J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*

CIRCUIT OPTIMIZATION

The behavior of integrated circuits is influenced by the manner in which the circuit is built. For circuits that are constrained to meet performance and time-to-market requirements, the use of automated techniques is essential to manage the complexity. These parameters include the number of functional units in a behavioral specification, the logic structure of the network, the layout topologies used, and many other factors. The behavior of the circuit can be influenced by a number of such variable parameters, and a designer must carefully select the correct parameters that meet the requirements with a reasonable overhead and in time to meet market deadlines.

Optimization theory plays a large role in helping this process. There is a large body of literature on optimizing linear functions, nonlinear functions, combinatorial functions, etc., that can be successfully applied to these problems if appropriate models are used to fit the problem into any of these paradigms. It is important to note that in addition to finding a model that fits into a known optimization problem form, it is imperative that the model should be a good reflection of reality. This chapter assumes the presence of such models and shows how they may be applied to circuit optimization problems.

A brief intuitive feel for optimization can be provided by the following example. The problem of optimization can be thought of as a multidimensional version of trying to find the lowest or highest point on a topographical map. The possible solutions here are locations, specified as x and y coordinates, each of which is associated with a particular height. In general, the map would be specified by a function and the number of dimensions would be the number of parameters on which the function depends. Unconstrained optimization does not limit the search space at all, while constrained optimization limits the search to a specific area (for example, finding the lowest point in the state of Colorado). Continuous optimization allows all possible solutions, while discrete optimization permits only a few discrete points as solutions (for example, find the highest point in Colorado that lies within the limits of a city with a population of over 10,000). Multicriterion optimization attempts to find a balance of two objectives that should be optimized (for example, attempting to find the highest point in the state of Colorado that has the least snowfall). The criteria, as is easily seen from the example, can often be conflicting, and the problem has to be restated in a form that makes it more unambiguous.

In this article, we will first provide a survey of optimization algorithms, followed by a set of examples that illustrate the application of some of these algorithms on circuit optimization problems.

Optimization Algorithms

In this section, we survey a number of commonly used approaches for optimization. The vastness of this field makes it infeasible to enumerate or describe all of the methods. While we will treat many prominent methods in this chapter, several other methods, such as Newton's and modified Newton or quasi-Newton methods and conjugate gradient methods, which are often useful in engineering optimization, are not covered here. For these and more, the reader is referred to a standard text on optimization, such as Refs. 1 and 2.

For the circuit designer, it is often not necessary or desirable to implement a complicated optimization algorithm when an optimized public domain or commercially available piece of software is available for the same purpose. However, understanding the underlying algorithms often helps a user to better utilize the capabilities of the optimizer. Some prominent examples of such software include the public-domain tool LPSOL (3) and the commercial tool CPLEX for linear programming and tools such as MINOS (4) and LANCELOT (5) for nonlinear programming. Another valuable resource is Ref. 6, which provides a brief explanation and C code for many common computational tasks, including optimization; related books in the same series address numerical recipes in other programming languages.

Nonlinear Optimization Problems. The "standard form" of a *constrained nonlinear optimization problem* is

$$\begin{aligned} &\text{Minimize} \quad f(\boldsymbol{X}) : \boldsymbol{R}^n \to \boldsymbol{R} \\ &\text{subject to} \quad \boldsymbol{g}(\boldsymbol{x}) \le 0 \\ &\qquad \boldsymbol{g} : \boldsymbol{R}^n \to \boldsymbol{R}^m,\ \boldsymbol{x} \in \boldsymbol{R}^n \end{aligned} \tag{1}$$

representing the minimization of a function f of n variables under constraints specified by inequalities determined by functions $\boldsymbol{g} = [g_1 \cdots g_m]^T \cdot f$ and g_i are, in general, nonlinear functions, so that the linear programming problem is a special case of the above. The parameters $\boldsymbol{x}$ may, for example, represent circuit parameters, and $f(\boldsymbol{x})$ and $g_i(\boldsymbol{x})$ may correspond to circuit performance functions. In this equation, as in the rest of this paper, we will denote a real vector space of k dimensions by $\boldsymbol{R}^k$.

Note that $\ge$ inequalities can be handled under this paradigm by multiplying each side by -1, and equalities by representing them as a pair of inequalities. The maximization of an objective function function $f(\boldsymbol{x})$ can be achieved by minimizing $-f(\boldsymbol{x})$.

The set $\mathscr{F} = \{\boldsymbol{x} \mid \boldsymbol{g}(\boldsymbol{x}) \le \boldsymbol{0}\}$ that satisfies the constraints on the nonlinear optimization problem is known as the feasible set, or the feasible region. If $\mathscr{F}$ is empty (nonempty), then the optimization is said to be *unconstrained* (*constrained*).

Several mathematical programming techniques can be used to solve the optimization problem above; some of these are outlined here. For further details, the reader is referred to a standard text on optimization, such as Refs. 1 and 2. Another excellent source for optimization techniques and their applications to integrated circuit (*IC*) design is a survey paper by Brayton, Hachtel, and Sangiovanni-Vincentelli (7).

The formulation above may not directly be applicable to real-life design problems, where often multiple conflicting objectives must be optimized. In such a case, one frequently uses techniques that map on the problem to the form in Eq. (1) (see section entitled "Multicriterion Optimization and Pareto Criticality").

Basic Definitions. Apart from being able to evaluate a function, it is very important to determine information about its variations. Such information is typically captured in the derivatives of the function, if the function is smooth. In particular, the first and second derivatives play a large role in optimization. We define two terms in this context with respect to a continuous function $f(\boldsymbol{x})$, where $\boldsymbol{x} = [x_1 x_2 \cdots x_n]^T$.

Definition.

The *gradient* of a continuous and differentiable function $f(\boldsymbol{x})$, denoted as $\nabla f(\boldsymbol{x})$, is given by the $1 \times n$ vector

$$\nabla f(x) = \left[\frac{\partial f(x)}{\partial x_1} \frac{\partial f(x)}{\partial x_2} \cdots \frac{\partial f(x)}{\partial x_n} \right] \tag{2}$$

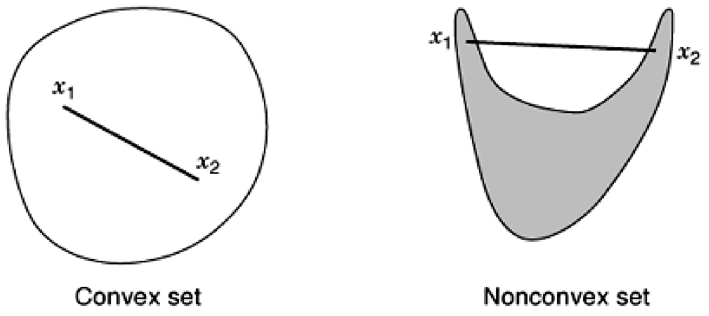

Fig. 1. Examples of convex and nonconvex sets.

Definition. The *Hessian* (sometimes also referred to as the *Jacobian*) of a continuous and twice differentiable function, denoted as $\nabla^2 f(\boldsymbol{x})$, is given by the $n \times n$ matrix

$$\nabla^2 f(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial^2 f(\boldsymbol{x})}{\partial^2 x_1} & \frac{\partial^2 f(\boldsymbol{x})}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(\boldsymbol{x})}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f(\boldsymbol{x})}{\partial x_1 \partial x_2} & \frac{\partial^2 f(\boldsymbol{x})}{\partial^2 x_2} & \cdots & \frac{\partial^2 f(\boldsymbol{x})}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(\boldsymbol{x})}{\partial x_1 \partial x_n} & \frac{\partial^2 f(\boldsymbol{x})}{\partial x_2 \partial x_n} & \cdots & \frac{\partial^2 f(\boldsymbol{x})}{\partial^2 x_n} \end{bmatrix} \tag{3}$$

In any discussion on optimization, it is virtually essential to understand the idea of a convex function and a convex set, since these have special properties, and it is desirable to formulate problems as convex programming problems, wherever it is possible to do so without an undue loss in modeling accuracy. (Unfortunately, it is not always possible to do so)

Definition.

A set C in $\boldsymbol{R}^n$ is said to be a *convex set* if, for every $\boldsymbol{x}_1, \boldsymbol{x}_2 \in C$, and every real number α, $0 \le \alpha \le 1$, the point $\alpha\boldsymbol{x}_1 + (1-\alpha)\boldsymbol{x}_2 \in C$.

This definition can be interpreted geometrically as stating that a set is convex if, given two points in the set, every point on the line segment joining the two points is also a member of the set. Examples of convex and nonconvex sets are shown in Fig. 1.

Two examples of convex sets are the following geometric bodies:

(1) An *ellipsoid* $E(\boldsymbol{x},\mathscr{B},r)$ centered at point $\boldsymbol{x}$ is given by the equation

$$\left\{\boldsymbol{y} \mid (\boldsymbol{y}-\boldsymbol{x})^T \mathcal{B}(\boldsymbol{y}-\boldsymbol{x}) \le r^2\right\} \tag{4}$$

If $\mathscr{B}$ is a scalar multiple of the unit matrix, then the ellipsoid is called a *hypersphere*.

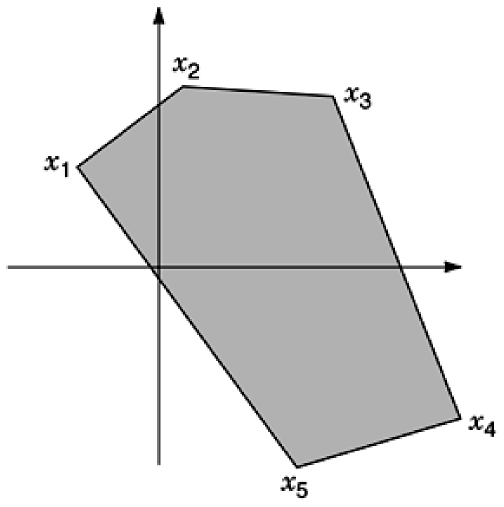

Fig. 2. The convex hull of five points.

(2) A (convex) *polytope* is defined as an intersection of half-spaces and is given by the equation

$$\mathcal{P} = \{\boldsymbol{x} \mid A\boldsymbol{x} \geq \boldsymbol{b}\}, \quad A \in \boldsymbol{R}^{m\times n}, \quad \boldsymbol{b} \in \boldsymbol{R}^{m} \tag{5}$$

corresponding to a set of m inequalities $\boldsymbol{a}^T{}_i\boldsymbol{x} \geq b_i$, $\boldsymbol{a}_i \in \boldsymbol{R}^n$.

Definition.

The *convex hull* of m points, $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m \in \boldsymbol{R}^n$, denoted co $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}$, is defined as the set of points $\boldsymbol{y} \in \boldsymbol{R}^n$ such that

$$\boldsymbol{y} = \sum_{i=1}^{m} \alpha_i \boldsymbol{x}_i, \qquad \alpha_i \geq 0 \;\forall\, i, \qquad \sum_{i=0}^{m} \alpha_i = 1 \tag{6}$$

The convex hull is the smallest convex set that contains the m points. An example of the convex hull of five points in the plane is shown by the shaded region in Fig. 2. If the set of points $\boldsymbol{x}_i$ is of finite cardinality (i.e., m is finite), then the convex hull is a polytope. Hence, a polytope is also often described as the convex hull of its vertices.

Definition.

A function f defined on a convex set Ω is said to be a *convex function* if, for every $\boldsymbol{x}_1$,$\boldsymbol{x}_2 \in \Omega$, and every α, $0 \leq \alpha \leq 1$,

$$f(\alpha\boldsymbol{x}_1 + (1-\alpha)\boldsymbol{x}_2) \leq \alpha f(\boldsymbol{x}_1) + (1-\alpha) f(\boldsymbol{x}_2). \tag{7}$$

f is said to be *strictly convex* if the inequality in Eq. (7) is strict for $0 < \alpha < 1$.

Geometrically, a function is convex if the line joining two points on its graph is always above the graph. Examples of convex and nonconvex functions on $\boldsymbol{R}^n$ are shown in Fig. 3.

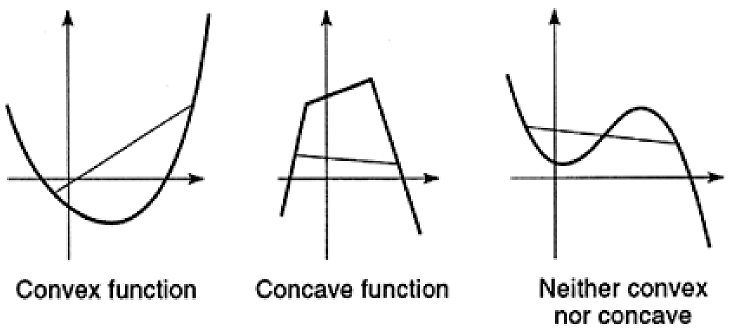

Fig. 3. Examples of functions that are convex, concave, or neither.

Definition.

A function g defined on a convex set Ω is said to be a *concave function* if the function $f = -g$ is convex. The function g is *strictly concave* if $-g$ is strictly convex.

Definition.

The *convex programming problem* is stated as follows:

$$\text{Minimize } f(\boldsymbol{x}) \tag{8}$$

$$\text{such that } \boldsymbol{x} \in S \tag{9}$$

where f is a convex function and S is a convex set. This problem has the property that any local minimum of f over S is a global minimum.

Definition.

A *posynomial* is a function h of a positive variable $\boldsymbol{x} \in \boldsymbol{R}^{\mathrm{n}}$ that has the form

$$h(\boldsymbol{x}) = \sum_j \gamma_j \prod_{i=1}^{n} x_i^{\alpha_{ij}} \tag{10}$$

where the exponents $\alpha_{ij} \in \boldsymbol{R}$ and the coefficients $\gamma_j > 0$.

For example, the function $3.7x^{1.4}{}_1x^3{}_2 + 1.8x^{-1}{}_1x^{2.3}{}_3$ is a posynomial in the variables x_1, x_2, x_3. Roughly speaking, a posynomial is a function that is similar to a polynomial, except that the coefficients γ_j must be positive, and an exponent α_{ij} could be any real number and not necessarily a positive integer, unlike a polynomial. A posynomial has the useful property that it can be mapped onto a convex function through an elementary variable transformation, $(x_i) = (e^{z_i})$ (8). Such functional forms are useful since in the case of an optimization problem where the objective function and the constraints are posynomial, the problem can easily be mapped onto a convex programming problem. A more inclusive function class with similar properties is the set of generalized posynomials (9).

Definition.

A generalized posynomial function $G_k(\boldsymbol{x})$, $\boldsymbol{x} \in \boldsymbol{R}^n$, where $k \geq 0$ is called the order of the function, is defined recursively as follows:

(1) A generalized posynomial of order 0, G_0, is simply the posynomial form defined earlier:

$$G_0(\boldsymbol{x}) = \sum_j \gamma_j \prod_{i=1}^{n} x_i^{\alpha_{ij}} \tag{11}$$

where the exponents $\alpha_{ij} \in \boldsymbol{R}$ and the coefficients $\gamma_j \in \boldsymbol{R}^+$.

(2) A generalized posynomial of order k is defined as

$$G_k(\boldsymbol{x}) = \sum_j \gamma_j \prod_{i=1}^{n} \left[G_{k-1,i}(\boldsymbol{x})\right]^{\alpha_{ij}} \tag{12}$$

where the exponents $\alpha_{ij} \in \boldsymbol{R}^+$ and the coefficients $\gamma_j \in \boldsymbol{R}^+$, and $G_{k-1,i}(\boldsymbol{x})$ is a generalized posynomial of order $k-1$.

Two features of the definition are noteworthy, and we explicitly point them out. First, while the exponents α_{ij} are unrestricted real numbers for G_0, they must necessarily be nonnegative for G_k, $k > 0$. Second, any generalized posynomial of order $k \geq 1$ is, by definition, also a generalized posynomial of order $l \geq k$. Therefore, in Eq. (12), $G_{k-1,i}$ may be any generalized posynomial whose order is no greater than $k-1$. As in the case of regular posynomials, the mapping $(x_i) = (e^{z_i})$ transforms a generalized posynomial of any order in the $\boldsymbol{x}$ space to a convex function in the $\boldsymbol{z}$ space.

Constrained Optimization Methods. Most problems in integrated circuit design involve the minimization or maximization of a cost function subject to certain constraints. In this section, a few prominent techniques for constrained optimization are presented. The reader is referred to Refs. 1,2, and 6 for details on unconstrained optimization.

Linear Programming. Linear programming is a special case of nonlinear optimization, and is the convex programming problem where the objective and constraints are all linear functions. The standard form of the problem is stated as

$$\begin{array}{lll} \text{Minimize} & \boldsymbol{c}^T\boldsymbol{x} & \\ \text{subject to} & A\boldsymbol{x} \geq \boldsymbol{b}, \quad \boldsymbol{x} \geq 0 & \\ & \boldsymbol{c}, \boldsymbol{x} \in \boldsymbol{R}^n, \quad \boldsymbol{b} \in \boldsymbol{R}^m, \quad A \in \boldsymbol{R}^{m\times n} & \end{array} \tag{13}$$

Although the requirement on the nonnegativity of $\boldsymbol{x}$ may appear to be a limitation at first, this does not mean that negative variables cannot be represented. For a variable x_i that may be negative, we may simply use the substitution $x_i = s_{i1} - s_{i2}$, where $s_{i1}, s_{i2} \geq 0$.

It can be shown that any solution to a linear program must necessarily occur at a vertex of the constraining polytope. The most commonly used technique for the solution of linear programs, the simplex method (1), is based on this principle. The computational complexity of this method can show an exponential behavior for pathological cases, but for most practical problems, it has been observed to grow linearly with the number of variables and sublinearly with the number of constraints. Algorithms with polynomial time worst-case complexity do exist; these include Karmarkar's method (10) and the Shor-Khachiyan ellipsoidal method (11). The computational complexity of the latter, however, is often seen to be impractical from a practical standpoint.

Every linear program is associated with a dual linear program. Duality is a symmetric relationship, so that the dual of the dual provides the primal. While it is generally true that any linear or nonlinear optimization problem has a dual form, only in case of a linear program is it always true that the optimal value of the dual is identical to the optimal value of the primal (original) problem. For a general nonlinear program, if we treat the primal as the minimization problem and the dual as the maximization problem, then the optimal value of the dual is less than or equal to that of the primal. The gap is referred to as the duality gap. For the primal form shown in Eq. (13), the dual is given by

$$\begin{array}{lll} \text{Maximize} & \lambda^T \boldsymbol{b} & \\ \text{subject to} & \lambda^T A \le \boldsymbol{c}^T, & \lambda \ge 0 \\ & \lambda \in \boldsymbol{R}^n. & \end{array} \tag{14}$$

If the values of $\boldsymbol{x}$ in Eq. (13) are restricted to the set of integers, the problem is referred to as an integer linear programming problem. It is important to note that integer linear programming is a harder problem than linear programming over a continuous space and that none of the existing algorithms for solving these problems shows polynomial time behavior, either in theory or empirically in practice, for a general integer linear program. However, for some special problem structures, such as shortest paths and network flows, polynomial time solutions do exist.

Network Flows. Network flow problems are a specific instance of a linear program that have an interpretation with respect to a graph construction. A network is a directed graph with two special vertices, namely, a source s and a sink t, such that every other vertex in the graph lies on a directed path from s to t. Each edges $e = (i,j)$ in the network is associated with a maximum nonnegative capacity, u_{ij}. In the absence of an edge, the capacity is considered to be zero. A flow through this network is a function that satisfies the following requirements.

Capacity constraints. For each edge $e = (i,j)$, $x_{ij} \le u_{ij}$, where u_{ij} is a constant.
Flow conservation. For each vertex i, $i \notin s, t$, the total inflow equals the total outflow, that is,

$$\sum_{e=(i,j)} x_{ij} = \sum_{e'=(k,i)} x_{ki} \tag{15}$$

The value of a flow f is given by $f = \Sigma_{e=(s,i)}\, x_{si} = \Sigma_{e'=(j,t)}\, x_{jt}$, and an objective function that is often used is to maximize f. Several problems can be formulated as maximum flow problems, and it is useful to obtain an intuitive feel for the problem statement. We may think of the source as a water pump of unlimited capacity, the sink as a reservoir of unlimited capacity, and the edges as pipes that have a limitation on how much water they may let through in a unit time. The problem of maximizing the flow is then that of determining the maximum volume of water per unit time that the entire network of pipes can let through.

A related problem is that of finding the minimum cut of a network. A cut is defined as a partition that divides the vertex set into two parts, X and Y, such that $s \in X$, $t \in Y$, and $X \cap Y = 0$. A minimum cut is one that minimizes the sum of the capacities of edges from X to Y. Intuitively, it is easy to see that the volume of water per unit time from the previous paragraph is limited by the bottleneck of the problem, which is the minimum cut (this fact is also provable). This leads to the following result, called the max-flow min-cut theorem: The value f of the maximum flow in a network is identical to the capacity of its minimum cut.

Interestingly, the problem of maximizing the flow in network with n vertices can be represented by a linear program as follows:

$$\begin{array}{ll} \text{Maximize} & f \\ \text{subject to} & \displaystyle\sum_{j=1}^{n} x_{ij} - \sum_{k=1}^{n} x_{ki} = \begin{cases} f & \text{if } i = 1 \\ 0 & \text{if } i \neq 1 \text{ or } n \\ -f & \text{if } i = n \end{cases} \\ & 0 \le x_{ij} \le u_{ij}, \qquad i, j = 1, 2, \ldots, n \end{array} \tag{16}$$

which is referred to a maximum flow problem. The special structure of this linear program can be exploited to provide fast polynomial-time solutions to the various network flow problems. For details, the reader is referred to Ref. 12.

A related problem is the minimum cost network flow, which can be stated as follows:

$$\begin{array}{ll} \text{Maximize} & \displaystyle\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \\ \text{subject to} & \displaystyle\sum_{j=1}^{n} x_{ij} - \sum_{k=1}^{n} x_{ki} = b_i, \qquad i = 1, 2, \ldots, n \\ & x_{ij} \ge 0, \qquad i, j = 1, 2, \ldots, n \end{array} \tag{17}$$

where c_{ij} and b_i are constants and the x_{ij}'s constitute the variables for this problem. Again, polynomial-time algorithms for solving this problem exist. A practically efficient algorithm that is not guaranteed to converge in polynomial time, but is empirically seen to, is an efficient adaptation of the simplex algorithm for linear programming, called the network simplex algorithm (12).

An interesting property of the network flow problems is that if the edge capacities are all integers (and so are the b_i's in the case of minimum cost flow problems), then an optimum integer solution exists and can be found in polynomial time. Therefore, for this specific structure, it is possible to solve the integer linear program in polynomial time, though this is not possible for general problem statements.

Lagrange Multiplier Methods. The Lagrangian multiplier methods are closely related to the first-order *Kuhn-Tucker* necessary conditions on optimality, which state that given an optimization problem of the form in Eq. (1), if f and $\boldsymbol{g}$ are differentiable at $\boldsymbol{x}^*$, then there is a vector $\lambda \in \boldsymbol{R}^m$, $(\lambda)_i \ge \mathbf{0}$, such that

$$\nabla f(\boldsymbol{x}^*) + \boldsymbol{\lambda}^T \nabla \boldsymbol{g}(\boldsymbol{x}^*) = 0 \tag{18}$$

$$\boldsymbol{\lambda}^T \boldsymbol{g}(\boldsymbol{x}^*) = \mathbf{0} \tag{19}$$

These correspond to $m + 1$ equations in $m + 1$ variables; the solution to these provides the solution to the optimization problem. The variables λ are known as the Lagrange multipliers. Note that since $g_i(\boldsymbol{x}^*) \le 0$, and because of the nonnegativity constraint on the Lagrange multipliers λ, it follows from Eq. (19) that $(\lambda)_i = 0$ for inactive constraints (constraints with $g_i(\mathbf{x}) < 0$).

Penalty Function Methods. Penalty function methods (13) convert the constrained optimization problem in Eq. (1) into an equivalent unconstrained optimization problem, since such problems are easier to solve than constrained problems, as

$$\text{Minimize } h(\boldsymbol{x}) = f(\boldsymbol{x}) + cP(\boldsymbol{x}) \qquad (20)$$

where $P(\boldsymbol{x}) : \boldsymbol{R}^n \rightarrow \boldsymbol{R}$ is known as a penalty function and c is a constant. The value of $P(\boldsymbol{x})$ is zero within the feasible region, and positive outside the region, with the value becoming larger as one moves farther from the feasible region; one possible choice when the $g_i(\boldsymbol{x})$'s are continuous is given by

$$P(\boldsymbol{x}) = \sum_{i=1}^{m} \max(0, -g_i(\boldsymbol{x})) \qquad (21)$$

For large c, it is clear that the minimum point of Eq. (20) will be in a region where P is small. Thus, as c is increased, it is expected that the corresponding solution point will approach the feasible region and minimize f. As $c \rightarrow \infty$, the solution of the penalty function method converges to a solution of the constrained optimization problem.

In practice, if one were to begin with a high value of c, one may not have very good convergence properties. The value of c is increased in each iteration until c is high and the solution converges.

Method of Feasible Directions. The method of feasible directions is an optimization algorithm that improves the objective function without violating the constraints. Given a point $\boldsymbol{x}$, a direction $\boldsymbol{d}$ is feasible if there is a step size $\overline{\alpha} > 0$ such that $\boldsymbol{x} + \alpha\boldsymbol{d} \in \mathscr{F} \; \forall \; 0 \le \alpha \le \alpha$, where $\mathscr{F}$ is the feasible region. More informally, this means that one can take a step of size up to $\overline{\alpha}$ along the direction $\boldsymbol{d}$ without leaving the feasible region. The method of feasible direction attempts to choose a value of α in a feasible direction $\boldsymbol{d}$ such that the objective function f is minimized along the direction, and α is such that x + $\alpha\boldsymbol{d}$ is feasible.

One common technique that uses the method of feasible directions is as follows. A feasible direction at $\boldsymbol{x}$ is found by solving the following linear program:

$$\begin{aligned} &\text{Minimize} && \epsilon \\ &\text{subject to} && \langle \nabla f(\boldsymbol{x}) \cdot \boldsymbol{d} \rangle \le \epsilon && (22) \\ & && \langle \nabla g_i(\boldsymbol{x}) \cdot \boldsymbol{d} \rangle \le \epsilon && (23) \\ &\text{and} && \text{normalization requirements on } \boldsymbol{d} \end{aligned}$$

where the second set of constraints are chosen for all $g_i \ge -b$, where b serves to incorporate the effects of near-active constraints to avoid the phenomenon of jamming (also known as zigzagging) (1). The value of b is brought closer to 0 as the optimization progresses. One common method that is used as a normalization requirement is to set $\boldsymbol{d}^T\boldsymbol{d} = 1$; others are given in Ref. 14. This constraint is nonlinear and nonconvex and is not added to the linear program as an additional constraint; rather, it is exercised by normalizing the direction $\boldsymbol{d}$ after the linear program has been solved. An appropriate step size in this direction is then chosen by solving a one-dimensional optimization problem.

Feasible direction methods are popular in finding engineering solutions because the value of $\boldsymbol{x}$ at each iteration is feasible, the algorithm can be stopped at any time without waiting for the algorithm to converge, and the best solution found so far can be used.

Multicriterion Optimization and Pareto Criticality

Most integrated circuit design problems involve tradeoffs between multiple objectives. In cases where one objective can be singled out as the most important one, and a reasonable constraint set can be defined in terms of the other objectives, the optimization problem can be stated using the formulation in Eq. (1). This is convenient since techniques for the solution of a problem in this form have been extensively studied and a wide variety of optimization algorithms are available.

Let $\boldsymbol{f}$ be a vector of design objectives that is a function of the design variables $\boldsymbol{x}$, where

$$\boldsymbol{f}(\boldsymbol{x}) : \boldsymbol{R}^n \rightarrow \boldsymbol{R}^m = (f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \cdots, f_m(\boldsymbol{x})) \tag{24}$$

It is extremely unlikely in a real application that all of the f_i's will be optimal at the same point, and hence one must trade off the values of the f_i's in a search for the best design point.

In this context, we note that at a point $\boldsymbol{x}$, we are interested in taking a step δ in a direction $\boldsymbol{d}$, $\|\boldsymbol{d}, \| = 1$, so that

$$f_i(\boldsymbol{x} + \delta\boldsymbol{d}) \leq f_i(\boldsymbol{x}) \quad \forall \quad 1 \leq i \leq m \tag{25}$$

A *Pareto critical point* is defined as a point $\boldsymbol{x}$ where no such small step of size less than δ exists in any direction. If a point is Pareto critical for *any* step size from the point $\boldsymbol{x}$, then $\boldsymbol{x}$ is a *Pareto point*. The notion of a Pareto critical point is, therefore, similar to that of a local minimum, and that of a Pareto point is similar to a global minimum. In computational optimization, one is concerned with the problem of finding a local minimum since, except in special cases, it is the best that one can be guaranteed of finding without an exhaustive search. If the set of all Pareto critical points is P_c, and the set of Pareto points is P, then clearly $P \subset P_c$. In general, there could be an infinite number of Pareto points, but the best circuit design must necessarily occur at a Pareto point $\boldsymbol{x} \in P$.

In Fig. 4, the level curves of two objective functions are plotted in $\boldsymbol{R}^2 \cdot f_1$ is nonlinear and has a minimum at $\boldsymbol{x}^* \cdot f_2$ is linear and decreases as both x_1 and x_2 decrease. The Pareto critical set, P_c, is given by the dashed curve. At a few of the points, the unit normal to the level lines of f_1 and f_2, that is, the negative gradients of f_1 and f_2, is shown. From the figure, it can be seen that if the unit normals at point $\boldsymbol{x}$ are not equal and opposite, then the unit normals will have a common downhill direction allowing a simultaneous decrease in f_1 and f_2, and hence, $\boldsymbol{x}$ would not be a Pareto critical point. Therefore, a Pareto critical point is one where the gradients of f_1 and f_2 are opposite in direction, that is, $\lambda \nabla f_1 = -\nabla f_2$, where λ is some scale factor.

In higher dimensions, a Pareto critical point is characterized by the existence of a set of weights, $w_i > 0 \; \forall \; 1 \leq i \leq m$, such that

$$\sum_{i=1}^{m} w_i \boldsymbol{\nabla} f_i = \boldsymbol{0}. \tag{26}$$

Some of the common methods that are used for multicriterion optimization are discussed in the following sections.

Weighted-sum optimization. The multiple objectives, $f_1(\boldsymbol{x}), \ldots, f_m(\boldsymbol{x})$ are combined as

$$F(\boldsymbol{x}) = \sum_{i=1}^{m} w_i f_i(\boldsymbol{x}) \tag{27}$$

where $w_i > 0 \; \forall \; i = 1, \ldots, m$, and the function $F(\boldsymbol{x})$ is minimized.

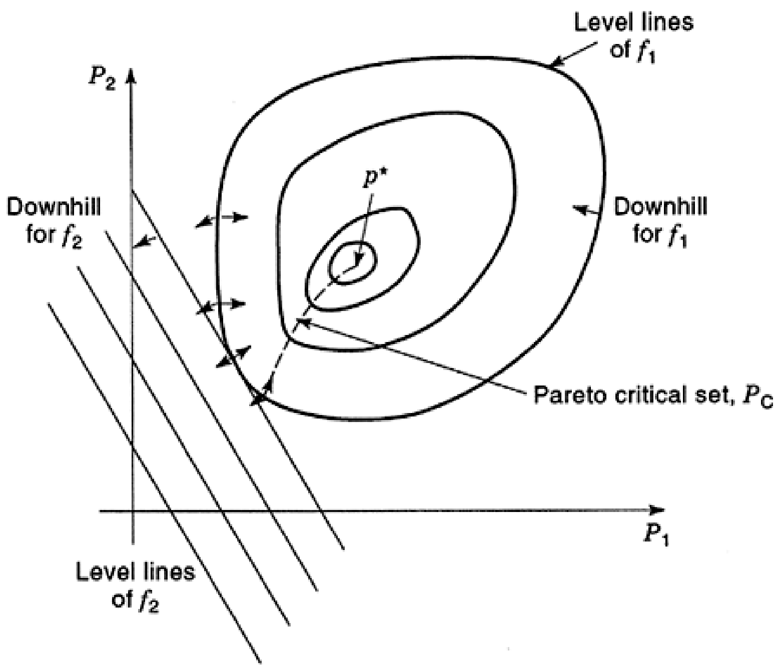

Fig. 4. Exact conflict at a Pareto critical point (©1981 IEEE) (7).

At any local minimum point of $F(\boldsymbol{x})$, the relation in Eq. (26) is seen to be valid, and hence, $\boldsymbol{x} \in P_c$. In general, $P \neq P_c$, but it can be shown that when each f_i is a convex function, then $P = P_c$; if so, it can also be shown that all Pareto points can be obtained by optimizing the function F in Eq. (27). However, for nonconvex functions, there are points $\boldsymbol{x} \in P$ that cannot be obtained by the weighted sum optimization since Eq. (26) is only a necessary condition for the minimum of F. A characterization of the Pareto points that can be obtained by this technique is provided in Ref. 7.

In practice, the w_i's must be chosen to reflect the magnitudes of the f_i's. For example, if one of the objectives is a voltage quantity the typical value of which is a few volts, and another is a capacitor value that is typically a few picofarads, the weight corresponding to the capacitor value would be roughly 10^{12} times that for the voltage, in order to ensure that each objective has a reasonable contribution to the objective function value. The designer may further weigh the relative importance of each objective in choosing the w_i's. This objective may be combined with additional constraints to give a formulation of the type in Eq. (1).

Minmax optimization. The following objective function is used for Eq. (1):

$$\text{Minimize } F(\boldsymbol{x}) = \max_{1 \le i \le m} w_i f_i(\boldsymbol{x}) \tag{28}$$

where the weights $w_i > 0$ are chosen as in the case of weighted sums optimization.

The above can equivalently be written as the following constrained optimization problem:

$$\begin{aligned} &\text{Minimize } r \\ &\text{subject to } w_i f_i(\boldsymbol{x}) \le r \end{aligned} \tag{29}$$

Minimizing the objective function described by Eq. (28) with different sets of w_i values can be used to obtain all Pareto points (7).

Since this method can, unlike the weighted-sum optimization method, be used to find *all* Pareto critical points, it would seem to be a more natural setting for obtaining Pareto points than the weighted-sum minimization. However, when the f_i's are convex, the weighted-sum approach is preferred since it is an unconstrained minimization and is computationally easier than a constrained optimization. It must be noted that when the f_i's are not all convex, the minmax objective function is nonconvex, and finding all local minima is a nontrivial process for any method.

Discrete Optimization

Simulated Annealing. The simulated annealing algorithm (15) is an approach that is very suitable for combinatorial optimization problems in which the number of possible solutions is very large, and most iterative improvement algorithms are liable to be stuck in a local minimum that is not globally optimal. In this section, we treat the simulated annealing algorithm as a procedure for combinatorial optimization. While extensions for continuous nonlinear optimization do exist, they are typically used to a lesser degree.

The basic operation during simulated annealing is a *move*, which is an alteration in the current solution, most often a minor perturbation. Like a greedy iterative algorithm, simulated annealing accepts a move to the perturbed solution if it resulted in a lower cost; unlike a greedy approach, however, simulated annealing may sometimes also accept a move if it has a larger cost than the current solution. This last property permits the algorithm to perform hill-climbing that helps it to exit from local minima in a quest for the global minimum.

The metaphor that is used here is the process of annealing a metal, where at high temperatures, the atoms may move freely and randomly in the metal. As the metal cools, the motion of the atoms becomes increasingly restricted and localized and hence less random. As the metal cools sufficiently slowly, the totality of these peregrinations lead to a state in which the atoms have been permitted to explore their freedoms and settle into the minimum energy state with the lowest cost.

In a similar manner, the simulated annealing approach performs a set of iterations in an outer loop that changes a parameter that is referred to as the *temperature*. Within this outer loop lies an inner loop in which a number of moves are made at any given temperature, and the temperature is gradually reduced from a high value to a low value according to a specified *cooling schedule*, which dictates how the temperature is changed from one iteration of this outer loop to the next. For high values of the temperature, almost all moves are accepted, regardless of whether they increase or decrease the cost of the solution. As the temperature becomes lower, cost-increasing moves are rejected with larger probabilities. The probability of acceptance is determined by the *Metropolis function*, defined as

$$M(\Delta\mathrm{Cost}, T) = e^{-\Delta\mathrm{Cost}/T}, \tag{30}$$

where ΔCost is the increase in cost due to the move. At any given temperature, if $\Delta\mathrm{Cost} \leq 0$, the move is accepted; if not, it is accepted with a probability of M (ΔCost, T), which clearly has the desired behavior. It is worth noting that $0 < M(\Delta\mathrm{Cost}, T) < 1$.

In practice, the acceptance of the move is determined by generating a random number under a uniform distribution; note that the probability that a uniformly distributed random variable is less than p, $0 \leq p \leq 1$ is given by p. Therefore, if the number is less than $M(\Delta\mathrm{Cost}, T)$, then the move is accepted.

The behavior of simulated annealing can be modeled using a Markov chain, where the next state after a move is dependent only on the current state and not on the history of how that state was attained. Using such models, it has been shown that simulated annealing will asymptotically find the optimal solution to a combinatorial problem. While such a proof is of limited comfort since an engineer can seldom wait until time reaches infinity, it is definitely true that the procedure, while slower than most other methods, has been successful in finding better solutions than several of its competing deterministic approaches for many problems. An interesting feature of simulated annealing is that it is independent of the initial solution since at a high temperature, almost all moves are accepted, and consequently the initial configuration is soon lost.

Genetic Algorithms. Genetic algorithms (16) are another set of nondeterministic algorithms that mimic the process of evolution to search for an optimal solution. Genetic algorithms begin with an arbitrary initial set of solutions, of varying costs, referred to as the *population*. Each individual in the population is characterized by a set of symbols, referred to as *genes*, and the set of genes that identify an individual are called *chromosomes*. In each iteration, or a *generation*, a segment of the population is altered through a set of random transformations, referred to as crossover, mutation, and inversion. New candidates are created by applying these transformations to one or more members of the population, with a *fitness function* being used to calibrate their ability to survive. In any generation, the fitness function for all members of the population is computed, and only the fittest survive.

Like simulated annealing, genetic algorithms also operate by permitting solutions that are worse than the best, with the difference that they maintain a number of solutions in the population instead of only one, and the manner in which moves are made is different. The likelihood that a solution would survive is dictated by the fitness of other competing solutions that are created in each generation.

The individual operations may be briefly described as follows. A *crossover* operation takes the genes of two parents and generates an offspring by combining the genes of each parent. A *mutation* operation, on the other hand, operates only on a single individual, and produces spontaneous random changes in an individual by altering a subset of these genes. Finally, an *inversion* operation takes a single chromosome and alters a randomly chosen segment of the chromosome by flipping it. Note that the inversion operation does not alter the set of genes associated with that individual solution but merely modifies the order. The significance of the crossover operation is similar to birth, which allows the chromosomes of two parents to be selectively combined. On the other hand, since crossover operations within a restricted population could result in inbreeding, the mutation operation plays the vital role of introducing new external sets of genes that are not to be found in the current population. Alternatively, mutation may reintroduce sets of genes that were rejected as being incompatible with some other genes, but that could provide good solutions when combined with others. The role of inversion is simply to permit the genes within a given solution to be permuted to help enlarge the space of possible solutions.

Application to Circuit Optimization Problems

Transistor sizing.

Problem Description. Circuit delays in integrated circuits often have to be reduced to obtain faster response times. A typical complementary metal oxide semiconductor (*CMOS*) digital integrated circuit consists of multiple stages of combinational logic blocks that lie between latches that are clocked by system clock signals. For such a circuit, delay reduction must ensure that valid signals are produced at each output latch of a combinational block, before any transition in the signal clocking the latch. In other words, the worst-case input-output delay of each combinational stage must be restricted to be below a certain specification.

Given the circuit topology, the delay of a combinational circuit can be controlled by varying the sizes of transistors in the circuit. Here, the size of a transistor is measured in terms of its channel width, since the channel lengths of MOS transistors in a digital circuit are generally uniform. In coarse terms, the circuit delay can usually be reduced by increasing the sizes of certain transistors in the circuit. Hence, making the circuit faster usually entails the penalty of increased circuit area. The area-delay tradeoff involved here is, in essence, the problem of transistor-size optimization.

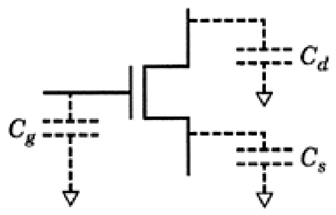

Fig. 5. An *RC* transistor model.

Three formal statements of the problem were proposed in Ref. 17:

$$\text{Minimize (area) subject to (delay)} \le T_{\text{spec}} \tag{31}$$

$$\text{Minimize (delay) subject to (area)} \le A_{\text{spec}} \tag{32}$$

$$\text{Minimize (area)} \times \text{(delay)}^c \tag{33}$$

where c is a constant. Of all of these, the first form is perhaps the most useful practical form, since a designer's objective is typically to meet a timing constraint dictated by a system clock.

Delay Modeling. We examine the delay modeling procedure used in the program TILOS (Timed Logic Synthesizer) at the transistor, gate, and circuit levels. Most existing transistor-sizing algorithms use minor variations on this theme.

A MOS transistor is modeled as a voltage-controlled switch with an on-resistance, R_{on}, between drain and source, and three grounded capacitances, C_d, C_s, and C_g, at the drain, source, and gate terminals, respectively, as shown in Fig. 5. The behaviors of the resistance and capacitances associated with a MOS transistor of channel width x are modeled as

$$R_{\text{on}} \propto 1/x \tag{34}$$

$$C_d, C_s, C_g \propto x \tag{35}$$

Other more accurate models that have good analytic properties are discussed in Ref. 9.

At the gate level, delays are calculated in the following manner. For each transistor in a pull-up or pull-down network of a complex CMOS gate, the largest resistive path from the transistor to the gate output is computed, as well as the largest resistive path from the transistor to a supply rail. Thus, for each transistor, the network is transformed into an *RC* line, and its Elmore time constant (18,19) is computed and is taken to be the gate delay. While finding the Elmore delay, the capacitances that lie between the switching transistor and the supply rail are assumed to be at the voltage level of the supply rail at the time of the switching transition and do not contribute to the Elmore delay.

Each R_i is inversely proportional to the corresponding transistor size x_i, and each C_i is some constant (for wire capacitance) plus a term proportional to the width of each transistor the gate, drain, or source of which is connected to node i. Thus, the delay can be written as a sum of terms formed by a product of resistance terms of the type $A/x_1 + A/x_2$ and capacitance terms of the type $Bx_2 + Cx_3 + D$, which yields a posynomial function of x_1, x_2, and x_3.

At the circuit level, the PERT technique (20) is used to find the circuit delay. A gate is said to be *ready for processing* when the signal arrival time information is available for all of its inputs. Initially, since signal arrival times are known only at the primary inputs, only those gates that are fed solely by primary inputs are ready for processing. These are placed in a queue and are scheduled for processing. In the iterative process, the gate at the head of the queue is scheduled for processing. Each processing step consists of

- Finding the latest arriving input to the gate, which triggers the output transition—this involves finding the maximum of all worst-case arrival times of inputs to the gate
- Adding the delay of the gate to the latest arriving input time, to obtain the worst-case transition time at the output
- Checking all of the gate that the current gate fans out to, to find out whether it is ready for processing—if so, the gate is added to the tail of the queue

The iterations end when the queue is empty. The *critical path*, defined as the path between an input and an output with the maximum delay, can be found by successively tracing back, beginning from the primary output with the latest transition time, and walking back along the latest arriving fan-in of the current gate, until a primary input is reached.

In the case of CMOS circuits, the rise and fall delay transitions are calculated separately. For inverting CMOS gates, the latest arriving input rise (fall) transition triggers off a fall (rise) transition at the output. This can easily be incorporated into the PERT method previously described, by maintaining two numbers, t_r and t_f, for each gate, corresponding to the worst-case rise (high transition) and fall (low transition) delays from a primary input. To obtain the value of t_f at an output, the largest value of t_r at an input node is added to the worst-case fall transition time of the gate; the computation of t_r is analogous. For noninverting gates, t_r and t_f are obtained by adding the rise (fall) transition time to the worst-case input rise (fall) transition time. Since each gate delay is a posynomial, and the circuit delay found by the PERT technique is a sum of gate delays, the circuit delay is also a posynomial function of the transistor sizes.

The Area Model. The exact area of a circuit cannot easily be represented as a function of transistor sizes. This is unfortunate, since a closed functional form facilitates the application of optimization techniques. As an approximation, the following formula is used by many transistor sizing algorithms, to estimate the active circuit area:

$$(\text{area}) = \sum_{i=1}^{n} x_i \qquad (36)$$

where x_i is the size of the ith transistor and n is the number of transistors in the circuit. In other words, the area is approximated as the sum of the sizes of transistors in the circuit, which, from the definition equation (10), is clearly a posynomial function of the x_i's.

The Sensitivity-based TILOS Algorithm. The algorithm that was implemented in TILOS (17,21) was the first used to recognize the fact that the area and delay can be represented as posynomial functions of the transistor sizes. The algorithm is heuristic and proceeds as follows.

An initial solution is assumed in which all transistors are at the minimum allowable size. In each iteration, the critical path for the circuit is first determined. Let N be the primary output node on the critical path. The algorithm then walks backward along the critical path, starting from N. Whenever an output node of a gate, Gate$_i$, is visited, TILOS examines the largest resistive path between V_{DD} (ground) and the output node if Gate$_i$'s t_r (t_f) causes the timing failure at N. This path contains a set of transistors connected between output and a supply node, including the transistor on the critical path. We classify the transistors as

- The *critical transistor*, that is, the transistor with a gate terminal on the critical path
- The *supporting transistors*, that is, transistors along the largest resistive path from the critical transistor to the power supply (V_{DD} or ground)
- The *blocking transistors*, that is, transistors along the highest resistance path from the critical transistor to the logic gate output

TILOS finds the sensitivity, which is the reduction in circuit delay per increment of transistor size, for each critical, blocking, and supporting transistor. Since the effect of changing a transistor size on the path delay is very localized, as it affects only the present gate, its fan-ins, and its fan-outs under the delay model discussed here, the sensitivities can be calculated very efficiently. The size of the transistor with the greatest sensitivity is increased by multiplying it by a constant, BUMPSIZE, a user-settable parameter that defaults to 1.5. The process above is repeated until all constraints are met, implying that a solution is found, or the minimum delay state has been passed, and any increase in transistor sizes would make it slower instead of faster, in which case TILOS cannot find a solution.

Since each iteration changes exactly one transistor size, the timing analysis method can employ incremental simulation techniques to update delay information from the previous iteration. This substantially reduces the amount of time spent in critical path detection.

Note that increasing the size of a transistor with negative sensitivity only means that the delay along the current critical path can be reduced by changing the size of this transistor, and does not necessarily mean that the circuit delay can be reduced; the circuit delay is the maximum of all path delays in the circuit, and a change in the size of this transistor could increase the delay along some other path, making a new path critical. This is the rationale behind increasing the size of the most sensitive transistor by only a small factor.

Transistor Sizing Using the Method of Feasible Directions. Shyu *et al.* (22) proposed a two-stage optimization approach to solve the transistor sizing problem. The delay estimation algorithm is identical to that used in TILOS. The algorithm can be summarized in the following pseudocode:

```
Use TILOS to size the entire circuit;
While (TRUE) {
    Select $G_1, \cdots, G_k$, the $k$ most critical paths,
    and $X = \{x_i\}$, the set of design parameters
    Solve the optimization problem
                minimize      $\sum_{x_i \in X} x_i$
                such that  $G_i(X) \leq T \; \forall i = 1, \ldots, k$
                      and  $x_i \geq$ minsize $\forall x_i \in X$.
    If all constraints are satisfied, exit
}
```

In the first stage, the TILOS heuristic is used to generate an initial solution. The heuristic finds a solution that satisfies the constraints, and only the sized-up transistors are used as design parameters. Although TILOS is not guaranteed to find an optimal solution, it can serve as an initial guess solution for an iterative technique. In the second stage of the optimization process, the problem is converted into a mathematical optimization problem, and is solved by a method of feasible directions (*MFD*) algorithm described earlier, using the feasible solution generated in the first stage as an initial guess.

To reduce the computation, a sequence of problems with a smaller number of design parameters is solved. At first, the transistors on the worst-delay paths (usually more than one) are selected as design parameters. If, with the selected transistors, the optimizer fails to meet the delay constraints and some new paths become the worst-delay paths, the algorithm augments the design parameters with the transistors on those paths and restarts the process. However, while this procedure reduces the run time of the algorithm, one faces the risk of finding a suboptimal solution since only a *subset* of the design parameters is used in each step.

The MFD optimization method proceeds by finding a search direction $\boldsymbol{d}$, a vector in the n-dimensional space of the design parameters, based on the gradients of the cost function and some of the constraint functions. Once the search direction has been computed, a step along this direction is taken, so that the decrease in the cost and constraint functions is large enough. The computation stops when the length of this step is sufficiently small.

Since this algorithm has the feature that once the feasible region (the set of transistor sizes for which all delay constraints are satisfied) is entered, all subsequent improvements will remain feasible, and the algorithm can be terminated at any time with a feasible solution.

For convergence, the MFD requires that the objective and constraint functions be continuously differentiable. However, since the circuit delay is defined as the maximum of all path delays, the delay constraint functions are usually not differentiable. To cope with the nondifferentiability of the constraint functions, a modification of the MFD is used that employs the concept of the *generalized gradient* (23). The idea is to use a convex combination of the gradients of the active or nearly active constraints near a discontinuity. For details of the scheme, the reader is referred to Ref. 22.

Lagrangian Multiplier Approaches. As can be seen from the approaches studied so far, the problem of transistor sizing can be formulated as a constrained nonlinear programming problem. Hence, the method of Lagrangian multipliers, described earlier, is applicable. Early approaches that used Lagrangian multipliers (24,25) rely on the user to provide critical path information, which may be impractical since critical paths are liable to change as sizing progresses. An alternative solution to transistor-size optimization using Lagrangian multipliers was presented by Marple. This technique uses a different area model and employs the idea of introducing intermediate variables to reduce the number of delay constraints from an exponential number to a number that is linear in the circuit size.

This technique begins with a prespecified layout and performs the optimization using an area model for that layout. While such an approach has the disadvantage that it may not result in the minimal area over *all* layouts, it still maintains the feature that the area and delay constraints are posynomials. Apart from the delay constraints, there also exist some area constraints, modeled by constraint graphs that are commonly used in layout compaction (26). These constraints maintain the minimum spacing between objects in the final layout, as specified by design rules.

The delay of the circuit is modeled by a delay graph $D(V,E)$, where V is the set of nodes (gates) in D, and E is the set of arcs (connections among gates) in D. This is the same graph on which the PERT analysis is to be carried out. Let m_i represent the worst-case delay at the output of gate i from the primary inputs. Then for each gate, the delay constraint is expressed as

$$m_i + d_j \le m_j \tag{37}$$

where gate $i \in$ fan-in (gate j) and d_j is the delay of gate j. Thus, the number of delay constraints is reduced from a number that could, in the worst case, be exponential in $|V|$, to one that is linear in $|E|$, using $|V|$ additional variables. These techniques are implemented in Refs. 27 and 28.

A more recent work (29) uses the idea of Lagrangian relaxation to solve the problem. The essential idea is to minimize, using the notation of the section entitled "Lagrange Multiplier Methods," the function $f(\boldsymbol{x}) + \lambda^T g(\boldsymbol{x})$ for a fixed value for the Lagrange multiplier vector λ. After this is done, the value of the λ is updated, and the procedure is continued until convergence. The results using this approach were found to be extremely fast.

Two-step Optimization. Since the number of variables in the transistor-sizing problem, which equals the number of transistors in a combinational segment, is typically too large for most optimization algorithms to handle efficiently, many algorithms choose a simpler route by performing the optimization in two steps. Examples of algorithms that use this idea to solve the transistor-sizing problem are iCOACH (30) and MOSIZ (31).

In the first step in MOSIZ, each gate is mapped onto an equivalent macromodeling primitive, such as an inverter. The transistor-sizing problem on this simplified circuit is then solved. Note that the number of variables is substantially reduced when each gate is replaced by a simple primitive with fewer transistors. The delay of each equivalent inverter, with the transistor sizes obtained above, is taken as the *timing budget* for the gate represented by that inverter, and the gate is optimized under the timing budget.

iCOACH uses macromodels for timing analysis of the circuit and has the capability of handling dynamic circuits. The optimizer employs a heuristic to estimate an *improvement factor* for each gate, which is related to the sensitivity of the gate. The improvement factor depends on the fan-in count, fan-out count, and the worst-case resistive path to the relevant supply rail. The improvement factor is then used to allocate a timing budget to each gate. In the second step, for each gate, a smaller transistor-sizing problem is solved, in which the area–delay product of the gate is minimized, subject to its delay being within its timing budget. The number of variables for each such problem equals the number of transistors within the gate, which is typically a small number. The optimization method used here is Rosenbrock's rotating coordinate scheme (32).

The two steps are repeated iteratively until the solution converges. While this technique has the obvious advantage of reducing the number of design parameters to be optimized, it suffers from the disadvantage that the solution may be nonoptimal. This stems from the simplifications introduced by the timing budget allocation; the timing budget allocated to each gate may not be the same as the delay of the gate for the optimal solution.

A more recent approach (33) performs a set of provably optimal iterations between the delay budgeting phase and the gate optimization phase. The method proceeds through two phases that are iteratively repeated: first, fixing the set of transistor sizes and finding a set of delay budgets, under a maximum delay perturbation, that would minimize the circuit area, and then finding the optimal sizes for each gate corresponding to those budgets. The method is found to be very fast in practice.

The Convex Programming-based Approach. The algorithm in iCONTRAST (34) solves the underlying optimization problem exactly. The objective of the algorithm is to solve the transistor-sizing problem in Eq. (28), where both the area and the delay are posynomial functions of the vector $\boldsymbol{x}$ of transistor sizes. The procedure described below may easily be extended to solve the formulations in Eqs. (29) and (30) as well; however, these formulations are not as useful to the designer. The variable transformation $(x_i) = (e^{z_i})$ maps the problem in Eq. (28) to

$$\begin{aligned}&\text{Minimize}\ \ \text{Area}\,(\boldsymbol{z}) = \sum_{i=1}^{n} e^{z_i}\\ &\text{subject to}\quad D(\boldsymbol{z}) \le T_{\text{spec}}\end{aligned} \tag{38}$$

The delay of a circuit is defined to be the maximum of the delays of all paths in the circuit. Hence, it can be formulated as the maximum of posynomial functions of $\boldsymbol{x}$. This is mapped by the above transformation onto a function $D(\boldsymbol{z})$ that is a maximum of convex functions; a maximum of convex functions is also a convex function. The area function is also a posynomial in $\boldsymbol{x}$, and is transformed into a convex function by the same mapping. Therefore, the optimization problem defined in Eq. (28) is mapped to a *convex programming* problem, that is, a problem of minimizing a convex function over a convex constraint set. Due to the unimodal property of convex functions over convex sets, any local minimum of Eq. (28) is also a global minimum. A convex programming method (35) is then used to find the unique global minimum of the optimization problem.

Concluding Remarks. The list of algorithms presented above are among the most prominent used for transistor sizing. For further details and a more complete survey, the reader is referred to Ref. 19. For a related problem, the gate-sizing problem for selecting optimal gate sizes from a standard cell library, the reader is referred to Refs. 36,37,38.

The reader must be cautioned here that the actual optimization problem in transistor sizing is not exactly a posynomial programming problem. The use of Elmore delay models (which are accurate within about 20%) to approximate the circuit delay, and the use of approximate area models allows the problem to be formulated as a convex program, and hence although one may solve this optimization problem exactly, one still must endure the inaccuracies of the modeling functions. In practice, in most cases, this is not a serious problem.

Design Centering. The design-centering problem is described in this section, and this discussion provides some insight into the formulation of the problem as a linear program and as a convex program.

Problem Description. While manufacturing a circuit, it is inevitable that process variations will cause design parameters, such as component values, to waver from their nominal values. As a result, the manufactured circuit may no longer meet some behavioral specifications, such as requirements on the delay, gain, and bandwidth, that it has been designed to satisfy. The procedure of design centering attempts to select the nominal values of design parameters so as to ensure that the behavior of the circuit remains within specifications, with the greatest probability. In other words, the aim of design centering is to ensure that the manufacturing yield is maximized.

The values of n design parameters may be ordered as an n-tuple that represents a point in $\boldsymbol{R}^n$. A point is *feasible* if the corresponding values for the design parameters satisfy the behavioral specifications on the circuit. The feasible region (or the region of acceptability), $R_f \subset \boldsymbol{R}^n$, is defined as the set of all design points for which the circuit satisfies all behavioral specifications.

The random variations in the values of the design parameters are modeled by a probability density function, $\Phi(\boldsymbol{z}) : \boldsymbol{R}^n \rightarrow [0,1]$, with a mean corresponding to the nominal value of the design parameters. The yield of the circuit Y as a function of the mean $\boldsymbol{x}$ is given by

$$Y(\boldsymbol{x}) = \int_{R_f} \Phi_{\boldsymbol{x}}(\boldsymbol{z}) d\boldsymbol{z} \tag{39}$$

The *design center* is the point $\boldsymbol{x}$ at which the yield, $Y(\mathbf{x})$, is maximized. There have traditionally been two approaches to solving this problem: one based on geometrical methods and another based on statistical sampling. In addition, several methods that hybridize these approaches also exist.

A common assumption made by geometrical design centering algorithms is that R_f is a convex bounded body. Geometrical algorithms recognize that the evaluation of the integral in Eq. (39) is computationally difficult and generally proceed as follows: the feasible region in the space of design parameters, that is, the region where the behavioral specifications are satisfied, is approximated by a known geometrical body, such as a polytope or an ellipsoid. The center of this body is then approximated and is taken to be the design center.

The Simplicial Approximation Method. The simplicial approximation method (39) is a method for approximating a feasible region by a polytope and finding its center. This method proceeds in the following steps:

(1) Determine a set of $m \geq n + 1$ points on the boundary of R_f.

(2) Find the convex hull (see the section entitled "Basic Definitions") of these points and use this polyhedron as the initial approximation to R_f. In the two-dimensional example in Fig. 6(a), the points 1, 2, and 3 are chosen in step (a), and their convex hull is the triangle with vertices 1, 2, and 3. Set $k = 0$.

(3) Inscribe the largest n-dimensional hypersphere in this approximating polyhedron and take its center as the first estimate of the design center. This process involves the solution of a linear program. In Fig. 6(a), this is the hypersphere C^0.

(4) Find the midpoint of the largest face of the polyhedron, that is, the face in which the largest (n–1)-dimensional hypersphere can be inscribed. In Fig. 6(a), the largest face is 2–3, the face in which the largest one-dimensional hypersphere can be inscribed.

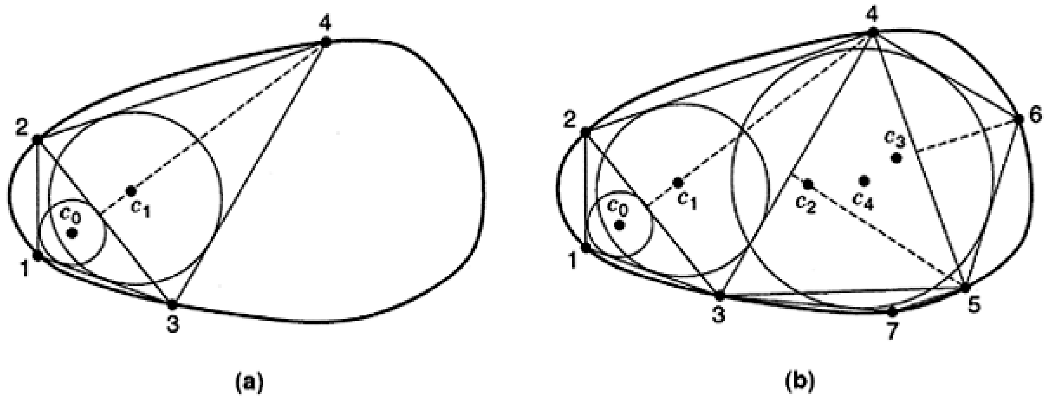

Fig. 6. The simplicial approximation method (©1977 IEEE) (39).

(5) Find a new boundary point on R_f by searching along the outward normal of the largest face found in step (d) extending from the midpoint of this face. This is carried out by performing a line search. In Fig. 6(a), point 4 is thus identified.

(6) Inflate the polyhedron by forming the convex hull of all previous points, plus the new point generated in step (e). This corresponds to the quadrilateral vertices 1, 2, 3, and 4 in Fig. 6(a).

(7) Find the center of the largest hypersphere inscribed in the new polyhedron found in step (f). This involves the solution of a linear program. Set $k = k + 1$, and go to step (d). In Fig. 6(a), this is the circle C^1.

Further iterations are shown in Fig. 6(b). The process is terminated when the sequence of radii of the inscribed hypersphere converges.

The procedure of inscribing the largest hypersphere in the polytope proceeds as follows. Given a polytope specified by Eq. (5), if the $\boldsymbol{a}_i$'s are chosen to be unit vectors, then the distance of a point $\boldsymbol{x}$ from each hyperplane of the polytope is given by $r = \boldsymbol{a}^{\mathrm{T}}{}_i\boldsymbol{x} - b_i$.

The center $\boldsymbol{x}$ and radius r of the largest hypersphere that can be inscribed within the polytope P are then given by the solution of the following linear program:

$$\begin{aligned} &\text{Minimize} && r \\ &\text{subject to} && a_i^{\mathrm{T}}\boldsymbol{x} - r \geq b_i \end{aligned} \tag{40}$$

Since the number of unknowns of this linear program is typically less than the number of constraints, it is more desirable to solve its dual (1). A similar technique can be used to inscribe the largest hypersphere in a face of the polytope; for details, see Ref. 39. For a generalization of the simplicial approximation method for the inscription of maximal norm bodies to handle joint probability density functions with (nearly) convex level contours, see Ref. 40.

If the above design-centering procedure is applied to a rectangular feasible region, the best possible results may not be obtained by inscribing a hypersphere. For elongated feasible regions, it is more appropriate to determine the design center by inscribing an ellipsoid rather than a hypersphere. The simplicial approximation handles this problem by scaling the axes so that the lower and upper bounds for each parameter differ by the same magnitude, and it is shown in Ref. 39 that one may inscribe the largest ellipsoid by inscribing the largest hypersphere in a transformed polytope. This procedure succeeds in factoring in reasonably the fact that feasible

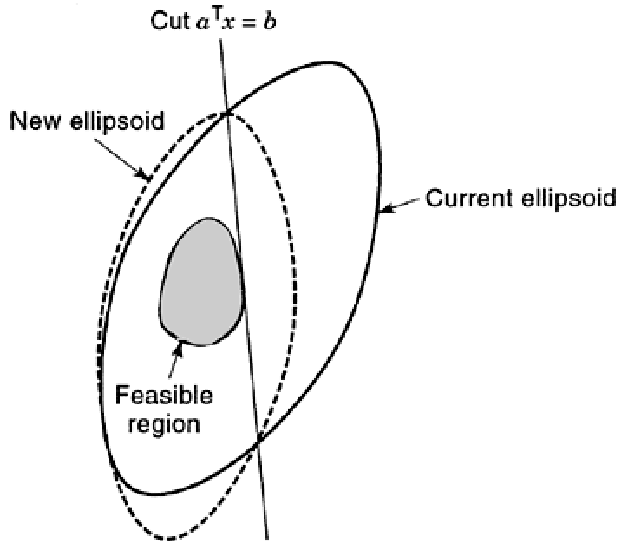

Fig. 7. The ellipsoidal method (©1991 IEEE) (41).

regions may be elongated; however, it considers only a limited set of ellipsoids that have their axes aligned with the coordinate axis, as candidates for inscription within the polytope.

The Ellipsoidal Method. This method, proposed in Ref. 41, is based on principles similar to those used by the Shor-Khachiyan ellipsoidal algorithm for linear programming (11). This algorithm attempts to approximate the feasible region by an ellipsoid, and takes the center of the approximating ellipsoid as the design center. It proceeds by generating a sequence of ellipsoids, each smaller than the last, until the procedure converges. Like other methods, this procedure assumes that an initial feasible point is provided by the designer. The steps involved in the procedure are as follows (see also Fig. 7):

(1) Begin with an ellipsoid E_0 that is large enough to contain the desired solution. Set $j = 0$.
(2) From the center of the current ellipsoid, choose a search direction, and perform a binary search to identify a boundary point along that direction. One convenient set of search directions are the parameter directions, searching along the ith, $i = 1, 2, \ldots, n$ in a cycle, and repeating the cycle, provided the current ellipsoid center is feasible. If not, a linear search is conducted along a line from the current center to the given feasible point.
(3) A supporting hyperplance (1) at the boundary point can be used to generate a smaller ellipsoid, E_{j+1}, that is guaranteed to contain the feasible region R_f, if R_f is convex. The equation of E_{j+1} is provided by an update procedure described in Ref. 41.
(4) Increment j, and go to step (a) unless the convergence criterion is met. The convergence criterion is triggered when the volume is reduced by less a given factor, ϵ. Upon convergence, the center of the ellipsoid is taken to be the design center.

Convexity-based Approaches. In the technique presented in Ref. 42, the feasible region, $R_f \subset \boldsymbol{R}^n$, is first approximated by a polytope described by Eq. (5). The algorithm begins with an initial feasible point, $\boldsymbol{z}_0 \in R_f$. An n-dimensional box, namely, $\{\boldsymbol{z} \in \boldsymbol{R}^n \mid z_{min} \leq z_i \leq z_{max}\}$, containing R_f is chosen as the initial polytope P_0. In each iteration, n orthogonal search directions, $\boldsymbol{d}_1, \boldsymbol{d}_2 \ldots \boldsymbol{d}_n$ are chosen (possible search directions include

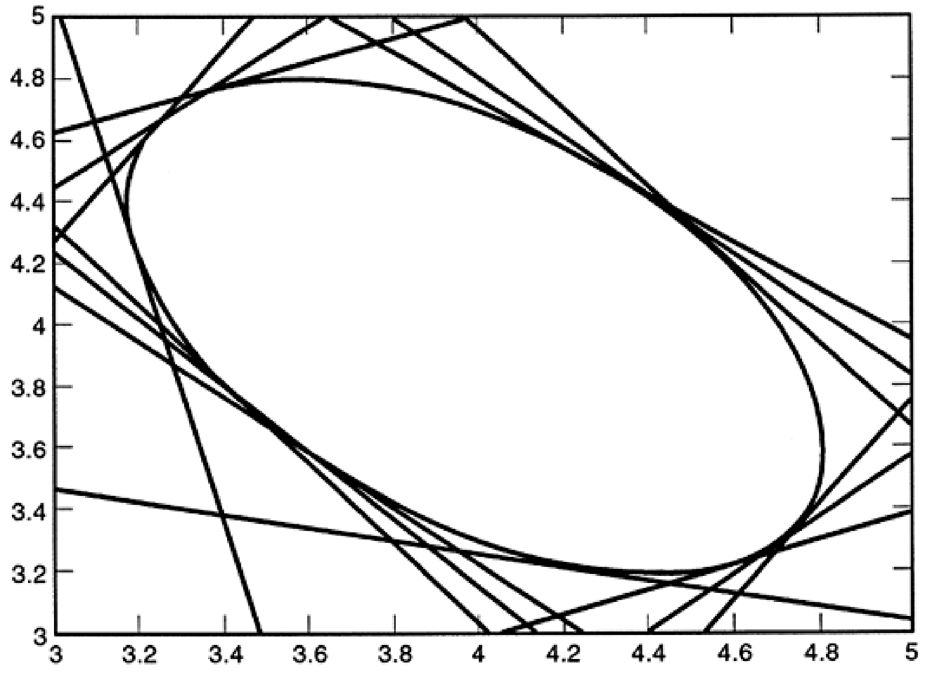

Fig. 8. Polytope approximation for the convexity-based methods (©1994 IEEE) (42).

the n coordinate directions). A binary search is conducted from $\boldsymbol{z}_0$ to identify a boundary point $\boldsymbol{z}_{bi}$ of R_f, for each direction $\boldsymbol{d}_i$. If $\boldsymbol{z}_{bi}$ is relatively deep in the interior of P, then the tangent plane to R_f at $\boldsymbol{z}_{bi}$ is added to the set of constraining hyperplanes in Eq. (5). A similar procedure is carried out along the direction $-\boldsymbol{d}_i$. Once all of the hyperplanes have been generated, the approximate center of the new polytope is calculated, using a method described in Ref. 42. Then $\boldsymbol{z}_0$ is reset to be this center, and the above process is repeated.

Therefore, unlike the simplicial approximation method that tries to expand the polytope outwards, this method starts with a large polytope and attempts to add constraints to shrink it inwards. The result of polytope approximation on an ellipsoidal feasible region is illustrated in Fig. 8.

When the probability density functions that represent variations in the design parameters are Gaussian in nature, the design-centering problem can be posed as a convex programming problem. The joint Gaussian probability density function of n independent random variables $\boldsymbol{z} = (z_1, \ldots, z_n)$ with mean $\boldsymbol{x} = (x_1, \ldots, x_n)$ and variance $\sigma = (\sigma_1 \ldots, \sigma_n)$ is given by

$$\Phi_{\boldsymbol{x}}(\boldsymbol{z}) = \frac{1}{(2\pi)^{n/2}\sigma_1\sigma_2\cdots\sigma_n} \exp\left[\sum_{i=0}^{i=n} -\frac{(z_i - x_i)^2}{2\sigma_i^2}\right] \tag{41}$$

This is easily seen to be a log-concave function of $\boldsymbol{x}$ and $\boldsymbol{z}$, that is, the logarithm of the function $\Phi_{\boldsymbol{x}}(\boldsymbol{z})$ is concave in $\boldsymbol{x}$ and $\boldsymbol{z}$. Also, note that arbitrary covariance matrices can be handled, since a symmetric matrix may be converted into a diagonal form by a simple linear (orthogonal) transformation. The design centering

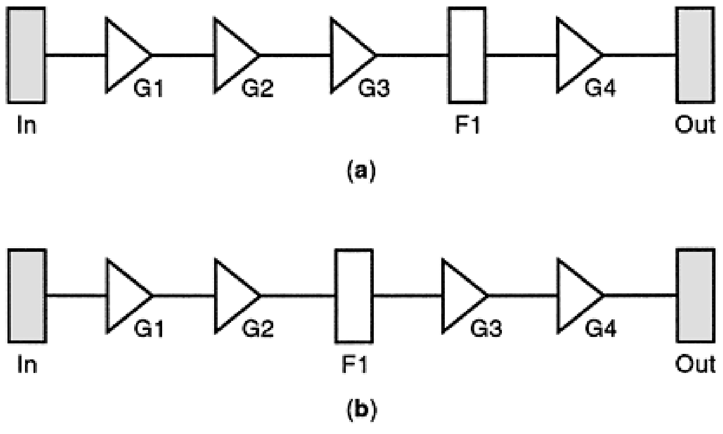

Fig. 9. Two different retimings for the same circuit (©1998 IEEE) (60).

problem is now formulated as

$$\begin{aligned}&\text{Maximize} \quad Y(\boldsymbol{x}) = \textstyle\int_{\mathcal{P}} \Phi_{\boldsymbol{x}}(\boldsymbol{z})d\boldsymbol{z} \\ &\text{such that} \quad \boldsymbol{x} \in \mathcal{P}\end{aligned} \tag{42}$$

where P is the polytope approximation to the feasible region R_f. It is a known fact that the integral of a log-concave function over a convex region is also a log-concave function. Thus, the yield function $Y(\boldsymbol{x})$ is log-concave, and the above problem reduces to a problem of maximizing a log-concave function over a convex set. Hence, this can be transformed into a convex programming problem. A convex programming algorithm (35) is then applied to solve the optimization problem.

Concluding Remarks. The list of algorithms above is by no means exhaustive but provides a general flavor for how optimization methods are used in geometrical design centering. The reader is referred to Refs. 43, 44,45,46 for further information about statistical design. In conclusion, it is appropriate to list a few drawbacks associated with geometrical methods: first, it is not always true that the feasible region will be convex; second, although some methods such as the simplicial approximation assume the center of the ellipsoid to be the design center, this is not accurate, as the precise design center can change depending on the probability distributions of the variables; and third, geometric methods suffer from the so-called "curse of dimensionality," whereby the computational complexity of these algorithms increases greatly with the number of variables.

Retiming.

Introduction. Retiming is a procedure that involves the relocation of flip-flops (FFs) across logic gates to allow the circuit to be operated under a faster clock. The chief idea is that while the FFs are relocated internally, the circuit must remain unchanged from an input-output perspective. Specifically, every path must have an identical latency (number of FF stages) in the original and in the retimed circuit. An example of a simple circuit under two different retimings is shown in Fig. 9.

The technique was first proposed by Leiserson, Rose, and Saxe (47,48), where the algorithmic basis of retiming circuits with edge-triggered FFs was described without specifically focusing on implementational aspects. Several papers have been published since then, such as Refs. 49,50,51,52,53,54,55,56, primarily dealing with algorithmic issues and extending the Leiserson–Rose–Saxe method to handle variations of the original Leiserson–Rose–Saxe problem.

Notation. A sequential circuit can be represented by a directed graph $G(V, E)$, where each vertex v corresponds to a gate, and a directed edge e_{uv} represents a connection from the output of gate u to the input of gate v, through zero or more registers. Each edge has a weight $w(e_{uv})$, which is the number of registers between the output of gate u and the input of gate v. Each vertex has a constant delay $d(v)$. A special vertex, the host vertex, is introduced in the graph, with edges from the host vertex to all primary inputs of the circuit and edges from all primary outputs to the host vertex.

A retiming is a labeling of the vertices $r : V \to Z$, where Z is the set of integers. The weight of an edge e_{uv} after retiming, denoted by $w_r(e_{uv})$ is given by

$$w_r(e_{uv}) = r(v) + w(e_{uv}) - r(u) \tag{43}$$

The retiming label $r(v)$ for a vertex v represents the number of registers moved from its output towards its inputs. One may define the weight of any path p originating at vertex u and terminating at vertex v (represented as $u \to v$), $w(p)$, as the sum of the weights on the edges on p, and its delay $d(p)$ as the sum of the weights of the vertices on p. A path with $w(p) = 0$ corresponds to a purely combinational path with no registers on it; therefore, the clock period can be calculated as

$$c = \max_{\forall\, p | w(p)=0} \{d(p)\} \tag{44}$$

Another important concept used in the Leiserson–Rose–Saxe approach is that of the W and D matrices that are defined as follows:

$$W(u, v) = \min_{\forall\, p:u;v} \{w(p)\} \tag{45}$$

$$D(u, v) = \max_{\forall\, p:u;v \text{ and } w(p)=W(u,v)} \{d(p)\} \tag{46}$$

The matrices are defined for all pairs of vertices (u, v) such that there exists a path $p : u \to v$ that does not include the host vertex. $W(u, v)$ denotes the minimum latency, in clock cycles, for the data flowing from u to v and $D(u, v)$ gives the maximum delay from u to v for the minimum latency.

The retiming problem as posed by Leiserson, Rose, and Saxe can be framed in the following two ways:

(1) The *minimum period* retiming problem, in which FFs are relocated to obtain a circuit with the minimum clock period, without any consideration to the area penalty due to an increase in the number of registers. Retiming for a specified clock period is a special case of this problem.
(2) The *constrained minimum area* retiming problem, in which FFs are relocated to achieve a given target clock period with the minimum register count. Constrained minimum area retiming is a much harder problem than minimum period retiming. Unconstrained minimum area retiming (i.e., retiming for minimum area without any regard for the final clock period) is a special case of minimum area retiming and can be solved efficiently because the time-consuming step of generating the period constraints (to be defined later) is not required.

Minimum period retiming. The minimum period retiming problem can be stated as follows:

$$\begin{aligned} &\text{Minimize } P \\ \text{subject to} \quad & r(u) - r(v) \le w(e_{uv}) \qquad \forall\, e_{uv} \in E \\ & r(u) - r(v) \le W(u, v) - 1 \quad \forall\, D(u, v) > P \end{aligned} \tag{47}$$

Note that this is not a linear program, since the second set of constraints depends on P. However, it can be solved efficiently by a binary search on the value of P. For a given value of P, there is no objective function, and the task is simply to find a feasible solution that satisfies all of the constraints, which are easily verified to be linear in the r variables.

The search could proceed as follows: starting with an initial interval, $[P_{\text{min}}, P_{\text{max}}]$, test to see whether the constraints are satisfiable for $P_{\text{mid}} = P_{\text{min}} + P_{\text{max}}/2$. If so, reduce the search interval to $[P_{\text{min}}, P_{\text{mid}}]$; otherwise, reduce it to $[P_{\text{mid}}, P_{\text{max}}]$. The search is completed when the interval is sufficiently small. The binary search approach is supported by the (provable) observation that if no such solution exists for a given value of P, then no solution exists for any smaller value of P.

Therefore, at each step of the binary search, we must find a feasible point that satisfies a set of linear constraints. While this may be solved as a linear program, we can observe that each constraint has a very specific form: it says that the difference between two variables should be no larger than a constant. Such a system is referred to as a system of *difference constraints* (57) and can be solved using graph traversal methods. Specifically, for each constraint of the type

$$r(u) - r(v) \le c_{uv} \tag{48}$$

one may build a constraint graph with one vertex for each r variable, with an edge from vertex v to vertex u with a weight of c. The solution to this system then corresponds to the shortest path in the graph from the host node, whose r variable is used as a reference and set to 0. For a general set of edge weights c_{uv} which may be larger or smaller than 0, the Bellman-Ford algorithm (57) may be applied to this graph to find the shortest paths. Note that there may be many feasible solutions to the set of inequalities, and the Bellman-Ford algorithm only identifies one of these.

Minimum area retiming. For minimum period retiming or for a retiming for any period, there are, in general, a number of solutions that correspond to different ways of using up the slacks in the constraint graph. The minimum area retiming problem for a target period P finds the solution that has the smallest number of FFs. It can be formulated as the following linear program:

$$\begin{aligned} \text{Minimize} \quad & \sum_{v \in V} \{[|\text{FI}(v)| - |\text{FO}(v)|]\, r(v)\} \\ \text{subject to} \quad & r(u) - r(v) \le w(e_{uv}) \qquad \forall\, e_{uv} \in E \\ & r(u) - r(v) \le W(u, v) - 1 \quad \forall\, D(u, v) > P \end{aligned} \tag{49}$$

where FI(v) and FO(v) represent the fan-in and fan-out sets of the gate v.

The significance of the objective function and the constraints is as follows (the reader is referred to Ref. 48 for details).

- The objective function represents the number of registers added to the retimed circuit in relation to the original circuit.
- The first constraint ensures that the weight e_{uv} of each edge (i.e., the number of registers between the output of gate u and the input of gate v) after retiming is nonnegative. We will refer to these constraints as *circuit constraints*.
- The second constraint ensures that after retiming, each path whose delay is larger than the clock period has at least one register on it. These constraints, being dependent on the clock period, are often referred to as *period constraints*.

It is easily verified, using the relations in Eqs. (13), (14), and (17), that the dual of this problem is an instance of a minimum-cost network flow problem.

Concluding Remarks. Some recent algorithms have presented fast and practical solutions to the retiming problem for large circuits. These include Refs. 58,59,60,61 and proceed primarily by pruning the number of constraints in the problem using insights available from a deeper study of the problem. Other approaches (62,63) provide a framework for incorporating long-path and short-path constraints together in a single formulation.

Placement. During the layout of a circuit, it is common for the preliminary layout of each module, such as a gate, to be designed independently. Subsequently, depending on the load to be driven, the module may be altered through sizing or other synthesis transformations. The placement problem involves the determination of locations for a set of modules that have a fixed size. The modules may be joined by a set of nets, each of which is connected to two or more modules. This connection point for a module is at a location that is fixed relative to a reference point on the module, such as the bottom left corner. The objective of the placement problem may be to minimize the packing area and to optimize the wire length or congestion or delay in the circuit.

While many algorithms for placement have been proposed (see, for example Refs. 64,65, and 26 for a survey), in this section, we will concentrate on two algorithms that use simulated annealing and genetic algorithms, respectively. The placement problem is amenable to being tackled using these approaches since it is inherently a combinatorial problem that has a very large search space, which is practically difficult to optimize over.

Placement by Simulated Annealing. One of the most successful placement algorithms for placement, which is often used as a benchmark against which to compare other placers, is based on the application of simulated annealing. The TimberWolf algorithm (66) is directed towards standard cell-based applications, where cells are arranged in rows so that all cells in a row have the same height but possibly different widths (26).

The skeleton of the algorithm progressing according to the simulated annealing procedure described earlier. At each temperature, a fixed number of moves is made. A move may consist of one of three actions: (i) moving a single cell to a new location in the same row or a different row, (ii) swapping two cells, and (iii) mirroring a cell while leaving its height and location unchanged. The first two types make up a majority of the moves, a relatively smaller number of the last type of move are introduced, and the ratio of the first type of move to the second is a parameter that is found to provide the best performance at values significantly larger than 1 (typically 3 to 8). The distance that a cell can move is bounded by a temperature-limited range limiter, which reduces this distance logarithmically with the temperature.

The cost function is a weighted sum of three components:

(1) The wire length cost, estimated as the sum, over all nets, of the semiperimeter of the bounding box of each net (with the option of weighting horizontal and vertical spans differently)
(2) A term that penalizes overlaps between cells, taken as the sum of the squares of overlaps between between all pairs of cells
(3) A term that penalizes nonuniformity of row lengths, taken as a sum, over all rows, of the absolute value of the difference between an expected row length and the actual row length

The cooling schedule for annealing is taken by multiplying the temperature T_i at the ith outer loop iteration by a factor $\alpha(T_i)$ to obtain the temperature for the $(i + 1)$th iteration. The value of α is typically set to a lower value (such as 0.8) for the upper and lower ranges of the temperature schedule and a higher value (such as 0.95) for the middle range.

Genetic Algorithms for Placement. The work in Ref. 67 presents an excellent example of the application of a genetic algorithm on the standard cell placement problem. The general framework is the same as described earlier. The genes correspond to ordered triples that define the cell identities and their x and y locations.

The crossover operation combines the genes of two parents to obtain an offspring chromosome. For a layout with n cells, the process of crossover can be performed by choosing the first $k < n$ cells from one chromosome, for some random value of k, and appending the remaining $n - k$ to the end of the string that identifies the offspring cell. Note that the ordering of the genes in the chromosome is important since two parents with the same genes but differently ordered chromosomes could lead to a different offspring even for the same value of k.

The mutation operation is a unary operation, and exchanges the locations of two cells. This corresponds to altering the genes in the chromosome so that their (x,y) locations are swapped. Finally, the inversion operation does not alter the placement correspond to a chromosome, but simply inverts the order of genes in a substring of the chromosome. Note that this permits a larger variation in the possible offspring that may be produced as a result of future crossover operations.

The fitness function is calculated as the inverse of the sum of the semiperimeters of all nets, with the horizontal and vertical directions being weighted differently. In each generation a set of crossover, mutation, and inversion operations are performed according to a specified rate. The crossover and mutation operations could potentially result in cell overlaps and are followed by a step that realigns the cells to remove these overlaps in the offspring. Following this step, the fitness function is evaluated for each offspring. Finally, a selection method is applied to cull the set of offspring to maintain the fittest of these in the population using either a deterministic or a random criterion, so as to maintain a constant population. The procedure continues for a certain number of generations, after which the fittest solution is selected.

Conclusion

The potential for applying optimization methods to circuit optimization is vast and has been shown to provide efficient optimal solutions. A good solution combines accurate models with an efficient optimizer, and compromising too much on either can lead to unusable solution. This article has presented a survey of such optimization algorithms and examples of their application to circuit optimization problems. Prominent conferences in this area, such as the ACM/IEEE Design Automation Conference and the IEEE/ACM International Conference on Computer-Aided Design and journals such as the IEEE Transactions on Computer-Aided Design, the IEEE Transactions on VLSI Systems, and the ACM Transactions on Design Automation of Electronic Systems publish the most recent advances in this area and are good references for further reading.

Acknowledgments

This effort was supported in part by the Semiconductor Research Corporation under Contract No. 99-TJ-692 and by the National Science Foundation under Grant No. CCR-9800992.

BIBLIOGRAPHY

1. D. G. Luenberger *Linear and Nonlinear Programming*, 2nd ed., Reading, MA: Addison-Wesley, 1984.
2. P. E. Gill W. Murray M. H. Wright *Numerical Linear Algebra and Optimization*, Reading, MA: Addison-Wesley, 1991, Vol. 1.

3. M. Berkelaar *LP_SOLVE User's Manual*, 1992.
4. B. A. Murtagh M. A. Saunders, MINOS 5.4 User's Guide, Tech. Rep. SOL 83-20R, Stanford University, Stanford, CA, 1995.
5. A. R. Conn N. I. M. Gould P. L. Toint *LANCELOT: A Fortran Package for Large-Scale Nonlinear Optimization*, Heidelberg: Springer, 1992.
6. W. H. Press S. A. Teukolsky W. T. Vetterling B. P. Flannery *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed., New York: Cambridge University Press, 1992.
7. R. K. Brayton G. D. Hachtel A. L. Sangiovanni-Vincentelli A survey of optimization techniques for integrated-circuit design, *Proc. IEEE*, **69**: 1334–1362, 1981.
8. J. Ecker Geometric programming: methods, computations and applications, *SIAM Rev.*, **22**: 338–362, 1980.
9. M. Ketkar K. Kasamsetty S. S. Sapatnekar Convex delay models for transistor sizing, in *Proc. ACM/IEEE Design Automation Conf.*, 2000.
10. N. Karmarkar A new polynomial-time algorithm for linear programming, *Combinatorica*, **4**: 373–395, 1984.
11. A. Schrijver *Theory of Linear and Integer Programming*, New York: Wiley, 1986.
12. M. S. Bazaraa J. J. Javis H. Sherali *Linear Programming and Network Flows*, New York: Wiley, 1977.
13. A. V. Fiacco G. P. McCormick *Nonlinear Programming: Sequential Unconstrained Minimization Techniques*, New York: Wiley, 1968.
14. G. Zoutendijk *Methods of Feasible Directions: A Study in Linear and Nonlinear Programming*, Amsterdam, The Netherlands: Elsevier, 1960.
15. S. Kirkpatrick C. Gelatt, Jr., M. Vecchi Optimization by simulated annealing, *Science*,**220**: 671–680, 1983.
16. D. E. Goldberg *Genetic Algorithms in Search, Optimization, and Machine Learning*, Reading, MA: Addison-Wesley, 1989.
17. J. Fishburn A. Dunlop TILOS: A posynomial programming approach to transistor sizing, in*Proc. IEEE Int. Conf. Comput.-Aided Design*, 1985, pp. 326–328.
18. J. Rubinstein P. Penfield M. A. Horowitz Signal delay in RC tree networks, *IEEE Trans. Comput.-Aided Design*,**CAD-2**: 202–211, 1983.
19. S. S. Sapatnekar S. M. Kang *Design Automation for Timing-Driven Layout Synthesis*, Norwell, MA: Kluwer Academic, 1993.
20. T. Kirkpatrick N. Clark PERT as an aid to logic design, *IBM J. Res. Devel.*,**10**: 135–141, 1966.
21. D. Hill D. Shugard J. Fishburn K. Keutzer *Algorithms and Techniques for VLSI Layout Synthesis*, Norwell, MA: Kluwer Academic, 1989.
22. J. Shyu J. P. Fishburn A. E. Dunlop A. L. Sangiovanni-Vincentelli Optimization-based transistor sizing, *IEEE J. Solid-State Circuits*,**23**: 400–409, 1988.
23. F. H. Clarke *Optimization and Nonsmooth Analysis*, New York: Wiley-Interscience, 1983.
24. M. A. Cirit Transistor sizing in CMOS circuits, in*Proc. 24th ACM/IEEE Design Automation Conf.*, June 1987, pp. 121–124.
25. K. S. Hedlund AESOP: A tool for automated transistor sizing, in Proc.*24th ACM/IEEE Design Automation Conf.*, June 1987, pp. 114–120.
26. N. Sherwani *Algorithms for VLSI Physical Design Automation*, Norwell, MA: Kluwer Academic, 1995.
27. D. Marple A. E. Gamal Area-delay optimization of programmable logic arrays, in*Proc. Adv. Res. VLSI*, Apr. 1986, pp. 171–194.
28. D. Marple A. E. Gamal Optimal selection of transistor sizes in digital VLSI circuits, in*Proc. Adv. Res. VLSI*, 1987, pp. 151–172.
29. C.-P. Chen C. C. N. Chu D. F. Wong Fast and exact simultaneous gate and wire sizing using lagrangian relaxation, in*Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, 1999, pp. 617–624.
30. H. Y. Chen S. M. Kang iCOACH: A circuit optimization aid for CMOS high-performance circuits, *Integration, VLSI J.*,**10**: 185–212, 1991.
31. Z. Dai K. Asada MOSIZ: A two-step transistor sizing algorithm based on optimal timing assignment method for multi-stage complex gates, in*Proc. 1989 Custom Integrated Circuits Conf.*, May 1989, pp. 17.3.1–17.3.4.
32. H. H. Rosenbrock An automatic method for finding the greatest or least value of a function, *Comput. J.*,**3**: 175–184, 1960.

33. V. Sundararajan S. S. Sapatnekar K. K. Parhi MINFLOTRANSIT: Min-cost flow based transistor sizing tool, in*Proc. ACM/IEEE Design Automation Conf.*, 2000.
34. S. S. Sapatnekar V. B. Rao P. M. Vaidya S. M. Kang An exact solution to the transistor sizing problem for CMOS circuits using convex optimization, *IEEE Trans. Comput.-Aided Des.***12**: 1621–1634, 1993.
35. P. M. Vaidya A new algorithm for minimizing convex functions over convex sets, *Proc. IEEE Found. Comput. Sci.*, **1989** (Oct.): 332–337.
36. W. Chuang S. S. Sapatnekar I. N. Hajj Delay and area optimization for discrete gate sizes under double-sided timing constraints, in*Proc. IEEE Custom Integrated Circuits Conf.*, 1993, pp. 9.4.1–9.4.4.
37. P. K. Chan Algorithms for library-specific sizing of combinational logic, in*Proc. ACM/IEEE Design Automation Conf.*, 1990, pp. 353–356.
38. S. Lin M. Marek-Sadowska E. S. Kuh Delay and area optimization in standard-cell design, in*Proc. ACM/IEEE Design Automation Conf.*, 1990, pp. 349–352.
39. S. W. Director G. D. Hachtel The simplicial approximation approach to design centering, *IEEE Trans. Circuits Syst.*,**CAS-24**: 363–372, 1977.
40. R. K. Brayton S. W. Director G. D. Hachtel Yield maximization and worst-case design with arbitrary statistical distributions, *IEEE Trans. Circuits Syst.*,**27**: 756–764, 1980.
41. H. L. Abdel-Malek A.-K. S. O. Hassan The ellipsoidal technique for design centering and region approximation, *IEEE Trans. Comput.-Aided Des.*,**10**: 1006–1014, 1991.
42. S. S. Sapatnekar P. M. Vaidya S. M. Kang Convexity-based algorithms for design centering, in*Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, 1993, pp. 206–209.
43. S. W. Director P. Feldmann K. Krishna “Statistical integrated circuit design”, *IEEE J. Solid-State Circuits*,**28**: 193–202, 1993.
44. M. D. Meehan J. Purviance *Yield and Reliability in Microwave Circuit and System Design*, Boston, MA: Artech House, 1993.
45. S. W. Director W. Maly A. J. Strojwas *VLSI Design for Manufacturing: Yield Enhancement*, Boston, MA: Kluwer Academic, 1990.
46. R. Spence R. S. Soin *Tolerance Design of Integrated Circuits*, Reading, MA: Addison-Wesley, 1988.
47. C. Leiserson F. Rose J. B. Saxe Optimizing synchronous circuitry by retiming, in*Proc. of the 3rd Caltech Conf. on VLSI*, 1983, pp. 87–116.
48. C. E. Leiserson J. B. Saxe Retiming synchronous circuitry, *Algorithmica*,**6**: 5–35, 1991.
49. N. Shenoy R. K. Brayton A. Sangiovanni-Vincentelli Retiming of circuits with single phase transparent latches, in*Proc. IEEE Int. Conf. Comput. Design*, 1991, pp. 86–89.
50. A. Ishii C. E. Leiserson M. C. Papaefthymiou Optimizing two-phase, level-clocked circuitry, in*Adv. Res. VLSI Parallel Systems: Proc. 1992 Brown/MIT Conf.*, 1992, pp. 246–264.
51. H.-G. Martin Retiming by combination of relocation and clock delay adjustment, in*Eur. Design Automation Conf.*, 1993, pp. 384–389.
52. T. Soyata E. G. Friedman J. H. Mulligan, Jr., Integration of clock skew and register delays into a retiming algorithm, in*Proc. IEEE Int. Symp. Circuits Syst.*, 1993, pp. 1483–1486.
53. A. T. Ishii Retiming gated-clocks and precharged circuit structures, in*Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, 1993, pp. 300–307.
54. B. Lockyear C. Ebeling Optimal retiming of level-clocked circuits using symmetric clock schedules, *IEEE Trans. Comput.-Aided Des.*, 1097–1109, 1994.
55. T. Soyata E. G. Friedman Retiming with non-zero clock skew, variable register and interconnect delay, in*Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, 1994, pp. 234–241.
56. K. N. Lalgudi M. Papaefthymiou DeLaY: An efficient tool for retiming with realistic delay modeling, in*Proc. ACM/IEEE Design Automation Conf.*, 1995, pp. 304–309.
57. T. H. Cormen C. E. Leiserson R. L. Rivest *Introduction to Algorithms*, New York: McGraw-Hill, 1990.
58. N. Shenoy R. Rudell Efficient implementation of retiming, in*Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, 1994, pp. 226–233.
59. R. B. Deokar S. S. Sapatnekar A fresh look at retiming via clock skew optimization, in*Proc. ACM/IEEE Design Automation Conf.*, 1995, pp. 310–315.
60. N. Maheshwari S. S. Sapatnekar Efficient retiming of large circuits, *IEEE Trans. VLSI Syst.***6**: 74–83, 1998.

61. N. Maheshwari S. S. Sapatnekar Optimizing large multiphase level-clocked circuits, *IEEE Trans. Comput.-Aided Des.*,**18**: 1249–1264, 1999.
62. M. Papaefthymiou Asymptotically efficient retiming under setup and hold constraints, in*Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, 1998, pp. 288–295.
63. V. Sundararajan S. S. Sapatnekar K. K. Parhi Marsh: Minimum area retiming with setup and hold constraints, in*Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, 1999, pp. 2–6.
64. M. Sarrafzadeh C. K. Wong *An Introduction to VLSI Physical Design*, New York: McGraw-Hill, 1996.
65. S. M. Sait H. Youssef *VLSI Physical Design Automation: Theory and Practice*, New York: IEEE, 1995.
66. C. Sechen A. L. Sangiovanni-Vincentelli TimberWolf3.2: A new standard cell placement and global routing package, in*Proc. ACM/IEEE Design Automation Conf.*, 1986, pp. 432–439.
67. K. Shahookar P. Mazumder A genetic approach to standard cell placement using meta-genetic parameter optimizations, *IEEE Trans. Comput.-Aided Des.*,**9**: 500–511, 1990.

SACHIN S. SAPATNEKAR
University of Minnesota