Optimization of Combinational Logic Circuits Based on Compatible Gates *

Maurizio Damiani † Jerry Chih-Yuan Yang Giovanni De Micheli

Center for Integrated Systems
Stanford University, Stanford CA 94305

Abstract - **This paper presents a set of new techniques for the optimization of multiple-level combinational Boolean networks. Such techniques are based on a temporary transformation of the network into an internally unate one. We describe first a technique based upon the selection of appropriate multiple-output subnetworks (consisting of so-called *compatible gates*) whose local functions can be optimized simultaneously. We then generalize the method to larger subsets of unate gates. Because simultaneous optimization of local functions can take place, our methods are more powerful and general than Boolean optimization methods using *don't cares* , where only single-gate optimization can be performed. In addition, our methods represent a more efficient alternative to Boolean relations-based optimization procedures because the problem can be modeled by a *unate* covering problem instead of the more difficult *binate* covering problem. The method is implemented in program *achilles* and compares favorably to SIS.**

1 Introduction

Logic synthesis has been traditionally divided into two-level and multiple-level synthesis. Two-level synthesis has been intensely researched from theoretical and engineering perspectives, and efficient algorithms for exact and approximate solutions are available [1]

Exact optimization algorithms for multiple-level logic networks have also been considered [2]. They are, however, generally impractical even for medium-sized networks. For this reason, many efficient approximation algorithms have been developed over the past decade. Such algorithms can be classified according to the algebraic/Boolean type of operations they perform. Algebraic techniques, such as factoring and kerneling, are described in [3].

As algebraic methods do not take full advantage of the properties of Boolean algebra, a spectrum of Boolean optimization techniques has been developed in parallel. Such techniques consist mainly of iteratively refining an initial network by identifying subnetworks to be optimized, deriving their associated degrees of freedom (expressed by so-called *don't care conditions*), and replacing such subnetworks by simpler, optimized ones.

The independent optimization of the local function of a network, called **single-gate optimization**, lies at one end of the spectrum. It has been shown [4, 5] that the degrees of freedom associated to a single gate can be represented by a *don't care set.* Once this set is obtained, two-level synthesis algorithms can be used to optimize the subnetwork [5].

The concurrent optimization of several local functions, called **multiple-gate optimization**, lies at the other end of the spectrum. Such methods have been shown to offer potentially better quality networks as compared to single-gate optimization because the degrees of freedom of multiple gates are used simultaneously. Heuristic approximations to multiple-gate optimization include the use of *compatible don't cares* [4] which allows us to extend *don't care* based optimization to multiple functions by restricting the *don't care* sets themselves. Although such methods are applicable to large networks, the restriction placed on *don't care* sets reduces the degrees of freedom and hence possibly the quality of the results. Exact methods for multiple-gate optimization, first analyzed in [6], have been shown to best exploit the degrees of freedom. Unfortunately these methods suffer from two major disadvantages. First, even for small subnetworks, the number of primes that have to be derived can be remarkably large; second, given the set of primes, it entails the solution of an often complex *binate covering problem*, for which efficient algorithms are still the subject of investigation. As a result, the overall efficiency of the method is limited, and only relatively small networks can currently be handled.

The binate nature of the covering problem arises essentially from the arbitrariness of the subnetwork selected for optimization. In this paper, we develop alternative techniques for the optimization of multiple-output subnetworks. These techniques are based upon a temporary transformation of a network into an internally unate one, and on an accurate choice of the subnetworks to be optimized. The difficult binate covering step is avoided, and yet an optimization quality superior to *don't care* -based methods with comparable efficiency is achieved because multiple local functions can be optimized simultaneously. To this regard, first we introduce the notion of **compatible set of gates** as a subset of gates whose optimization can be solved *exactly* by classical two-level synthesis algorithms. We show that the simultaneous optimization of compatible gates allows us to reach optimal solutions not achievable by conventional *don't care* methods. We then leverage upon these results and present an algorithm for the optimization of more general subnetworks in an internally unate network. The algorithms have been implemented and tested on several benchmark circuits, and the results in terms of literal savings as well as CPU time are very promising.

2 Terminology

Let $\mathcal{B}$ denote the Boolean set $\{0,1\}$. A k-dimensional Boolean vector $\mathbf{x}=[x_1,\cdots,x_k]^T$ is an element of the set $\mathcal{B}^k$ (bold-facing is hereafter used to denote vector quantities. In particular, the symbol $\mathbf{1}$ denotes a vector whose components are all 1).

A n_i-input, n_o-output Boolean function **F** is a mapping **F**: $\mathcal{B}^{n_i} \rightarrow \mathcal{B}^{n_o}$. A **literal function**, or **literal**, is the function expressed by a variable or its complement. A **cube** c is the product of some literals. A **gate** refers to the local function that the output of the gate represents. A Boolean function can also be represented as a set using its minterms of the ON-set. For the rest of the paper, we interchangeably use set ($\subseteq, \supseteq$) and function ($+, \cdot$) notations to describe Boolean functions for notational convenience.

The **cofactors** (or **residues**) of a function **F** with respect to a variable x_i are the functions $\mathbf{F}_{x_i} = \mathbf{F}(x_1,\ldots,x_i=1,\ldots,x_n)$ and $\mathbf{F}_{x_i'} = \mathbf{F}(x_1,\ldots,x_i=0,\ldots,x_n)$. The **universal quantification** or **consensus** of a function **F** with respect to a variable x_i is the function $\forall_{x_i}\mathbf{F} = \mathbf{F}_{x_i}\mathbf{F}_{x_i'}$. A scalar function F_1 **contains** F_2 (denoted by $F_1 \supseteq F_2$) if $F_2 = 1$ implies $F_1 = 1$. The containment relation holds for two vector functions if it holds component-wise.

A function **F** is termed **positive unate** in x_i if $\mathbf{F}_{x_i} \supseteq \mathbf{F}_{x_i'}$, and **negative unate** if $\mathbf{F}_{x_i} \subseteq \mathbf{F}_{x_i'}$. Otherwise the function is termed **binate** in x_i. A function **F** positive unate in a variable x_i can always be expressed without using the literal x_i' [7].

The desired terminal behavior of a combinational network is **specified** by two functions, **ON(x)** and **DC(x)**, the latter in particular representing the input combinations that either do not occur or such that the value of some of the network outputs is regarded as irrelevant [5].

The functions **ON** and **DC** identify the set of possible terminal behaviors for the network: specifications are met by an implementation, realizing a function **F(x)** if and only if **F(x)=ON(x)** for every input **x** not in **DC**.

Another, equivalent, description of the set of terminal behaviors is in terms of the functions $\mathbf{F}_{min} = \mathbf{ON} \cdot \mathbf{DC}'$ and $\mathbf{F}_{max} = \mathbf{ON} + \mathbf{DC}$. Specifications are met by **F** if

$$\mathbf{F}_{min} \subseteq \mathbf{F} \subseteq \mathbf{F}_{max} \tag{1}$$

*This research is sponsored by NSF and DEC under a PYI award and by ARPA and NSF under contract MIP 8719546.

†Now with the Dipartimento di Elettronica ed Informatica, Universitá di Padova, Via Gradenigo 6/A, Padova, Italy.

We consider hereafter specifications directly in terms of a pair $\mathbf{F}_{min}, \mathbf{F}_{max}$.

2.1 Previous Work

Most Boolean methods for multiple-level logic synthesis rely upon two-level synthesis engines. For this reason and in order to establish some essential terminology, we first review some basic concepts of two-level synthesis.

Two-level Synthesis

Consider the synthesis of a (single-output) network whose output y is to satisfy Eq. (1), imposing a realization of y as a sum of cubes c_k:

$$F_{min} \subseteq y = \sum_{k=1}^{N} c_k \subseteq F_{max}. \tag{2}$$

The upper bound in Eq. (2) holds *if and only if* each cube c_k satisfies the inequality

$$c_k \subseteq F_{max}. \tag{3}$$

Any such cube is termed an **implicant**. An implicant is termed **prime** if no literal can be removed from it without violating the inequality (3). For the purpose of logic design, only prime implicants need be considered [7, 1]. Each implicant c_k has a **cost** w_k associated to it, which depends on the technology under consideration. For example, in PLA minimization all implicants take the same area, and therefore have identical cost; in a multiple-level context, the number of literals can be taken as cost measure [3]. The cost of a sum of implicants is usually taken as the sum of the individual costs.

Once the list of primes has been built, a minimum-cost cover of $\mathbf{F}_{min}$ is determined by solving:

$$\textbf{minimize}: \sum_{k=1}^{N} \alpha_k w_k; \ \textbf{subject to:} \ F_{min} \subseteq \sum_{k=1}^{N} \alpha_k c_k \tag{4}$$

where the Boolean parameters α_k are used in this context to **parameterize** the search space: they are set to 1 if c_k appears in the cover, and to 0 otherwise. The approach is extended easily to the synthesis of multiple-output circuits by defining **multiple-output primes** [7, 1]. A multiple-output prime is a prime of the product of some components of $\mathbf{F}_{max}$. These components are termed the **influence set** of the prime.

Branch-and-bound methods can be used to solve exactly the problem. Engineering solutions have been thoroughly analyzed, for example, in [1], and have made two-level synthesis feasible for very large circuits.

Eq. (4) can be rewritten as

$$\forall_{x_1,\ldots,x_n} \left(\sum_{k=1}^{N} \alpha_k c_k(\mathbf{x}) + F'_{min}(\mathbf{x}) \right) = 1 \ . \tag{5}$$

The left-hand side of Eq. (5) represents a Boolean function F_α of the parameters α_i only; the constraint equation (4) is therefore equivalent to

$$F_\alpha = 1 \ . \tag{6}$$

The conversion of Eq. (4) into Eq. (6) is known in the literature as *Petrick's method* [7].

Two properties of two-level synthesis are worth remarking in the context of this paper. First, once the list of primes has been built, we are guaranteed that no solution will violate the upper bound in Eq. (1), so that only the lower bound needs to be considered (as explicited by Eq. (4)). Similarly, only the upper bound needs to be considered during the extraction of primes. Second, the effect of adding/removing a cube from a partial cover of $\mathbf{F}_{min}$ is always predictable: that partial cover is increased/decreased. This property eases the problem of sifting the primes during the covering step, and it is reflected by the unateness of F_α: intuitively, by switching any parameter α_i from 0 to 1, we cannot decrease the chances of satisfying Eq. (6). These are important attributes of the problem that need to be preserved in its generalizations.

Don't care -based Multiple-level Optimization

Two-level optimization is the basic engine in *don't care* -based multiple-level logic optimization, where it is used to iteratively optimize single-output gates in the network.

Consider a single-output subnetwork, with local output y, to be re-synthesized. The primary output $\mathbf{F}$ of the overall network can be expressed in terms of the signal y:

$$\mathbf{F} = \mathbf{F}(\mathbf{x}, y) = y'\mathbf{F}_{y'} + y\mathbf{F}_y = (y\mathbf{1} + \mathbf{F}_{y'})(y'\mathbf{1} + \mathbf{F}_y). \tag{7}$$

By replacing Eq. (7) in Eq. (1), it follows that y must satisfy:

$$\mathbf{F}_{min} \subseteq y'\mathbf{F}_{y'} + y\mathbf{F}_y \subseteq \mathbf{F}_{max}. \tag{8}$$

A constraint on y similar to Eq. (1) can be obtained from Eq. (8) as follows. The upper bound in Eq. (8) holds if and only if $y'\mathbf{F}_{y'} \subseteq \mathbf{F}_{max}$ and $y\mathbf{F}_y \subseteq \mathbf{F}_{max}$, *i.e.*

$$y' \subseteq \mathbf{F}_{max} + \mathbf{F}'_{y'}; \quad y \subseteq \mathbf{F}_{max} + \mathbf{F}'_y. \tag{9}$$

Eq. (9) can be rewritten as

$$\mathbf{F}'_{max}\mathbf{F}_{y'} \subseteq y\mathbf{1} \subseteq \mathbf{F}_{max} + \mathbf{F}'_y. \tag{10}$$

Similarly, the lower bound holds if and only if $\mathbf{F}_{y'} + y\mathbf{1} \supseteq \mathbf{F}_{min}$ and $\mathbf{F}_y + y'\mathbf{1} \supseteq \mathbf{F}_{min}$, *i.e.*

$$\mathbf{F}_{min}\mathbf{F}_{y'} \subseteq y\mathbf{1} \subseteq \mathbf{F}'_{min} + \mathbf{F}_y \tag{11}$$

Eq. (10) and (11) can be merged together, to obtain:

$$\mathbf{F}_{min}\mathbf{F}'_{y'} + \mathbf{F}'_{max}\mathbf{F}_{y'} \subseteq y\mathbf{1} \subseteq (\mathbf{F}_{max} + \mathbf{F}'_y)(\mathbf{F}'_{min} + \mathbf{F}_y). \tag{12}$$

Eq. (12) represents the exact degrees of freedom available in the synthesis of the signal y, and is formally identical to Eq. (1): the value of y is undetermined corresponding to those points for which the lower bound differs from the upper bound. Such points are the local *don't cares* for y, and are denoted by $\mathbf{DC}_y(\mathbf{x})$. Once the bounds (or, equivalently, the *don't cares*) for y are computed, ordinary two-level synthesis algorithms can be applied.[1]

Boolean Relations-based Multiple-level Optimization

Don't care -based methods allow the optimization of only one single-output subnetwork at a time. It has been shown in [6] that this strategy may potentially produce lower-quality results with respect to a more general approach attempting the simultaneous optimization of multiple-output subnetworks.

Let $\mathbf{y} = [y_1, y_2, \cdots, y_m]$ denote the outputs of a subnetwork, to be re-synthesized, and let $\mathbf{F}(\mathbf{x}, \mathbf{y})$ denote the network outputs, expressed in terms of the variables y_i. From equation(1), the functional constraints on $\mathbf{y}$ are expressed by

$$\mathbf{F}_{min} \subseteq \mathbf{F}(\mathbf{x}, \mathbf{y}) \subseteq \mathbf{F}_{max}. \tag{13}$$

An equation like Eq. (13) describes a **Boolean Relation**[2]. The synthesis problem consists of finding a minimum-cost realization of $y_1, \ldots, y_m$ such that Eq. (13) holds. An exact solution algorithm, targeting two-level realizations, is presented in [6].

The difficulty with solutions to Boolean relation is twofold: First, when trying to express Eq. (13) in a form similar to Eq. (12), the isolation of a particular y_i results in dependence of y_j in the upper and lower bounds of the expression. This makes simultaneous optimization of $y_1 \ldots y_m$ difficult. Second, Eq. (13) requires a solution to the *binate covering problem* in the covering step. Fast binate covering solvers are the subject of ongoing research [8]. Nevertheless, the binate nature of the covering step reflects an intrinsic complexity which is not found in the unate case. In particular, since $\mathbf{F}$ is in general binate with respect to $\mathbf{y}$, the effect of adding / removing a prime to a partial solution is no longer trivially predictable, and both bounds in Eq. (13) may be violated by the addition of a single cube. As a consequence, branch-and-bound solvers may (and usually do) undergo many more backtracks than with a unate problem of comparable size, resulting in a substantially increased CPU time.

[1]In practice, y is re-synthesized by taking advantage also of the other internal signals available in the network. Implicants and primes are in this context expressed in terms of primary inputs and other network variables.

[2]An alternative formulation of a Boolean Relation is by means of a **characteristic equation**: $R(\mathbf{x}, \mathbf{y}) = 1$, where R is a Boolean function. It could be shown that the two formulations are equivalent.

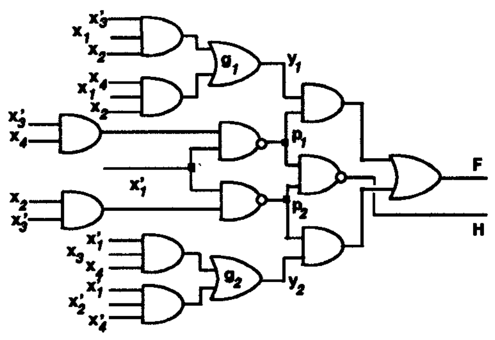

Figure 1: Gates g_1 and g_2 are compatible.

3 Compatible Gates

The analysis of Boolean relations points out that binate problems arise because of the generally binate dependence of $\mathbf{F}$ on the variables y_i. We introduce the notion of **compatible gates** in order to perform multiple-gate optimization while avoiding the binate covering problem. In the rest of the paper, given a network output expression $\mathbf{F}(\mathbf{x}, \mathbf{y})$, $\mathbf{x}$ is the set of input variables and $\mathbf{y}$ is the set of gate outputs to be optimized.

Definition 1 *A subset of gates $S = \{g_1, \ldots, g_k\}$ in a Boolean network with outputs $y_1 \ldots y_m$ is said to be* **compatible** *if the function $\mathbf{F}$ can be expressed in terms of the network outputs as*

$$\mathbf{F}_k = \sum_{j=1}^{m} y_j \mathbf{p}_j + \mathbf{q}; \tag{14}$$

modulo a phase change in the variables y_j or $\mathbf{F}$. In Eq. (14), the functions $p_j = p_j(x_1, \ldots, x_n)$ and $q = q(x_1, \ldots, x_n)$ do not depend on $y_1, \ldots, y_m$.

Compatible gates allow optimization of multiple gates without having to solve the binate covering problem. Intuitively, compatible gates are selected such that their optimization can only affect the outputs in a monotonic or unate way, and thereby forcing the covering problem to be unate.

Example 1 *Consider the two-output circuit in Figure 1. Gates g_1 and g_2 are compatible because F and H can be written as*

$$F = (x_1 + x_3 + x_4')y_1 + (x_1 + x_2' + x_3)y_2,$$

$$H = 0y_1 + 0y_2 + ((x_1 + x_3 + x_4')(x_1 + x_2' + x_3))';$$

The compatibility of a set S of gates is a Boolean property. In order to ascertain it, one would have to verify that all network outputs can indeed be expressed as in Definition (3). This task is potentially very CPU-intensive. In Appendix A, we present algorithms for constructing subsets of compatible gates from the network topology only.

3.1 Optimizing Compatible Gates

The functional constraints for a set of compatible gates can be obtained by replacing Eq. (14) into Eq. (13). From Eq. (14) we obtain:

$$\mathbf{F}_{min} \subseteq \sum_{j=1}^{m} y_j \mathbf{p}_j + \mathbf{q} \subseteq \mathbf{F}_{max}, \tag{15}$$

Eq. (15) can be solved using steps similar to that of two-level optimization. In particular, the optimization steps consist of *implicant extraction* and the *covering* steps.

Implicant Extraction

Assuming that $\mathbf{q} \subseteq \mathbf{F}_{max}$, the upper bound of Eq. (15) holds *if and only if* for each product $y_j \mathbf{p}_j$ the inequality

$$y_j \mathbf{p}_j \subseteq \mathbf{F}_{max}$$

is verified, *i.e.* if and only if

$$y_j \mathbf{1} \subseteq \mathbf{F}_{max} + \mathbf{p}_j'; \quad j = 1, \ldots, m. \tag{16}$$

	Primes	Influence sets
c_1	$x_1'x_2'x_3$	y_1, y_2
c_2	$x_1'x_2'x_4'$	y_1, y_2
c_3	$x_1'x_3x_4$	y_1, y_2
c_4	x_2x_4	y_1, y_2
c_5	$x_1x_2x_3'$	y_1, y_2
c_6	x_2x_3'	y_2
c_7	$x_1'x_3'x_4'$	y_2
c_8	$x_1'x_2'$	y_1
c_9	$x_1'x_4$	y_1

Table 1: Multiple-output primes for Example (2).

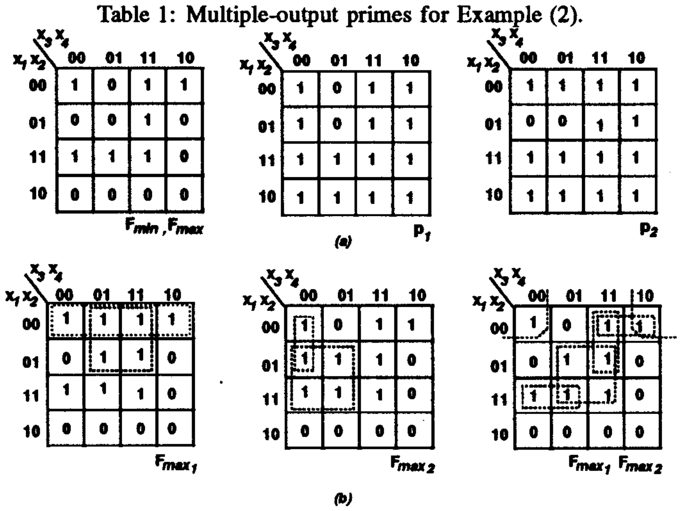

Figure 2: (a): Maps of $F_{min}, F_{max}, p_1, p_2$. (b) Maps of $F_{max,1}, F_{max,2}$ and of the product $F_{max,1}F_{max,2}$. Primes of y_1 and y_2 are shown in the maps of $F_{max,1}$ and $F_{max,2}$, respectively. The map of $F_{max,1}F_{max,2}$ shows the primes common to y_1 and y_2.

or, equivalently,

$$y_j \subseteq F_{max,j}; \quad j = 1, \ldots, m \tag{17}$$

where $F_{max,j}$ is the product of all the components of $\mathbf{F}_{max} + \mathbf{p}_j'$. A cube c can thus appear in a two-level expression of y_j if and only if $c \subseteq F_{max,j}$. As this constraint is identical to Eq. (3), the prime-extraction strategies [7, 1] of ordinary two-level synthesis can be used.

Example 2 *Consider the optimization problem for gates g_1 and g_2 in Fig. (1).*

From Example (1),

$$p_1 = (x_1 + x_3 + x_4')';$$

$$p_2 = (x_1 + x_2' + x_3)'.$$

We assume no external don't care set. Consequently, $F_{min} = F_{max} = x_1x_2x_3' + x_2x_3x_4 + x_1'x_2'(x_3 + x_4')$. The Karnaugh maps of F_{min} and F_{max} are shown in Fig. (2a), along with those of p_1 and p_2. Fig. (2b) shows the maps of $F_{max,1} = F_{max} + p_1'$ and $F_{max,2} = F_{max} + p_2'$, used for the extraction of the primes of y_1 and y_2, respectively. The list of all multiple-output primes is given in Table (1). Note that primes 1 through 5 can be used by both y_1 and y_2.

Covering Step

Let N indicate the number of primes. For example, in the problem of Example (2), $N = 9$. We then impose a sum-of-products representation associated with each variable y_j:

$$y_j = \sum_{k=1}^{N} \alpha_{jk} c_k \tag{18}$$

with the only restriction that $\alpha_{jk} = 0$ if y_j is not in the influence set of c_k. Since the upper bound of Eq. (15) is now satisfied by construction (*i.e.* by implicant computation), the minimization of

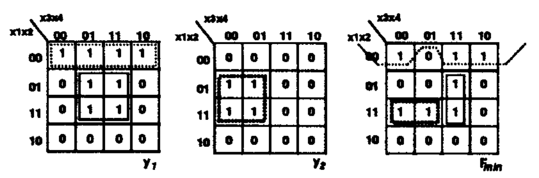

Figure 3: A minimum-cost solution for the covering of $\mathbf{F}_{min}$.

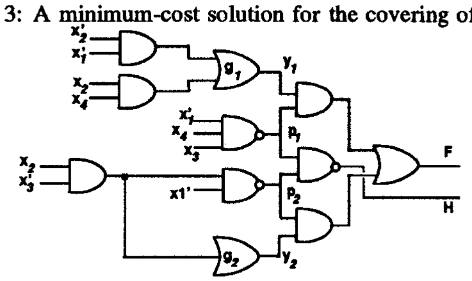

Figure 4: Network resulting from the simultaneous optimization of compatible gates g_1 and g_2.

$y_1, \ldots, y_m$ can be formulated as a minimum-cost covering problem

$$\mathbf{F}_{min} \subseteq \mathbf{q} + \sum_{j=1}^{m}\sum_{k=1}^{N} \alpha_{jk} c_k \mathbf{p}_j \qquad (19)$$

whose similarity with Eq. (4) is evident, the products c_{jk} $\mathbf{p}_j$ now playing the role of the primes of two-level synthesis.

Example 3 *In the optimization problem of Example (2), we are to solve the covering problem*

$$F_{min} \subseteq p_1y_1 + p_2y_2.$$

Using the set of primes found in Example (2), y_1 and y_2 are expressed by

$$y_1 = \alpha_{1,1}c_1 + \alpha_{1,2}c_2 + \alpha_{1,3}c_3 + \alpha_{1,4}c_4 + \alpha_{1,5}c_5 + \alpha_{1,8}c_8 + \alpha_{1,9}c_9$$

$$y_2 = \alpha_{2,1}c_1 + \alpha_{2,2}c_2 + \alpha_{2,3}c_3 + \alpha_{2,4}c_4 + \alpha_{2,5}c_5 + \alpha_{2,6}c_6 + \alpha_{2,7}c_7$$

The optimum solution has cost 6 and is given by $y_1 = x_1'x_2' + x_2x_4$; $y_2 = x_2x_3'$, corresponding to the assignments

$$\alpha_{1,1} = \alpha_{1,2} = \alpha_{1,3} = \alpha_{1,5} = \alpha_{1,9} = 0; \quad \alpha_{1,4} = \alpha_{1,8} = 1;$$

$$\alpha_{2,1} = \alpha_{2,2} = \alpha_{2,3} = \alpha_{2,4} = \alpha_{2,5} = \alpha_{2,7} = 0; \quad \alpha_{2,6} = 1.$$

The initial cost, in terms of literals, was 12. The solution corresponds to the cover shown in Fig. (3), and resulting in the circuit of Fig. (4).

It is worth contrasting, in the above example, the role of y_1 and y_2 in covering F_{min}. Before optimization, p_1y_1 covered the minterms $x_1x_2x_3'x_4'$, $x_1x_2x_3'x_4$, $x_1x_2x_3x_4$ of F_{min}, while p_2y_2 covered $x_1'x_2'x_3'x_4'$, $x_1'x_2'x_3x_4'$, $x_1'x_2x_3x_4$, $x_1'x_2'x_3x_4$. After optimization, y_1 and y_2 essentially "switched role" in the cover: p_2y_2 is now used for covering $x_1x_2x_3'x_4'$, $x_1x_2x_3'x_4$, while p_1y_1 covers all other minterms.

In the general case, the possibility for any of $y_1, \ldots, y_m$ to cover a minterm of F_{min} is evident from Eq. (15). Standard single-gate optimization methods based on *don't care* [5] regard the optimization of each gate $g_1, \ldots, g_m$ as separate problems, and therefore this degree of freedom is not used. For example, in the circuit of Fig. (1), the optimization of g_1 is distinct from that of g_2. The *don't care* conditions associated to (say) y_1 are those minterms for which either $p_1 = 0$ or such that $p_2y_2 = 1$, and are shown in the map of Fig. (5), along with the initial cover. It can immediately be verified that y_1 can only be optimized into $x_1x_2x_3' + x_2x_4$, saving only one literal.

The *don't cares* for y_2 are also shown in Fig. (5). No optimization is possible in this case. Note also that the optimization result is (in this particular example) independent from the order in which g_1 and g_2 are optimized. Unlike the compatible gates case, it is impossible for the covers of y_1 and y_2 to "switch" role in covering F_{min}.

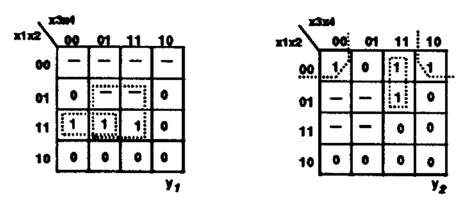

Figure 5: *Don't care* conditions associated with y_1 and y_2: only 1 literal can be removed.

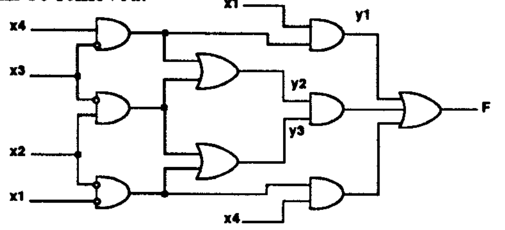

Figure 6: Network for Example (4).

4 Unate Optimization

In the previous section we showed that in the case of compatible gates, the functional constraints expressed by Eq. (13) can be reduced to Eqs. (17) and (19), which could be solved by a two-step procedure similar to that of two-level optimization. We now generalize the compatible gates results to the optimization of arbitrary subsets S of unate gates.

4.1 Optimizing Unate Subsets

Assume, for the sake of simplicity, that F is positive unate with respect to $y_1, \ldots, y_m$. We can perform optimization on the subset of unate gates in a style that is totally analogous to compatible gates by dividing it into *implicant extraction* and *covering* steps.

Implicant Extraction

In this step, for each y_i to be optimized, a set of *maximal functions* is extracted. In particular, the maximal functions of each each y_i can be expressed as Eq. (20), which is similar to Eq. (17).

$$y_i \subseteq F_{max,j}; j = 1, \ldots, m; \qquad (20)$$

From Eq. (20), appropriate implicants can then be extracted.

Intuitively, the maximal functions are the largest functions that can be used while satisfying the bound $\mathbf{F} \subseteq \mathbf{F}_{max}$. Therefore, they represent the upper bounds on y_i. We introduce the following definition:

Definition 2 *A set of functions*

$$\{F_{max,1}(\mathbf{x}), F_{max,2}(\mathbf{x}), \ldots, F_{max,m}(\mathbf{x})\}$$

is said to be **maximal** *if*

$$\mathbf{F}(\mathbf{x}, F_{max,1}(\mathbf{x}), F_{max,2}(\mathbf{x}), \ldots, F_{max,m}(\mathbf{x})) \subseteq \mathbf{F}_{max} \qquad (21)$$

and the inequality (21) is violated only when any F_{max_j} is replaced by a larger function $\hat{\mathbf{F}} \supset F_{max_j}$.

By substituting the maximal functions for each y_i, Eq. (13) can be reduced to the following:

$$\mathbf{F}_{min} \subseteq \mathbf{F}(\mathbf{x}, \mathbf{y}) \qquad (22)$$

The remaining task is to find a minimum-cost covering solution for Eq. (22). The following theorem, whose proof is in [9], provides means for finding a set of maximal functions. It also shows that computing such functions has complexity comparable with computing ordinary *don't care* sets.

Theorem 4.1 *Maximal functions can be obtained as*

$$F_{max,j} = f_j + DC_j \qquad (23)$$

where DC_j represents the don't care set associated with y_j, assuming that $y_k = F_{max,k}, k = 1, \ldots, j-1$ and $y_k = f_k; k = j+1, \ldots, m$.

This theorem states that the maximal function for vertex i depends on the maximal functions already calculated ($j < i$). This means that unlike the case of compatible gates, maximal function for a given vertex is not unique.

Example 4 *For the network of Fig. (6), assuming no external don't care conditions, we find the maximal functions for y_1, y_2, and y_3. The DC_{y_i} terms correspond to the observability don't care at y_i, computed using the F_{max} of the previous gates.*

$$y_1 = x_1 x_3' x_4; \quad y_2 = x_3'(x_4 + x_2); \quad y_3 = x_3' x_2 + x_1' x_2'$$

Maximal functions derived by Theorem (4.1) are :

$$F_{max,1} = x_1 x_3' x_4 + DC_{y_1} = x_3' x_4 + (x_3' + x_4) x_1' x_2'$$

$$\begin{aligned} F_{max,2} &= x_3'(x_4 + x_2) + DC_{y_2}(y_1 = F_{max,1}) \\ &= x_4 + x_3' x_2' + x_1 x_2' + x_3 x_2 \end{aligned}$$

$$\begin{aligned} F_{max,3} &= x_3' x_2' + x_1' x_2 + DC_{y_3}(y_1 = F_{max,1}, y_2 = F_{max,2}) \\ &= x_3' x_2 + x_1' x_2' + x_4 x_3' \end{aligned}$$

Covering Step

Eq. (20) allows us to find a set of multiple-output primes for $y_1, \ldots, y_m$. The covering step then consists of finding a minimum-cost sum such that Eq. (22) holds.

We now present a reduction for transforming the covering step to the one presented for compatible gates. We first illustrate the reduction by means of an example.

Example 5 *In Fig. (6), consider the combination of inputs $\mathbf{x}$ resulting in $F_{min}(\mathbf{x}) = 1$. To each such combination we can associate the set of values of y_1, y_2, y_3 such that $F(\mathbf{x}, \mathbf{y}) = 1$. For instance, for the entry $x_1 x_2 x_3 x_4 = 1001$, it must be $F_{x_1 x_2' x_3' x_4}(\mathbf{y}) = y_1 + y_2 y_3 = 1$. Let us now denote with $G(\mathbf{y})$ the left-hand side of this constraint, i.e. $G(\mathbf{y}) = y_1 + y_2 y_3$. Notice that $G(\mathbf{y})$ is unate in each y_i and changes depending on the combination of values currently selected for x_1, x_2, x_3, x_4.*

Any constraint $G(\mathbf{y}) = 1$ can be represented in a canonical form:

$$G(\mathbf{y}) = (G_{y_1' y_2' y_3'} + y_1 + y_2 + y_3)(G_{y_1' y_2' y_3} + y_1 + y_2) \\ \ldots (G_{y_1 y_2 y_3'} + y_3) G_{y_1 y_2 y_3} = 1$$

which, in turn, is equivalent to the 8 constraints

$$\begin{array}{lllll} G_{y_1' y_2' y_3'} + & y_1 + & y_2 + & y_3 & = 1 \\ G_{y_1' y_2' y_3} + & & y_2 + & y_3 & = 1 \\ \ldots & & & & \\ G_{y_1 y_2 y_3'} + & & & y_3 & = 1 \\ G_{y_1 y_2 y_3} & & & & = 1 \end{array} \qquad (24)$$

By introducing an auxiliary variable z_i for each y_i, we can rewrite Eq. (24) as:

$$G(\mathbf{z}) + z_1' y_1 + z_2' y_2 + z_3' y_3 = 1 \quad \forall\ z_1, z_2, z_3\ .$$

or, equivalently,

$$G'(\mathbf{z}) \subseteq z_1' y_1 + z_2' y_2 + z_3' y_3\ .$$

In this particular example, we get

$$(z_1 + z_2 z_3)' \subseteq z_1' y_1 + z_2' y_2 + z_3' y_3\ .$$

More generally, corresponding to each combination $\mathbf{x}$ such that $F_{min}(\mathbf{x}) = 1$, the constraint $F(\mathbf{x}, y) = 1$ can be re-expressed as

$$F(\mathbf{x}, \mathbf{z}) + z_1' y_1 + z_2' y_2 + \ldots + z_m' y_m = 1\ .$$

The transformation shown in Example (5) is formalized by the following theorem (whose proof can be found in [9]):

Theorem 4.2 *Let $z = [z_1, \ldots, z_m]$ denote m auxiliary Boolean variables. Eq. (22) holds if and only if*

$$\mathbf{F}_{min} \subseteq \mathbf{F}(\mathbf{x}, \mathbf{z}) + \sum_{j=1}^{m} y_j (z_j' \mathbf{1}) \qquad (25)$$

Eq. (25) has the same format of Eq. (15), with $\mathbf{q}$ and $\mathbf{p}_j$ being replaced by $\mathbf{F}(\mathbf{x}, \mathbf{z})$ and $z_j' \mathbf{1}$, respectively. Theorem (4.2) thus allows us to reduce the covering step to the one used for compatible gates. Theorems (4.1) and (4.2) show that the algorithms presented in Section 3 can be used to optimize arbitrary sets of gates with the same parity, without being restricted to sets of compatible gates only.

5 Implementation and Results

The implementation of the algorithms presented in Sections 3 and 4 is as follows. The original networks are first transformed into a unate, NOR-only description. All internal functions are represented using BDDs [10]. For each unoptimized gate g_i, the following heuristic is used. First, we try to find a set of comptible gates for g_i, called S_c. In the case where not enough compatible gates can be found, we find a set of gates that are unate with respect to g_i, called S_a.

In the case where S_c is optimized, we use Eq. (14) to extract the functions $\mathbf{p}_j$ and $\mathbf{q}$. In particular, $\mathbf{q}$ is extracted by simulating the network with outputs y_j stuck-at 0. The functions $\mathbf{p}_j$ are then extracted by simulating the network with y_j stuck-at 1, with $y_i; i \neq j$ stuck-at 0.

In the case of optimizing arbitrary unate network S_a, Theorem (4.1) is used to determine the maximal functions for each y_j. Note that optimizing S_c is preferable because for a set of m compatible gates, $m + 1$ simulations are needed to obtain all the required *don't cares* . For S_a, two simulations (with y_j stuck-at-0 and stuck-at-1) are required for the extraction of the *don't care* set of each variable y_j, resulting in a total of $2m$ simulations.

A set of primes for the gate outputs is then constructed. Because of the large possible set of primes, we limit our prime selection to single-literal primes only. The BDD of $\mathbf{F}(\mathbf{x}, \mathbf{z})$ is then built, and the covering problem solved. Networks are then iteratively optimized until no improvement occurs, and eventually folded back to a binate form. The algorithms presented in this paper were implemented in C program called *achilles*, and tested against a set of MCNC synthesis benchmarks.

Table (2) provides a comparison of *achilles* with SIS using *script.rugged*. The column *Initial Stat.* lists the network statistics before optimization, where *Int.* is number of internal interconnections and *gates* is the gate count. The column *Interconn.* shows number of interconnections after optimization. The *gates* column compares final gate counts. *Literal* column shows the final literals in factored form. The results in the table show that *achilles* performs better than SIS for all figures of merits. In particular, *achilles* does 11% better than SIS in factored literals.

Note that *script.rugged* was chosen because it is the most robust script of the SIS script suite, and it matches closely to our type of optimization. Our objective was to compare optimization results based only on Boolean operations, namely compatible gates versus *don't cares* . The *script.rugged* calls *full_simplify*[11], which computes observability *don't cares* to optimize the network.

The table shows that the *achilles* runtimes are competitive with that of SIS. In this first implementation, we are more interested in the quality of the optimization than the efficiency of the algorithms, therefore an *exact* covering solver is used. We can improve the runtime in the future by substituting a faster heuristic or approximate solvers (such as used in ESPRESSO [1]).

6 Conclusion

In this paper we presented a comparative analysis of approaches to multi-level logic optimization, and presented new algorithms for simultaneous multiple-gate optimization. The algorithms are based on the notion of **compatible gates**. We identify the main advantage of the present approach over previous solutions in its capability of exact minimization of suitable multiple-output networks, by means of traditional two-level optimization algorithms. Preliminary experimental results show an improvement of 11% over existing methods.

Appendix A Finding Compatible Gates

In this section, we describe an algorithm for finding compatible gates based on network topology. In this analysis, we make the assumption that the network is transformed into its equivalent NOR-only form. In this case, the parity of a path is simply the parity of the path length.

In defining Equation (14) for compatible gates, it is evident that the dependency of $\mathbf{F}$ on $y_1, \ldots, y_m$ must be unate. In order to increase the chances of finding sets of compatible gates, it is thus convenient to transform a network into an internally unate one. This

	Initial Stat.		Interconn.		Literals(fac)		Gates		*CPU time*	
Circuit	Int.	Gates	Achilles	SIS	Achilles	SIS	Achilles	SIS	Achilles	SIS
cm85a	108	63	67	77	42	46	31	34	1.5	1.2
cm162a	113	60	99	102	47	49	41	52	1.8	1.3
pm1	130	60	67	78	47	52	31	36	1.6	1.3
9symml	375	152	288	325	163	186	88	101	108.4	64.2
alu2	924	262	366	570	303	362	215	231	309.7	403.0
alu4	1682	521	902	1128	612	703	420	487	1612.6	1718.5
apex6	1141	745	1009	1315	687	743	589	639	115.1	30.3
C499	945	530	913	945	505	552	498	530	202.1	133.6
C880	797	458	643	731	355	409	295	342	340.6	30.7
C1908	936	489	828	891	518	542	445	482	422.1	138.8

Table 2: Optimization Results. Runtimes are in seconds on DEC5000/240.

is done by duplicating those gates whose fanouts contain reconvergent paths with different inversion parity. The resulting network is therefore at most twice the size of the original one. In practice, the increase is smaller.

Definition 3 *A network is termed* **unate** *with respect to a gate* g *if all reconvergent paths from* g *have the same parity of inversions. A network is* **internally unate** *if it is unate with respect to each of its gates. All paths from g to a primary output z_i in an internally unate network has parity π_i, which is defined to be the* **parity of** g *with respect to z_i.*

Theorem (3.1) below provides a sufficient conditions for a set S of gates to be compatible. Without loss of generality, the theorem is stated in terms of networks with one primary output. The following auxiliary definitions are required:

Definition 4 *The* **fanout gate set** *and* **fanout edge set** *of a gate g, indicated by $FO(g)$ and $FOE(g)$, respectively, are the set of gates and interconnections contained in at least one path from g to the primary output.*
The **fanout gate set** *and* **fanout edge set** *of a set of gates $S = \{g_1, \dots, g_k\}$, indicated by $FO(S)$ and $FOE(S)$, respectively, are:*

$$FO(S) = \bigcup_{i=1}^{|S|} FO(g_i); \quad FOE(S) = \bigcup_{i=1}^{|S|} FOE(g_i); \tag{26}$$

Theorem A.1 *In a NOR-only network, let $S = \{g_1, \dots, g_m\}$ be a set of gates of parity π, and not in each others' fanout. Let $y_1, \dots, y_m$ denote the respective outputs. The following propositions hold:*

1) if each gate in $FO(S)$ with parity π has at most one input interconnection in $FOE(S)$, then the primary outputs can be expressed as in Eq. (14) for some suitable functions p_j and q;

2) if each gate in $FO(S)$ with parity π' has at most one input in $FOE(S)$, then it can be shown that the output can be expressed as in Eq. (14).

The proof can be found in [9].

Theorem (A.1) also provides a technique for constructing a set of compatible gates directly from the network topology, starting from a "seed" gate g and a parameter ($rule$) that specifies the desired criterion of Theorem (A.1) (either 1 or 2) to be checked during the construction. The algorithm is as follows:

```
$COMPATIBLES(g, rule)$
$label\_fanout(g, FO)$;
$S = \{g\}$;
for($i = 0; i \leq NGATES; i++$) {
  if (($is\_labeled(g_i) = FALSE$) & ($parity(g_i) = parity(g)$)) {
    $label\_fanout(g_i, TMP)$;
    $compatible = dfs\_check(g_i, parity(g), rule)$
    if($compatible$) {
      $label\_fanout(g_i, FO)$;
      $S = S \cup \{g_i\}$;
    }
  }
}
```

$COMPATIBLES$ starts by labeling "FO" the fanout cone of g, as no gates in that cone can belong to a compatible set containing g. Labeled gates represents elements of the set $FO(S)$. All gates g_i that are not yet labeled and have the correct parity are then examined for insertion in S. To this purpose, the fanout of g_i that is not already in $FO(S)$ is temporarily labeled "TMP", and then visited by dfs_check in order to check the satisfaction of $rule$. If g_i is compatible, it becomes part of S and its fanout is merged with $FO(S)$. The depth-first traversal of dfs_check is stopped whenever the primary outputs or gates already in $FO(S)$ are reached, or a violation of $rule$ is detected.

References

[1] R. Rudell and A. Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization," *IEEE Transactions on CAD/ICAS*, vol. 6, no. 5, pp. 727–750, Sept. 1987.

[2] E. L. Lawler, "An approach to multilevel boolean minimization," *ACM Journal*, vol. 11, no. 3, pp. 283–295, July 1964.

[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, Nov. 1987.

[4] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney, "The transduction method - design of logic networks based on permissible functions," *IEEE Transactions on Computers*, vol. 38, no. 10, pp. 1404–1424, Oct. 1989.

[5] 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.

[6] R. Brayton and F. Somenzi, "An exact minimizer for boolean relations," in *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pp. 316–319, Nov. 1989.

[7] E. J. McCluskey, *Logic Design Principles With Emphasis on Testable Semicustom Circuits*. Prentice-Hall, 1986.

[8] S. W. Jeong and F. Somenzi, "A new algorithm for the binate covering problem and its application to the minimization of boolean relations," in *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pp. 417–420, 1992.

[9] M. Damiani, J. Yang, and G. De Micheli, "Optimization of combinational logic circuits based on compatible gates," tech. rep., Computer Systems Laboratory, Stanford University, 1993.

[10] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677–691, Aug. 1986.

[11] H. Savoj, R. K. Brayton, and H. Touati, "Extracting local don't cares and network optimization," in *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pp. 514–517, Nov. 1991.