, and therefore the number of cofactors to be computed for an n–input function is less than 2^n. In general, for a symmetry set containing m variables, only $m + 1$ cofactors are different (corresponding to $0, 1, \ldots, m$ variables set to 1). Assuming the n variables of $\mathcal{F}$ are grouped into k symmetry sets of size $n_1, \ldots, n_k$ (where $\sum_{i=0}^{k} n_i = n$), then the number of required cofactors is $\prod_{i=0}^{k}(n_i + 1) \leq 2^n$.

Although in the worst case logic equations might have no symmetry at all, our experience with commercial libraries (such as CMOS3, LSI Logic or Actel) is that the library elements are highly symmetrical, the average S_i being less than 2, as shown in Figure 8.

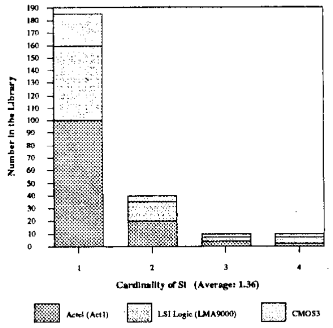


Figure 8: Distribution of symmetry sets S_i.

As a final remark, the unateness information and symmetry classes are used together to further reduce the search space. Unate and binate symmetry sets are distinguished, since both unateness and symmetry properties have to be the same for two variables to be interchangeable. Thus $S_i = S_i^u + S_i^b$, where S_i^u is the number of sets of cardinality i made of unate variables, S_i^b is the number of sets of cardinality i made of binate variables. This further reduces the number of permutations to $\prod_{i=1}^{q} S_i^u! \cdot S_i^b! = \prod_{i=1}^{q} S_i^u! \cdot (S_i - S_i^u)! < \prod_{i=1}^{q} S_i!$.

3.2.2 Boolean covering

We describe here a procedure for Boolean covering of a subject graph. We consider first Boolean covering for area optimization.

We define a *cluster* as a connected sub-graph of the subject graph, having only one vertex with zero out-degree, called the root. It is characterized by its depth (longest directed path to the root) and its number of inputs. The associated *cluster function* is the Boolean function obtained by collapsing [3] the Boolean expressions associated to the vertices into a single Boolean function.

As an example, consider the subject graph shown in Figure 9. The root of the subject graph is v_f, corresponding to variable f. The base functions for the decomposition of the subject graph are the 2-input AND and OR functions.

$$\begin{aligned} f &= j + t \\ j &= xy \\ x &= e + z \\ y &= a + c \\ z &= \overline{c} + d \end{aligned}$$

We consider the clusters, that have v_j as a root and we denote them by $\{\kappa_{j,1}, \ldots, \kappa_{j,N}\}$. The corresponding cluster functions are:

$$\begin{aligned} \kappa_{j,1} &= xy \\ \kappa_{j,2} &= x(a + c) \\ \kappa_{j,3} &= (e + z)y \\ \kappa_{j,4} &= (e + z)(a + c) \\ \kappa_{j,5} &= (e + \overline{c} + d)y \\ \kappa_{j,6} &= (e + \overline{c} + d)(a + c) \end{aligned}$$

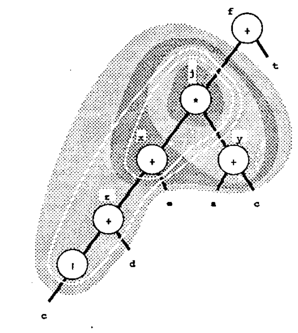


Figure 9: Clusters of the Boolean covering algorithm.

The covering algorithm attempts to match each *cluster function* $\kappa_{j,k}$ to a library element. The area cost of a cover is computed by adding to the cost of the matching of the cluster $\kappa_{j,k}$ under consideration the cost of the clusters corresponding to the support variables in the cluster function of $\kappa_{j,k}$. When the library under consideration includes the base functions, then there is always at least one match for each vertex v_j. When more than one match is found, then the minimal area-cost match is selected.

Let us consider now the problem of minimizing the the delay at vertex v and let us assume fanout independent propagation delays. In this case, the propagation delay through a cluster is added to the maximum of the arrival times at its inputs, to compute the arrival time the vertex v_j. When matchings exist for multiple clusters, then the minimal arrival time match is selected.

The Boolean covering algorithm is based on the dynamic programming argument for structural matching described in Section 3.1.2. However, it is important to remark that its complexity is higher, because when visiting each vertex of the subject graph multiple clusters are matched. A heuristic to bound the number of clusters is to limit the depth of the clusters. In this case, the number of clusters can be considered constant and the algorithm complexity linear in the size of the subject graph. However, limiting the cluster depth weakens the optimality property.

The Boolean covering algorithm still yields optimum solutions for decompositions of the subject graph such that no vertex, other than the inputs, has multiple out-degree, and when the depth of the clusters is unbounded. It is important to stress that the optimality is related, as in the case of structural covering, to the particular decomposition, i.e. the quality of the results could be improved by changing the decomposition of the subject graph. Similarly the overall quality of a mapped circuit depends on the partitioning step as well. Therefore the global optimality of the covering step *per se* has a limited practical value, and near optimal covering solutions are often more than adequate to obtain good mapped networks.

3.3 Covering algorithms and phase assignment

We consider in this section the phase-assignment problem in connection with the matching problem, because they are closely interrelated in affecting the cost of an implementation. Therefore we consider the possibility of a match for a cell that implements the function's complement. Since a complemented signal can be used to feed the following stages, we assume that signals and their complements are available. The goal of considering the phase-assignment problem with the technology mapping is to find the best cover, regardless of the phase of the signals. Since inverters may be required, the cost of the inverters has to be taken into account along with the cost of the cover. Therefore we reconsider the covering algorithms.

3.3.1 Structural covering

The optimal phase assignment can be achieved by using a clever trick. Consider the subject graph and the target graphs after decomposition into the base function. All connections between base gates are replaced by inverter pairs, that leave the overall function unchanged. The dynamic programming covering algorithm can now take advantage of the existence of both phases for every signal in the subject graph. It is important however that the newly introduced inverters are removed, when not contributing to lowering the overall mapping cost. For this reason, a fake element is added to the library. It consists of an inverter-pair, whose actual implementation is a connection, and whose cost is zero. Because of the optimality of the covering algorithm, the computed solution using inverter pairs has lower (or at most equal) cost than a solution computed without the inverter pairs. The only drawback is a slightly increased computational cost, due to the larger size of the subject and target graphs.

3.3.2 Boolean covering

The Boolean match is redefined as follows. Let the cluster function be: $\mathcal{F}(x_1, \ldots, x_n)$. We denote the phase of variable x_i by: $\phi_i \in \{0, 1\}$, where $x_i^{\phi_i} = x_i$ for $\phi_i = 1$, $x_i^{\phi_i} = \overline{x}_i$ for $\phi_i = 0$.

Given a cluster function $\mathcal{F}(x_1, \ldots, x_n)$, and a target function $\mathcal{G}(y_1, \ldots, y_n)$, a match exists if there is an ordering ψ and a phase assignment $\{\phi_1, \ldots, \phi_n\}$, of the input variables of $\mathcal{F}$, such that one of the following equation is a tautology.

$$\begin{aligned} \mathcal{F}(x_1^{\phi_1}, x_2^{\phi_2}, \ldots, x_n^{\phi_n}) &= \mathcal{G}(\psi(x_1, x_2, \ldots, x_n)) \\ \overline{\mathcal{F}}(x_1^{\phi_1}, x_2^{\phi_2}, \ldots, x_n^{\phi_n}) &= \mathcal{G}(\psi(x_1, x_2, \ldots, x_n)) \end{aligned}$$

In other words, if we define the *NPN-equivalent* set of a function $\mathcal{F}$ as the set of all the functions obtained by input variable Negation, input variable Permutation and function Negation we say that a cluster function $\mathcal{F}$ matches a target function $\mathcal{G}$ when there exist a NPN-equivalent function which is tautological to $\mathcal{G}$.

For example, any function $\mathcal{F}(a, b)$ in the set: $\{a + b, \overline{a} + b, a + \overline{b}, \overline{a} + \overline{b}, ay_2, \overline{a}b, a\overline{b}, \overline{a}\overline{b}\}$ can be covered by the library element: $\mathcal{G}(x_1, x_2) = x_1 + x_2$. Note that in this example $\mathcal{G}(x_1, x_2)$ has $n = 2$ inputs, and can match $n! \cdot 2^n = 8$ functions.

The covering algorithm described in Section 3.2.2 can still be used, provided that the matching algorithm is extended. Therefore the Boolean matching algorithm needs to compare two BDDs for all possible variable orders ψ and for all possible phase assignments ϕ, or, at least, until a match is detected. Even though in the worst case $n! \cdot 2^n$ comparisons are required, the state space reduction techniques based on symmetry and unateness considerations described in Section 3.2.1 still apply. For example, only binate variables need to be checked in both phases [16]. Experimental results have shown that Boolean matching with phase assignment is very efficient.

4 Concurrent logic optimization and technology mapping

We consider in this section the possibility of combining logic optimization and technology mapping in a single step. As we mentioned before, most approaches to technology mapping presume a preventive technology independent logic optimization. The reason for separating these two steps lies in the fact that the multiple-level optimization techniques do not impose constraints on the structure of the expressions being manipulated, such as having to match some library element. For example, the extraction of a sub-expression is done regardless whether this sub-expression has a match. It is thought that by constraining the optimization techniques to provide valid matches, the solution space would be reduced and the results would be poor.

We concentrate here on a subset of the logic optimization algorithms that are based on the use of *don't care* conditions. The importance of the use of *don't care* conditions in multiple-level logic synthesis is well recognized [1]. We consider here *don't care* conditions that are specified at the network boundary and that arise from the network interconnection itself [1]. Since the topology of the network changes during the covering stage, *don't care* conditions are dynamically computed.

Consider a partially mapped network, as shown in Figure 10. The interconnection of the cells in the mapped portion of the network induce some relations among the variables. For example, if vertex v_f, corresponding to variable f is mapped to a cell specified by function $\mathcal{G}$, the equation $f = \mathcal{G}$ implies that the relations among the variables given by $f \oplus \mathcal{G}$ can never happen. The union of these relations has been termed *satisfiability don't care* (SDC) set [1]. Consider again the partially mapped network. We are interested in the subset of the SDC set that can affect a cluster function $\mathcal{F}$. In this case, let M be the subset of network variables that are not part of the support of $\mathcal{F}$. By taking the iterated consensus of the SDC set with respect to M we obtain the set of impossible patterns for $\mathcal{F}$, that is termed *controllability don't care* (CDC) set of $\mathcal{F}$. The consensus is computed by eliminating iteratively the variables in M from SDC; the consensus of SDC with respect to $m \in M$ is $SDC_{m=0} \cdot SDC_{m=1}$.

By using the *controllability don't care* set of $\mathcal{F}$ while trying to match $\mathcal{F}$, we combine Boolean simplification with technology mapping. An extension to this operation can be achieved by using the *controllability don't care* set of the unmapped network, i.e. the set of patterns that the mapped network cannot generate. In this case, a matching can exploit the use of a variable that is not in the support of $\mathcal{F}$, to reduce the cost of a cover. This approach is then analogous to the use of Boolean division, which is now performed concurrently to technology mapping.

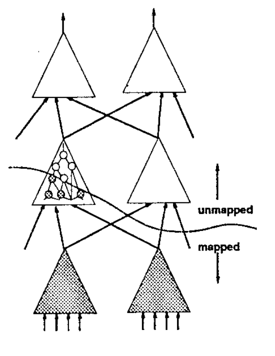


Figure 10: Example of a partially mapped network.

4.1 Use of don't care conditions

The simplest approach to using *don't care* conditions in technology mapping is to simplify the cluster functions before matching. This approach has a potential pitfall. *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 a design style based on cell generators [2], 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. For example, consider a cluster function $\mathcal{F} = (x+y)z$ and a *don't care* set $\mathcal{DC} = \bar{y}\bar{z}$. Then $(x+y)z$ is the representation that requires the least number of literals (3), and the corresponding logic gate is implemented by 6 transistors. On the other hand, $x\bar{y} + yz$ requires one more literal (4), but it is implemented by only 4 pass-transistors, and it is likely to be faster.

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

Boolean matching that incorporates the *don't care* information can be done using the algorithm presented in Section 3.2.1. Unfortunately, when *don't care* conditions are considered, the target function $\mathcal{F}$ cannot be uniquely characterized by a symmetry set. Therefore the techniques based on symmetry sets presented in the previous section no longer apply and the Boolean matching algorithm would require in the worst case $n! \cdot 2^n$ BDD comparisons.

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

4.2 Compatibility graph and Boolean matching

We consider in this section a formalism that allows us to use efficiently *don't care* sets in matching. First we introduce a representation of n-variable functions that exploits the notion of symmetry sets and NPN-equivalence and that can be used to determine matchings while exploiting the notion of *don't care* conditions. For a given number of input variables n, let $G(V, E)$ be a graph whose vertex set V is in one-to-one correspondence with the ensemble of all different NPN equivalent functions, and whose edge set $E = \{(v_i, v_j)\}$ is such that the functions represented by v_i and v_j differ by one minterm. Such a graph $G(V, E)$ for $n = 3$ is shown in figure 11.

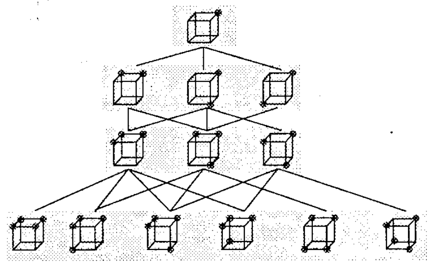


Figure 11: Matching compatibility graph for 3-variable Boolean space.

Consider the vertex $v_\mathcal{F}$ corresponding to a cluster function $\mathcal{F}$. The cluster function matches 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 correspond to minterms in the *don't care* set of $\mathcal{F}$. The graph $G(V, E)$ is called *matching compatibility* graph, because it shows which matchings are *compatible* with the given function.

Mailhot proposed to use the compatibility graph for Boolean matching with *don't care* information as follows. The graph is annotated, by adding the information whether a vertex corresponds to a library cell and, if so, its cost. Then, each vertex is annotated with the paths to the other vertices corresponding to library elements. Since each edge of the compatibility graph corresponds to a minterm, then each path is denoted by a set of minterms. For any a cluster function $\mathcal{F}$, the Boolean matching algorithm of Section 3.2.1 can find efficiently the vertex $v_\mathcal{F}$ in the graph corresponding to it. This vertex may be annotated by a library element or not. The library elements that can be matched to $\mathcal{F}$ correspond to those that can be reached from $v_\mathcal{F}$ by a path whose minterm set is included in the *don't care* set for $\mathcal{F}$. Therefore all the possible matches of $\mathcal{F}$ modulo the *don't care* set can be found and the best one chosen.

To date, this approach has been successfully implemented for functions of 3 and 4 variables, where there are 14 and 222 different vertices in the compatibility graph. All the paths can be computed once for any library and stored. Therefore the matching algorithm is still very efficient. For functions of 5 variables or more, the same approach can be used, even though it is not convenient to store the compatibility graph and the paths because of their size. (There are 616,126 and $\simeq 2 \times 10^{14}$ vertices in the compatibility graphs for 5 and 6 variables respectively.)

5 Technology mapping for functional libraries

We considered so far libraries that are arbitrary collections of combinational gates. One of the difficulties of technology mapping stems from the lack of completeness of the set of functions describing the gates. We comment now on the technology mapping problem for libraries that can be specified without resorting to a full enumeration. A simple example is the case of n-input NAND/NOR gates. A more relevant one is considering the family of gates that can be implemented with a ceiling on the number of devices that can be placed in series and/or in parallel. This example is relevant to the case of functional cell generators, that can synthesize arbitrary gates subject to these bounds [2].

In this case, technology mapping consists of manipulating the equations of a Boolean network, so that they satisfy given constraints. To be more specific, if we represent an expression at a vertex of a Boolean network by a reducible graph, whose series/parallel components are in one to one relation with the conjunction/disjunctions of the expression, then the depth and the breadth of the graph relate to the maximum number of devices in series and/or in parallel. Therefore expression graphs can be labeled as feasible or unfeasible. A technique for constrained mapping was presented by Berkelaar and Jess [2]. Their algorithm processes the expression graphs one at a time. It uses expression substitutions, i.e. the replacement of an expression by a new variable, to decompose the unfeasible graphs.

A similar technology mapping problem arises when considering RAM-based *field programmable gate arrays* (FPGA). Some FPGA architectures can implement any combinational function with a bound on the number of inputs. Again, this problem can be solved by decomposing iteratively the unfeasible expressions of a Boolean network, until all functions satisfy the given bound. Some other FPGA architectures, such as the one marketed by Xilinx Inc., can implement multiple-output combinational functions, with a bound on the number of inputs. To be more specific, let us consider the Xilinx 3000 family, that is an array of programmable modules (cells) that can implement any 5-input single-output or two-output function, where two-output functions cannot have more than 4 common inputs. To use efficiently the two-output cell feature, bounded-input function pairs sharing common inputs have to be detected. A heuristic algorithm was proposed [8], where the information of the overlap of the expression support was used to drive pairwise decomposition algorithms. We refer the interested reader to [8] for further details.

Specialized technology mapping algorithms have also been proposed for fuse/antifuse *electrically programmable gate arrays* (EPGA). These are arrays of uncommitted modules, where the personalization and wiring is achieved by fuse/antifuse technology. For example, the modules of the EPGAs marketed by Actel Inc. are programmed by stuck-at 0/1 of some inputs and/or by bridging some input pairs. Therefore, also in this case, there is no need for an explicit enumeration of the library cells, because their logic function can be derived from that of the uncommitted module in a functional way.

An approach to technology mapping for EPGAs was proposed in [7]. It is also based on the Boolean covering algorithm, described in Section 3.2.2. It differs in the matching step. The entire library is represented only by the uncommitted programmable module, by means of the family of BDDs corresponding to all possible variable orderings. This family is a multiple-rooted two-terminal directed acyclic graph, called global BDD or GBDD. The cluster function is also represented by a BDD. Contrarily to the approach used in Section 3.2.1, the BDDs are now reduced [4], for the sake of efficiency. Let us consider the cells that correspond to a personalization of the uncommitted module by stuck-at 1/0. The search for a match of the cluster function can be cast into checking the isomorphism between the cluster BDD and a rooted subgraph of the GBDD. When a match is found, the labels on the path joining a root of the GBDD to the root of the subgraph determine the personalization pattern. A similar, but more involved, technique is used to determine a match corresponding to a personalization requiring a bridging of input variables. We refer the interested reader to [7] for further details.

6 Summary

Technology mapping is a very important task of logic synthesis for array-based and cell-based digital circuits, where gate primitives are described by cell libraries. Technology mapping provides the link between the functional specification of a circuit and its structural implementation, by taking into account the technological parameters of the libraries. Technology mapping is a complex task, even when considering only multiple-level combinational circuits. Therefore, practical approaches are based on rule-based systems or on heuristic algorithms.

Rule based technology mappers perform step-wise transformations in a network. Rules select a portion of the circuit to be replaced by an appropriate pattern, that is stored in a database. Most commercial implementation of technology mappers use rule-based systems. While this approach can be very versatile in providing rules of different kinds and in supporting various technologies and libraries (including multiple-output and sequential gates), the creation and management of the database (such as adding/deleting library elements) is a delicate task. The completeness of the database can affect significantly the quality of the results.

Algorithms for technology mapping have been the object of intensive investigation. They involve two major tasks: matching and covering. Two major approaches have been pursued. The former uses a graph model of the networks and graph matching algorithms, while the latter uses Boolean operations based on BDD comparisons. The structural approach was used by Keutzer in program *dagon* [13] and by Rudell and others in program *mis* [3]. The Boolean approach was introduced by Mailhot in program *ceres* [16]. Experimental results have shown that the Boolean approach is competitive with the structural approach in both quality of the results and computing time. Both approaches are also competitive with rule-based systems on standard libraries.

The Boolean approach to technology mapping has opened new frontiers. The use of *don't care* information allows to improve the quality of the solution by merging logic optimization and technology mapping into a single step. In addition, the Boolean approach has shown to provide a promising framework for developing specialized mapping algorithms into new technologies, such as electrically programmable gate arrays.

Acknowledgements

The author wish to thank Frédéric Mailhot for many discussions on the subject. This work was supported in part by the National Science Foundation under grant MIP-8719546, by DEC, ATT and NSF under a PYI award, and by ARPA under grant J-FBI-88-101.

References

[1] K. A. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, C. R. Morrison, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. Multilevel logic minimization using implicit don't cares. *IEEE Transactions on CAD/ICAS*, Vol 7, No. 6, pp. 723–740, June 1988.

[2] M. R. C. M. Berkelaar and J. A. G. Jess. Technology mapping for standarcell generators. *International Conference on Computer-Aided Design*, pp. 470–473, November 1988.

[3] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A. R. Wang. Mis: A multiple-level logic optimization system. *IEEE Transactions on CAD/ICAS*, Vol. 6, No. 6, pp. 1062–1081, November 1987.

[4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, Vol. 35, No. 8, pp. 677–691, August 1986.

[5] J. Darringer, D. Brand, W. Joyner, and L. Trevillyan. Lss: A system for production logic synthesis. *IBM Journal of Res. and Dev.*, Vol 28, No 5, pp. 537-545, Sep 1984.

[6] E. Detjens, G. Gannot, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. Technology mapping in mis. *International Conference on Computer-Aided Design*, pp. 116–119, November 1987.

[7] S. Ercolani and G.De Micheli. Technology Mapping for Electrically Programmable Gate Arrays. *Design Automation Conference*, June 1991.

[8] D.Filo, J. Yang, F. Mailhot and G. De Micheli. Technology Mapping for A Multiple-Output RAM-Based Field Programmable Gate Array. *EDAC, Proceedings of the European Design Automation Conference*, Amsterdam, pp. 534-538, February 1991.

[9] M. Garey and D.Johnson. *Computers and Intractability* W.Freeman and Co., San Francisco, 1979.

[10] D. Gregory, K. Bartlett, A. de Geus, and G. Hachtel. Socrates: A system for automatically synthesizing and optimizing combinational logic. *Design Automation Conference*, pp. 79-85, June 1986.

[11] J.Ishikawa, H.Sato, M.Hiramine, K.Ishida, S.Oguri, Y.Kazuma and S.Murai. A Rule-based reorganization system: Lores/ex. *Proceedings International Conference on Computer Design*, pp.262-266, October 1988.

[12] C.Hoffman and M.O'Donnel. Pattern matching in trees. *Journal of ACM* Vol 28, No. 1, pp.68-95, January 1982.

[13] K. Keutzer. Dagon: Technology binding and local optimization by dag matching. *Design Automation Conference*, pp. 341–347, June 1987.

[14] M. C. Lega. Mapping properties of multi-level logic synthesis operations. *International Conference on Computer Design*, pp. 257–261, October 1988.

[15] R. Lisanke, F. Brglez, and G. Kedem. Mcmap: A fast technology mapping procedure for multi-level logic synthesis. *International Conference on Computer Design*, pp. 252–256, October 1988.

[16] F. Mailhot and G. De Micheli, Technology Mapping with Boolean Matching *European Design Automation Conference*, Glasgow, Scotland, pp. 212-216, March 1990.

[17] C. R. Morrison, R. M. Jacoby, and G. D. Hachtel. Techmap: Technology mapping with delay and area optimization. In G. Saucier and P. M. McLellan, editors, *Logic and Architecture Synthesis for Silicon Compilers*, pp. 53–64. North-Holland, 1989.

[18] R. Rudell. *Logic Synthesis for VLSI Design.* Memorandum UCB/ERL M89/49. PhD thesis, U. C. Berkeley, April 1989.