INVITED PAPER

# Technology-Dependent Logic Optimization

*The survey paper presents the state-of-the-art algorithms for technology-dependent optimizations, along with a comparison of their relative power to optimize the implementations of logic circuits.*

By Rajeev Murgai

**ABSTRACT** | In modern design flows, technology-dependent logic optimizations consist of technology mapping and transformations applied after mapping. Advances have been made in technology mapping in the last decade. However, it is inherently an intractable problem and state-of-the-art algorithms produce suboptimal netlists in terms of area and timing. Also, during the design flow, when physical information becomes available, wire loads and net delays change the timing values at the gates. That presents opportunities for further timing and area optimization. This is accomplished by transforms such as gate sizing, gate replication, buffer optimization, restructuring and remapping, and pin permutation. In this paper, we survey algorithms for technology-dependent optimizations, along with a comparison of their relative power to optimize the netlist.

**KEYWORDS** | Buffer optimization; gate replication; gate sizing; logic synthesis; technology mapping

## I. MOTIVATION

Modern flow for a design typically starts from a high-level description of the design, also called the register–transfer level (RTL) design (for instance, written in Verilog, VHDL, or System Verilog), design constraints (such as the target clock cycle), and technology libraries, as shown in Fig. 1. RTL or datapath synthesis transforms this description into a control-data flow graph, in which the datapath operators are implemented with primitive gates. Logic synthesis takes the control-data flow graph and produces a gate-level netlist with minimum cost. The cost could be the netlist area, delay, or area subject to delay constraints.

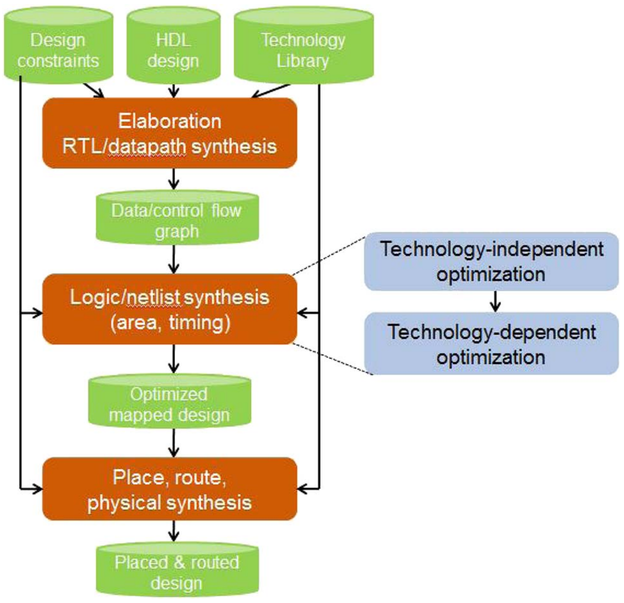
Design constraints
HDL design
Technology Library
Elaboration RTL/datapath synthesis
Data/control flow graph
Logic/netlist synthesis (area, timing)
Technology-independent optimization
Technology-dependent optimization
Optimized mapped design
Place, route, physical synthesis
Placed & routed design

**Fig. 1.** *Design flow.*

Physical design involves floorplanning, macro and cell placement, clock tree synthesis, and routing and physical synthesis, and generates a placed and routed design.

Logic synthesis is usually separated into two stages: technology-independent optimization followed by technology-dependent optimization (Fig. 1). Technology-independent optimization uses an abstract cost for design area and/or delay to derive an optimized netlist consisting of generic gates. Technology-dependent optimization takes this optimized netlist, uses technology information such as library cells (their area and delay) and design rules, and derives a netlist composed of library gates satisfying the required area and delay objectives. This paper presents an overview of the latter problem.

Manuscript received October 16, 2014; revised June 13, 2015 and August 4, 2015; accepted August 28, 2015. Date of current version October 26, 2015.
The author is with Synopsys India Private Ltd, Noida, U.P. 201301, India (e-mail: rajeev.murgai@synopsys.com).

Digital Object Identifier: 10.1109/JPROC.2015.2484299

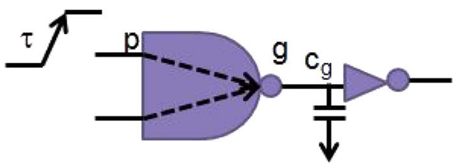

**Fig. 2.** ***Gate delay modeling.***

In modern design flows, technology-dependent logic optimizations consist of technology mapping and postmapping transformations. Technology mapping is inherently a difficult problem. The current mapping algorithms cannot provide optimum solutions for minimum delay or area for industrial circuits in the presence of large gate libraries, complex design constraints, realistic and accurate delay models, and in the absence of the interconnect load and delay information. Thus, there is a need and scope for applying postmapping logic transformations and even integrating them with layout steps. In this paper, we survey several such transformations, namely gate resizing, fanout optimization by buffering, gate replication, simple gate decomposition/collapsing, resynthesis & remapping, and pin permutation.

The paper is organized as follows. Section II gives an overview of the gate delay models and timing analysis. Section III provides an overview of the technology mapping problem for area and timing. Section IV gives details of postmapping optimizations, such as gate resizing, fanout optimization, gate replication or cloning, resynthesis and remapping, and pin permutation. It also presents scope and relative impact of each optimization. In Section V, we present a brief overview of logic resynthesis techniques to fix congestion.

## II. PRELIMINARIES

### A. Gate Delay Models

We describe three widely used gate delay models, all of which use linear or bilinear equations to compute the pin-to-pin delay through a gate.[1]

Given a single-output gate $g$, let $\delta(p,g)$ denote the delay from an input pin $p$ of $g$ to the output pin of $g$; this is also known as the arc delay. We will use $g$ to denote the output pin of $g$. The load $c_g$ refers to the cumulative capacitance seen at the output pin of $g$ (Fig. 2). It is the sum of the input pin capacitances $\gamma_p$ of all the fanout pins $p$ of $g$ and the loads of the wires connected to the output pin of $g$. The drive capacity of $g$, denoted $\lambda_g$, is the maximum capacitance $g$ can drive without compromising the signal integrity or the delay specifications.

The simplest delay model is the load-independent delay model (LIDM), in which $\delta(p,g)$ is independent of the load capacitance $c_g$ and is given by the intrinsic delay $\alpha(p,g)$

$$\delta(p,g) = \alpha(p,g). \tag{1}$$

In the more realistic load-dependent delay model (LDDM), the delay $\delta(p,g)$ depends linearly on both the intrinsic delay as well as the output load $c_g$

$$\delta(p,g) = \alpha(p,g) + \beta(p,g)\ c_g \tag{2}$$

where $\beta(p,g)$ is the load coefficient of the arc from input pin $p$ to $g$.

Finally, in the input-slew-dependent delay model (ISDDM), $\delta(p,g)$ depends not only on the output load $c_g$, but also on the slew (transition time) $\tau$ at the input pin $p$. The slew $\tau$, in turn, depends on the gate $h$ that is driving the pin $p$ as well as on the total capacitive load seen by $h$:[2]

$$\begin{aligned}\tau(h) &= a(h) + b(h)c_h\\ \delta(p,g) &= \alpha(p,g) + \beta(p,g)\ c_g\\ &\qquad + \big(\kappa(p,g) + \nu(p,g)\ c_g\big)\ \tau(h).\end{aligned}$$

The gate-dependent delay coefficients $\alpha$, $\beta$, $a$, $b$, $\kappa$, and $\nu$ are determined by first gathering delay and output slew values for a range of input slew and output load values (e.g., using a circuit simulator such as SPICE) and then fitting the aforementioned linear and bilinear forms on this data (e.g., by the method of least squares).

ISDDM is a popular delay model used in industrial libraries. However, most of the research in timing optimization has been done assuming LIDM and LDDM.

### B. Timing Analysis

Using any of the delay models, the delay on a path from a circuit primary input (PI) or a latch output to a circuit primary output (PO) or a latch input is the sum of the pin-to-pin delays through the gates on the path plus the delays of the wires on the path. The circuit delay is the maximum of all such path delays in the circuit. Given the arrival times $A$ at the primary inputs, a forward delay trace through the circuit computes signal arrival times at every pin of every gate and at the primary outputs. Given the required times $R$ at the primary outputs, a backward delay trace computes required times of signals at every pin of every gate and at the primary inputs. The slack of a pin/pad (a pad is a PI or a PO) is the difference between its required time and arrival

[1]Although popular, lookup-table-based delay models are not considered here.

[2]In more accurate models, the slew $\tau(h)$ at the output of the gate $h$ is also a function of the slew at the inputs of $h$, in addition to the output load $c_h$.

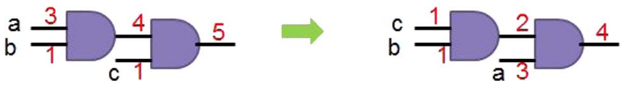

**Fig. 3.** *Delay-reduction rule in rule-based mappers: move late-arriving signal a closer to the root.*

time. A pin/pad is critical if its slack is negative. The primary goal in timing optimization is to make the circuit meet the required time constraints at POs and latch inputs, which is identical to making all the pins and pads noncritical. It is useful to have the concept of $\epsilon$-criticality. A pin/pad is $\epsilon$-critical if its slack lies within $\epsilon$ of the minimum slack in the circuit. Several timing optimization algorithms proceed by identifying and optimizing $\epsilon$-critical pins.

## III. TECHNOLOGY MAPPING

Technology mapping is the process of implementing a netlist with gates from a technology library. It is usually applied on a netlist that has been optimized with technology-independent optimization. The goal of technology mapping could be to minimize the area of the mapped netlist, minimize the delay, or minimize the area subject to delay constraint.

### A. Rule-Based Mappers

The first technology mappers were rule based, such as LSS [15], SOCRATES [4], and LORES [25]. Rules encapsulated circuit transformations that helped improve the circuit quality, usually locally. Most of these rules were based on a technology library and were handcrafted for a particular technology and design style. A rule was expressed as a pair: (target graph, replacement graph). It was applied by identifying a portion of the circuit which was isomorphic to the target graph and replacing it with the replacement graph. For instance, one delay-reduction rule is to place a late-arriving signal closer to the root of a symmetric tree, as shown in Fig. 3. Here, the late-arriving signal $a$ is moved closer to the root of the AND tree. Similarly, replace a NAND–NOT–NOR combination with AND–OR–INVERT gate.

In rule-based systems such as LSS, SOCRATES, and LORES, the circuit quality improved considerably by iteratively applying the rules. Although the rule-based approach provides a flexible and easy-to-develop way of applying technology-dependent transformations for optimizing different cost functions, it suffers from difficulties such as modifying the rule base when a new gate is added to the library, migrating to a new library, etc. Today the algorithmic optimization techniques such as resynthesis, remapping, fanout optimization, gate replication, etc., have become sophisticated and powerful, almost obviating the rule-based paradigm.

### B. Algorithmic Technology Mapping for Minimum Area

The first algorithmic breakthrough came in the form of a dynamic programming algorithm [27]. Keutzer [27] transformed the problem of technology mapping as a directed acyclic graph (DAG) covering problem. The optimized circuit/netlist is converted into a DAG, where each vertex is restricted to a base gate—for instance, a two-input NAND gate, or an inverter. This DAG is called a subject graph. The decomposition into base gates is called technology decomposition. The logic function for each library gate is likewise represented using the base functions. This generates pattern graphs. There may be more than one way to represent the gate function and so more than one pattern graph may result from a gate. A cover of the subject graph is a collection of pattern graphs such that every node of the subject graph is contained in one (or more) of the pattern graphs. The cover is further constrained so that each primary output is an output of some pattern graph (output constraint), and each input required by a pattern graph is either a primary input or an output of some other pattern graph (implication constraint). For minimum area, the cost of a cover is the sum of the area costs of the gates in the cover. The technology mapping problem may be viewed as the optimization problem of finding a minimum cost cover of the subject graph by choosing from the collection of pattern graphs for all gates in the library. This problem, known in the literature as the binate covering problem, is hard—in fact NP-hard, though efficient heuristics exist.

A commonly used heuristic divides the subject graph into trees and covers the trees optimally by tree patterns in polynomial time using a dynamic programming approach. This approach works as follows. Consider a tree $T$ of the subject graph that needs to be mapped for minimum area cost. The nodes of $T$ are visited (and mapped) from inputs (or leaves) of $T$ to its root. This is called the forward phase or match generation. When a node $v$ is visited, all the nodes $u$ in its transitive fanin in $T$ have already been visited and mapped optimally [so their implementation area costs $A(u)$ are known]. To map $v$, all possible pattern graphs that can be rooted at $v$ are determined. These patterns match with some subtree rooted at $v$. For each such match $M$, where $M$ corresponds to the library gate $g$, the cumulative area cost $A(v,M) = a(g) + \sum_i A(i)$, where $a(g)$ is the area cost of gate $g$ and $i$s are inputs to the match $M$. If a match input is a leaf node of $T$, its cost is 0. The match $M^*$ that results in the minimum total cost is stored at $v$. In other words

$$\begin{aligned} M^*(v) &= \arg\min_M \{A(v,M)\} \\ A(v) &= A(v,M^*). \end{aligned}$$

Once the root $r$ of $T$ has been visited, the optimum mapping for $T$ is generated as follows. The optimum match

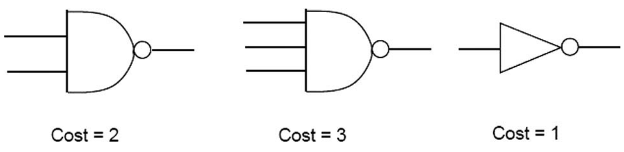

**Fig. 4.** ***Example cell library.***

at $r$, $M^*(r)$, is selected. Then, for each input $v$ of $M^*(r)$, $M^*(v)$ is selected. This process is repeated until the tree inputs are reached. This phase or step is called the backward phase or the covering step. DAGON [27] and misII technology mapper [17] are based on this approach, which we illustrate in the following example.

*Example 3.1:* Fig. 4 shows a simple cell (gate) library with three gates: a two-input NAND gate with area cost of two units, a three-input NAND gate with area cost of three units, and an inverter with area cost of one unit. The pattern graphs for these gates using two-input NAND gates and inverter are shown in Fig. 5. Note that the three-input NAND gate has a single pattern. Assume our netlist consists of a single four-input NAND gate, as shown in Fig. 6. We are interested in finding the minimum cost mapping of this netlist. First, we derive its subject graph in terms of two-input NAND gates and inverters. Many such subject graphs are possible, and we choose the one shown in Fig. 6. Note that it is a tree. We wish to find a minimum cost cover of this subject graph with the pattern graphs of Fig. 5. Fig. 7 shows two covers, (A) and (B), of the subject graph. The chosen patterns are shown as dotted rectangles. Note that (A) has a cost of eight units: it uses three two-input NAND gates and two inverters, whereas (B) has a cost of six units: it uses a two-input NAND gate, a three-input NAND gate, and an inverter, and is the best possible cover. To see that this indeed is the best cover, consider the root two-input NAND gate of the subject graph. Two patterns can be rooted at it.

1) A two-input NAND gate: The best cover of the subject graph in this case is this two-input NAND gate, along with the lowest cost covers of each of the subtrees rooted at the two inputs of the NAND gate. Considering all possible patterns rooted at these inputs recursively leads to the cover (A), which has a cost of eight units.
2) A three-input NAND gate: The best cover of the subject graph then corresponds to this three-input NAND gate, along with the best possible covers of each of the subtrees rooted at the three inputs of this NAND gate. Two of the inputs are primary inputs. So we recursively carry out the algorithm on the third NAND gate input. This finally leads to the cover (B) with a cost of six units.

The cover with the minimum cost is picked, i.e., (B). This, in brief, is how the dynamic programming algorithm works on trees.

The only requirement imposed on the library is that it be complete, i.e., an arbitrary logic function should be realizable in terms of the gates in this set. Although two-input NAND gates and inverters form a complete set, it is desirable to put more gates in the library in order to obtain high-quality results. In fact, modern day libraries routinely contain hundreds of gates. Most of the gates can be represented in terms of the base gates in several different ways. The set of pattern graphs for the entire library can become very large, leading to huge mapper runtimes. To reduce the complexity, library gates are partitioned into NPN equivalence classes. Two gates are NPN equivalent if one can be obtained from the other by renaming the inputs and inverting any inputs and/or outputs. Only one representative gate in each equivalence class is then represented in terms of pattern graphs. During match generation, matches are generated in the subject graph ignoring

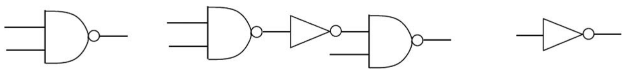

**Fig. 5.** ***Pattern graphs for the gates in the cell library.***

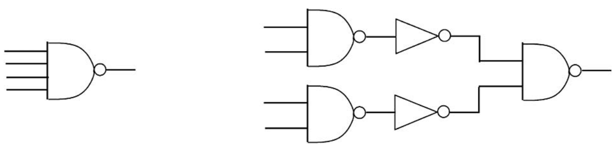

**Fig. 6.** ***Four-input NAND gate and one of its subject graphs.***

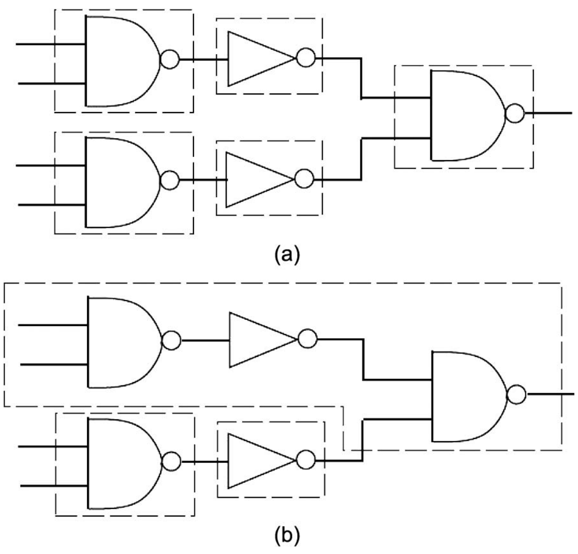

**Fig. 7.** ***Technology mapping: two covers: (a) cost = 8; and (b) cost = 6.***

inversions at the inputs and output of the match. Then, the ignored inversions are accounted for by looking for gates $g$ in the matched class that have the same inversions with respect to the representative gate of the class.

*2) Boolean Matching:* Keutzer's dynamic programming formulation solved the technology mapping problem by covering the subject graph with pattern graphs. This approach can be termed structural, since it requires representing each library gate as a tree of base gates. However, gates such as XORs and multiplexors do not have a tree decomposition. To cover these cases, Boolean matching techniques have been proposed, such as [7], [9], [12], and [16]. In Boolean matching, a library gate $g$ is represented by its Boolean function $f(g)$ in terms of either its truth table or its reduced ordered binary decision diagram (ROBDD). Similarly, the function $f(s)$ of the subgraph $s$ (of the subject graph) being matched is also likewise represented. To check if the library gate $g$ implements the function of the subgraph $s$, we check for equivalence of functions $f(g)$ and $f(s)$. This check is trivial for ROBDDs, assuming input correspondence (for instance, identical input names) between $f(g)$ and $f(s)$ is known. In general, $f(g)$ and $f(s)$ are in terms of different input variables. A correspondence between the inputs needs to be derived as part of establishing $f(g) = f(s)$. In the worst case, all $n!$ permutations of the inputs need to be tried for an exact match, where $n$ is the number of inputs. If $f(g)$ and $f(s)$ are not equal under any input permutation, the matching fails. For truth tables, matching can be made efficient by using NPN classes and encoding the truth table as an integer [24].

*3) Handling Multiple Decompositions:* The mapping quality depends on how the given netlist is decomposed into base gates, since there could be multiple ways in which a complex gate is decomposed in terms of two-input NAND gates and inverters. For a long time, it was an open problem as to how to construct an optimum mapped netlist over all possible node decompositions. Lehman *et al.* [31] solved this problem by introducing the concept of choice nodes in the subject graph. Choice nodes allowed multiple decompositions of a complex gate to be represented in the network at the same time. Special nodes, called choice nodes, were introduced to select the best decomposition on the fly during mapping. We illustrate this with the help of an example. Consider a network consisting of a single function $f = ab + ac$. It has two decompositions:

a) $x = ab$; $y = ac$; $f = x + y$;
b) $z = b + c$; $f = az$.

These are shown in Fig. 8(a) and (b), respectively. Both decompositions are represented in the subject graph of Fig. 8(c). A choice node is introduced, which is fed by the root gates of the two decompositions. During the forward phase of the mapper, when the choice node (which has a special identifier) is visited, the mapper selects the root gate and hence the decomposition that has smaller cost.

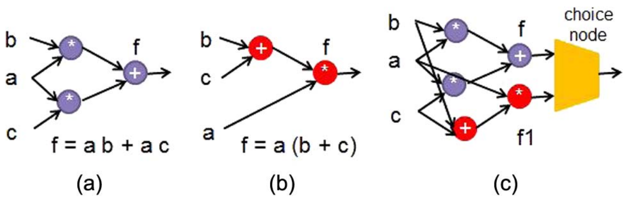

**Fig. 8.** *Choice node for multiple decompositions.*

*4) Reducing Structural Bias:* Chatterjee *et al.* [10] used the idea of choice nodes in a more general way. Observing that the minimum-literal netlist may not yield the smallest mapped area after mapping, they saved several versions of the netlist generated during technology-independent optimization. These versions are combined into a single composite netlist. Equivalence checking is used to identify structurally identical as well as logically identical nodes. Only a single copy of structurally identical nodes is retained. Choice nodes are introduced to select between logically identical (but structurally different) nodes. Fig. 9 illustrates this idea. $N_1$ and $N_2$ are two versions of the netlist generated during logic optimization. They are combined into a single netlist $N$, where structurally equivalent nodes $p_1$ from $N_1$ and $p_2$ from $N_2$ are merged into a single node $p$. A choice node is introduced in $N$ between logically equivalent (but structurally different) nodes $x_1$ and $x_2$.

## C. Minimum-Delay Technology Mapping

The need for automatically synthesizing high-performance designs is the main motivation for minimum-delay technology mapping. One of the earliest algorithmic attempts was by Rudell [46], in which he extended the tree-based min-area mapping to minimum delay. Two main changes were made.

1) Start with a netlist with the minimum number of logic levels (measured in terms of two-input AND/OR gates). This was followed by a timing-driven decomposition into two-input gates.
2) The cost $A$ was changed from area to arrival time. During the match generation phase, arrival time at the output of node $v$ using a match $M$ corresponding to gate $g$ is computed as maximum over all inputs $i$, the sum of arrival time at input $i$, and the gate delay from $i$ to the gate output

$$\begin{aligned} A(v,M) &= \max_i \{A(i) + \delta(i,g)\} \\ M^*(v) &= \arg\min_M \{A(v,M)\} \\ A(v) &= A(v,M^*). \end{aligned}$$

This solution works with load-independent delay model. However, when load-dependent model is being

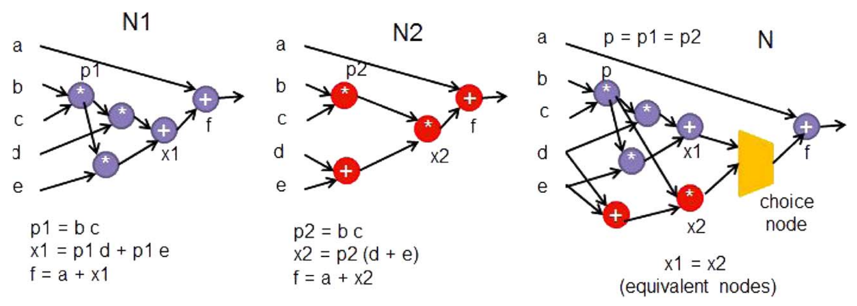

**Fig. 9.** *Reducing structural bias.*

used, the gate delay cannot be computed during the forward phase, since the fanouts of the vertex $v$ are unmapped, and therefore the load at $v$ is not known. Rudell proposed a load-binning solution to address this issue. First, an output load range is estimated for each node $v$. This range is discretized into loads $\ell_1, \ell_2, \ldots, \ell_k$. During the forward pass, the algorithm computes and stores at each node $v$ the min-delay match $M^*(v, \ell_i)$ for each discretized load $\ell_i$. During the backward pass, when we reach a vertex $v$, its fanouts have already been mapped and hence its output load is known. This load is snapped to the nearest discretized load, say $\ell_j$. We pick the minimum delay match $M^*(v, \ell_j)$.

Rudell's algorithm was optimum under LIDM only for trees or leaf dags.[3] For DAGs, Kukimoto *et al.* [29] proposed an optimum minimum-delay technology mapping algorithm under LIDM. This algorithm is based on a generalized min-delay mapper for lookup-table-based field-programmable gate arrays (FPGAs), proposed by Cong and Ding [13]. It works just as the tree mapper, except it generates matches across tree boundaries. It is possible that during the backward pass, some logic is covered multiple times (for instance, by two different fanouts of a tree root). This can result in an area penalty.

We would like to mention that minimum-delay DAG technology mapping problem is NP-complete for load-dependent delay model [38].

### D. Min-Area Mapping Subject to Delay Constraints

An important variant of the technology mapping problem is to generate a minimum area netlist that meets the delay (or clock cycle) constraints. Let us call this problem area-delay (AD) mapping problem. For the circuit to function correctly, the delay of a path starting from the output of a sequential element and ending at the input pin of a sequential element should be within the clock cycle. The min-delay mapper minimizes each path delay, often to a value below the clock cycle, but at the expense of additional area.

Chaudhary and Pedram [11] propose an algorithm to solve the AD mapping problem. It works in the same way as the min-area or min-delay mappers—by traversing the subject graph from inputs to outputs. However, at each vertex, instead of storing a single solution, it maintains a set of solutions. A solution is a pair (A, D), where A is the area and D is the arrival time corresponding to a match. Given a match $M$ at a vertex $v$, the solution sets at the inputs of $M$ are used to calculate the solution set at the output of the match $M$. The union of solution sets from all matches at $v$ is computed. A solution that has worse (or equal) area and delay values as compared to another solution is dropped. Finally, when an endpoint of the subject graph (i.e., a primary output or flop/latch input) is reached and its solution set is computed, the solution whose D value meets the clock cycle (or the delay constraint) and has the smallest area value A is the desired solution to the AD mapping problem.

A drawback of the algorithm is that the solution sets tend to grow large, often resulting in large runtime and memory overhead. One way to address this problem is by limiting the maximum number of solutions at a vertex. However, this may result in suboptimality.

## IV. POSTMAPPING OPTIMIZATION

Exact technology mapping is an intractable problem under most practical scenarios: unconstrained area minimization [27], unconstrained delay minimization under load-dependent delay model [38], area-constrained delay minimization even under load-independent delay model [32], etc. This is due to the inherent difficult nature of the problem, the delay model, the complex interactions between the gate being mapped and the unmapped portion of the logic, and absence of the layout information. For instance, we have the following.

1) Typical mapping algorithms work from circuit inputs toward outputs. Assume that the goal of mapping is to generate a minimum-delay circuit. When visiting and mapping a gate $g$, these algorithms need to compute the arrival time at the output of $g$. This, in turn, requires that $c_g$, the capacitive load at the output pin of $g$, be known ($c_g$ is the sum of the input pin capacitances of the fanout gates of $g$ and

[3]A leaf dag is a tree, with multiple fanouts allowed at primary inputs.

the loads of the wires connected to the output of $g$). However, that is not possible because of the following reasons.

a) The fanouts of $g$ have not been mapped yet.

b) Since mapping is done before the layout, information about the gate locations and net routes is not available during mapping. The wire lengths and loads cannot be determined then. Mappers usually use statistical wire load models to estimate the wire lengths and loads. However, these models are inherently inaccurate.

The mapping algorithms compute the arrival time at $g$ either for all possible $c_g$ values (which renders the algorithms more complex and CPU intensive) or for a discrete subset (which is at the cost of accuracy).

2) For deep submicrometer technologies, a significant component of the circuit delay is due to the wire delays. However, during mapping, in the absence of the layout information, wire delays can only be estimated say using statistical models. Also, timing values change after clock synthesis due to clock skew. The arrival times at the input pins of the fanout gates of $g$ are thus not computed accurately.

Due to the inherent complexity of the problem, heuristics are often used during technology mapping. In addition, inaccuracies in delay values are introduced due to the missing layout information. These factors render the resulting mapped circuit suboptimal and leave room for further improvement after mapping. For instance, in a mapped circuit, the fanout gates of a gate $g$ are known and so is $c_g$ (modulo the wiring load). This information can be used to select the delay-optimal size for $g$, given the current choices for the remaining gates. If layout details are also available, the wire load uncertainty can also be removed.

Postmapping transformations can improve the circuit characteristics (such as delay, area, power, routing congestion, signal integrity) before, during, or after the layout. Such transformations utilize a technology library to modify the mapped circuit. They may either focus on a part of the circuit and peephole optimize it, or consider the entire circuit and improve some particular characteristic using a specific transform. For instance, in the LATTIS system proposed by Fishburn [20], a number of transforms such as gate resizing, buffer insertion, gate duplication, DeMorgan, timing-directed factorization, remapping, and pin permutation were performed on a mapped netlist. The goal was to speed up the circuit with minimum area penalty.

We present the following logic transforms: gate resizing, net buffering or fanout optimization, gate replication, simple gate decomposition and collapsing, resynthesis and remapping, and pin permutation. We focus on three widely used design objectives. Constrained delay minimization (CDM) minimizes circuit delay subject to constraints such as area, pin drive capacities, slews, hold time, etc. Unconstrained delay minimization (UDM) minimizes circuit delay without regard to any constraints. UDM is a special case of CDM. Constrained area minimization (CAM) attempts to recover maximum circuit area without violating any constraints including those on pin delays. Typically, a combination of these objectives is used to optimize the circuit. For instance, after minimizing the circuit delay with CDM, CAM is applied to recover area from nondelay-critical portions of the circuit.

## A. Gate Resizing

We are given a mapped circuit composed of gates from a gate library. For each gate $g$, several different sizes $1, \ldots, k, \ldots$ are available in the library, each size having identical logic function but different area, input loading $\gamma$, drive capacity $\lambda$, and delay coefficients such as $\alpha$ and $\beta$. The gate resizing problem is to select the size of each gate such that some objective function (such as circuit delay, circuit area) is minimized or reduced without violating any constraints.

Gate resizing is an in-place optimization technique. It has minimal impact on placement and routing of cells, and can be applied when more accurate wire load and delay estimates become available (e.g., during or after placement, or postrouting).

We now consider gate resizing in the context of all three design objectives.

*1) Unconstrained Delay Minimization:* The UDM problem for gate resizing a circuit composed of single-output gates is solvable in polynomial time under LIDM using a dynamic programming algorithm [29]. This algorithm works as follows. Traverse the network gates in a topological order from primary inputs toward primary outputs. When a gate $g$ is reached, the best possible sizes for the gates already visited (including those in the transitive fanin of $g$) have been computed, and the minimum possible arrival times at all the input pins of $g$ are known. Let $g^k$ denote assigning the size $k$ to gate $g$. For each available size $k$ of the gate $g$, compute $A(g^k)$, the arrival time at the output of $g^k$, using the arrival times at its input pins $j$ and the pin-to-pin delays $\alpha(j, g^k)$. Pick the size $k^*$ that minimizes $A(g)$ and replace $g$ with $g^{k^*}$. Continue the traversal and size selection until the primary outputs are reached. The algorithm is exact in that it yields the minimum possible circuit delay. Its runtime complexity is linear in the size of the circuit and the gate library.

Although the above algorithm can be adapted for LDDM, it ceases to be exact. When gate $g$ is being considered for resizing to size $k$, $A(g^k)$ depends on the input pin capacitances of the fanout gates. However, the best sizes of the fanout gates (and hence their input capacitances) are not known yet. So $A(g^k)$ is simply an estimate given the current choice of the fanout gates, and the size $k^*$ may not be truly optimum. For a tree circuit, Rudell proposed a technique that considers all possible resizing choices for the fanout gate $f$ and determines and stores the best size of $g$ for each such choice $\ell$ [46]. Later, when $f$ is being considered for resizing, say to size $\ell$, the corresponding best size of $g$ is looked up and used to compute the arrival time at the corresponding input of $f$ and finally at the output of $f^{\ell}$, $A(f^{\ell})$. For a nontree circuit, a straightforward application of this technique does not work. Assume that the gate $g$ has two fanouts $f_1$ and $f_2$. The resizing choice $\ell$ for the fanouts corresponds to a size for $f_1$ and a size for $f_2$. When the algorithm visits the first fanout $f_1$ to compute its best size, the choice of size of $f_2$ is not known. Thus, one best choice of $g$ is not possible for $f_1$. The algorithm needs to consider $f_1$ and $f_2$ simultaneously for resizing. However, similar simultaneity constraints are introduced by other fanin gates $v$ of $f_1$ and $f_2$. The fanouts of all such gates $v$ must also be resized along with $f_1$ and $f_2$. This may result in a runtime explosion. For instance, if ten gates have to be simultaneously resized and each gate has five sizes, $5^{10} \sim 10$ million different choices need to be explored. So Rudell's technique is practical only for a tree circuit. Indeed, under LDDM, as well as ISDDM, gate resizing becomes NP-complete for general circuits [38]. The problem remains NP-complete for a circuit with multiple-output gates even under LIDM [32]. So far we assumed a single delay value for all parameters of a pin-to-pin path (or arc) within a gate. In practice, however, there are separate values for rise and fall delay and capacitance parameters. In that case, the UDM problem becomes NP-complete for single-output gate circuits even under LIDM [40].

Given the intractability of the practical gate resizing problem, several heuristics have been proposed: greedy, global sizing—a greedy technique enhanced with a mechanism to get out of local minima [14], a cutset-based approach [50], sizing down noncritical fanouts of critical gates [20], etc. They differ in how they pick the gates for resizing and decide their new sizes.

2) *Constrained Delay Minimization:* Gate resizing problem for area-constrained delay minimization is NP-complete under the simplest case: LIDM and for single-output gates [32]. We present a greedy heuristic in Fig. 10 [8].

The greedy algorithm first determines the best "local" size for each gate $g$ in the critical subcircuit of the circuit $C$. Specifically, it recalculates delays in as many gates as necessary according to the delay model so that the slacks of all the pins of $g$ can be correctly recalculated. The size which provides the best slack is kept as $s(g)$. Next, out of the entire set $T$ of best resizings for the circuit, only those that are useful (i.e., guarantee not to degrade the circuit's performance) are selected in the set $T'$. A cost-benefit analysis is done to compute the set $T'$. The circuit is then updated, i.e., the gate resizings contained in $T'$ are implemented, and the circuit's delays are recalculated. This process continues while there are useful gate resizings. When $T'$ is empty, the algorithm terminates. The set of resizings $T'$ is applied in the order, from the "best" to the "worst." If during this application enough area is not available for a resizing, it is not performed. This preserves the area constraint.

This heuristic is versatile. It can be applied under any delay model and in the presence of multiple-output gates. It is also economical in area, because it works only on the critical portion of the circuit. However, being greedy, it can get stuck in a local minimum. Some possible ways to fix this problem are to accept even "bad" resizings or to perturb the circuit enough so that it gets out of the local minimum [14].

3) *Size Down:* The greedy algorithm presented above only resizes critical gates of the circuit. One can also resize noncritical gates and still obtain delay improvements. For instance, given a critical gate $g$, $A(g)$ can be reduced if its output load $c_g$ can be reduced. To do so, the algorithm identifies noncritical fanout gates $f$ of $g$ and resizes/shrinks them so that input pin capacitance of $f$ is reduced. This shrinking can be carried out for the fanouts of all the critical gates $g$ in the circuit.

4) *Constrained Area Minimization:* One reasonably effective algorithm to solve constrained area minimization is as follows. First, identify a set of candidate gates whose possible shrinking does not increase the circuit delay. For each candidate gate $g$, the size $g^{k*}$ that yields the maximum area reduction without causing the circuit delay to increase is computed. All candidate gates are shrunk to these sizes. A delay trace is then performed on the entire design. If the circuit delay does not become worse, stop. Otherwise, identify an appropriate subset of resized gates which caused the circuit delay to increase and undo their resizings.

So far, we assumed that the gate sizes are discrete. Resizing algorithms have also been proposed for the case where a gate or a transistor can be sized continuously. In such a case, the delay is modeled as a continuous function of the sizes of the transistors/gates and mathematical programming techniques are used to derive the optimum sizes of the transistors in the entire design. Such approaches include those that model area and delay as posynomials [21], linear programming approaches using a piecewise linear delay model [5], and convex programming-based approaches [49].

```
resize_circuit($C$)
    repeat
        for all gates $g \in \epsilon$-critical $C$
            $s(g) = find\_best\_local\_size(g)$
        $T \leftarrow \{g | s(g)$ "improves" $g\}$
        $T' \leftarrow useful\_subset(C, T)$
        update $C$ with $T'$
    while $T' \neq \emptyset$
```

**Fig. 10.** *Greedy gate resizing algorithm.*

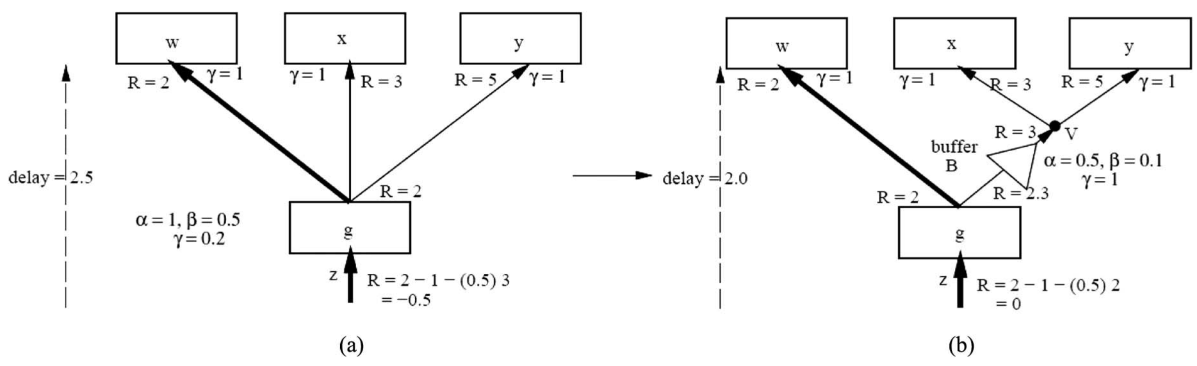

**Fig. 11.** ***Buffering can reduce the critical net delay.***

## B. Fanout Optimization/Buffering

Fanout optimization seeks to optimally distribute a signal from the driver gate (source) to the fanout gates (sinks) using buffers, without violating the drive capacity of the driver or those of the inserted buffers. The objective could be a combination of the following: minimize delay through the critical parts of the net, fix the overloading problem at the source of a large fanout net incurring minimum delay and area penalties, reduce the slew (slope degradation) of the signal with minimum delay and area penalties. For instance, buffering minimizes the arrival time at the output of a critical gate $g$ by reducing the total capacitive load driven by $g$. The added delay due to inserted buffers is along the noncritical paths.

Buffering algorithms are often based on computing the required times. Let $R(g)$ denote the required time at the output of a gate $g$. Given the required times $R(x)$ at the fanout gates $x$ of $g$, $R(g)$ can be computed as

$$R(g) = \min_{x \in FO(g)} \{R(x) - \alpha(p,x) - \beta(p,x)\ c_x\} \tag{3}$$

where $p$ is the input pin of $x$ to which $g$ is connected. The following example illustrates how buffering can increase the required time at the input of the source gate, thus reducing the delay through the critical portion of the net.

*Example 4.1:* Consider the net driven by gate $g$ of Fig. 11(a). The net has three sinks: $w$, $x$, and $y$. Let the required times at the inputs of the sinks be 2, 3, and 5, respectively (so $w$ is the sink with the tightest timing requirement). Let $\gamma$, the input pin capacitance for each sink pin, be 1. $R(g) = \min\{2, 3, 5\} = 2$. The total capacitive load driven by $g$, $c_g = 1 + 1 + 1 = 3$. Given that $\alpha(z, g) = 1$ and $\beta(z, g) = 0.5$, $R(z) = R(g) - 1 - (0.5)c_g = -0.5$. The delay from $z$ to the critical sink $w$ is $2 - (-0.5) = 2.5$.

Let us introduce a fanout tree and a buffer $B$ at $g$, as shown in Fig. 11(b). Let $\alpha(U, B) = 1$, $\beta(U, B) = 0.5$, and $\gamma(B) = 1$. Then, $R(B) = 3$ and the required time at the input $U$ of $B$, $R(U) = 3 - 0.5 - (0.1)(1 + 1) = 2.3$. $R(g) = \min\{2.0, 2.3\} = 2.0$, same as before. But now $c_g = 1 + \gamma(B) = 1 + 1 = 2$. Then, $R(z) = 2.0 - 1 - 0.5(2) = 0.0$, an improvement of 0.5 over the unbuffered case. The delay from $z$ to $w$ is now 2.0.

Buffering caused $R(z)$ to increase because the capacitive load seen at $g$ is reduced. Although the buffer $B$ adds additional delay in the paths from $g$ to $x$ and $y$, these are not the most critical paths in the net. In this example, $R(g)$ remained the same. Since the capacitive load at the output of $g$ decreased by one unit, this reduced the delay through $g$ and increased $R(z)$. In fact, it is possible that $R(g)$ reduces but $R(z)$ increases [e.g., consider $\alpha(B) = 1.0$].

The only external values needed to compute the required time at the input of the source $g$ are the required time $R(g)$ at the output of $g$ and the capacitive load $c_g$. A buffering scheme for the net rooted at $g$ can then be encapsulated by a $(c_g, R(g))$ pair, also called a solution [42].

*2) Local Fanout Optimization (LFO):* The fanout optimization problem for a single gate/net is called local fanout optimization (LFO). For UDM, the problem can be stated as follows.

Given a library of buffers and inverters, the source gate $g$ of a net $N$, $n$ sinks of the net $N$ with separate required times $R(i)$ and polarities, find a tree of buffers and inverters that distributes the signal at $g$ to all the sinks in the correct polarities and maximizes the required time at the input of the source $g$ without violating the drive limits of $g$ or the inserted buffers.

For CDM and CAM, the problem can be similarly stated.

Two flavors of fanout optimization can be distinguished. In the first, we have two degrees of freedom: 1) determine the topology or structure of each net tree; and 2) select specific buffers/inverters and their locations on the tree edges and nodes. We will call this local fanout optimization/net topology unknown (LFO–NTU). An example of this flavor was shown in Fig. 11, where a branching node (Steiner node) $v$ was introduced on the net tree. In the second flavor, each net's tree topology is already determined, i.e., all the

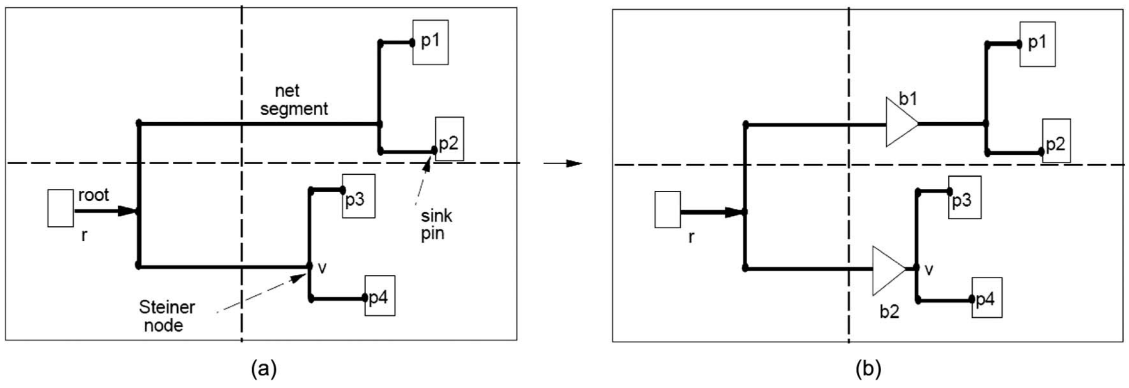

**Fig. 12.** ***Buffering a fixed topology net.***

branching nodes are already known, determined either by a fanout tree generator or a global router; only the buffer types and their insertion lo, cations need to be determined. We call this LFO–NTF (for net topology fixed). An example of such a case is shown in Fig. 12. Note that LFO–NTF is a special case of LFO–NTU, in that LFO–NTF has just the second degree of freedom of LFO–NTU.

LFO–NTU problems for both UDM and CAM are NP-complete [6], [57]. Several heuristic solutions such as combinational merging [22], recursively partitioning the fanouts into critical and noncritical subtrees [53], two-level trees, balanced trees, LT trees [57], etc., have been proposed. In combinational merging, the sinks are sorted in increasing order of their required times in a list $L$. Then, we take a group of sink nodes with the largest required times, make them the children of a new node $v$, and remove them from $L$. We then compute the best buffer $b*$ that maximizes the required time at $v$, insert $b*$ at $v$, and merge the corresponding required time at $v$ with $L$. This process is repeated until only one entry remains in $L$, which is then driven directly by the source gate. The main issue in this algorithm is deciding the number of sinks to be selected at each step. In two-level trees, each sink is separated from the source gate by one Steiner node. LT trees are defined inductively as follows: 1) a leaf is an LT tree; 2) a two-level tree is an LT tree; and 3) let $T$ be a tree rooted at $r$ such that one child of $r$ is an LT tree and all the other children of $r$ are leaves. Then, $T$ is an LT tree. They are a small subset of all possible tree structures, but can effectively perform both large capacitive load buffering and critical signal isolation. A dynamic programming algorithm can be used to determine the shape of the best LT tree [57]. The best buffering choices can be then made using the optimum buffer selection algorithm, as described below for LFO–NTF.

On the other hand, LFO–NTF for UDM under LDDM can be solved in polynomial time by a dynamic programming algorithm [42], [57]. This algorithm traverses the net from the sink nodes to the root. It builds a set of solutions at each node. A solution at a node captures a buffering scheme on the subtree rooted at that node (the buffering scheme indicates which buffers will be inserted at which net nodes). As mentioned earlier, a solution is a pair $(c, R)$, where $c$ is the total capacitive load seen at the node for the buffering scheme and $R$ is the corresponding required time. At the net root, the best solution $\sigma^*$ is chosen. This is the solution that maximizes the required time at the inputs of the root cell (and hence minimizes the delay through the most critical parts of the net). The buffering scheme corresponding to $\sigma^*$ is a set of buffers that should be inserted along with their node locations. For the input-slew-dependent delay model ISDDM, the algorithm becomes more complex. When a node $v$ is visited, the input slew is not known. Thus, $R$ cannot be uniquely determined; it becomes a function (a piecewise linear function, in fact) of the input slew and is stored as such in the $(c, R)$ pair [33].

The buffering algorithms for CDM and CAM need to track the area used on the net [37]. This is done by storing at $v$ the total area $a$ of the buffers inserted in the subtree rooted at $v$. Thus, the solution becomes a triplet $(c, R, a)$. At the root, the best solution is appropriately chosen. For instance, in CAM, the best solution is the one that minimizes the total buffer area on the net without lowering the slack of the source gate below the minimum circuit slack. With the introduction of the third component, the algorithm starts exhibiting exponential behavior in the worst case. But, in practice, the runtimes are acceptable for most of the nets. Theoretical complexities of CAM and CDM remain open problems.

Extensions of the basic algorithm to handle optimal wire segmenting to determine good candidate buffer locations [1], noise [2], resistive shielding effects [3], and low power [33] have also been proposed.

*3) Global Fanout Optimization (GFO):* GFO is concerned with fanout optimization of the entire circuit. The objective could be to minimize the circuit delay (equivalently, maximize the minimum required time at the primary inputs), minimize the buffer area subject to the delay constraints, etc.

Given a Boolean network $\eta$ consisting of gates (belonging to a gate library $\mathcal{L}$) and nets, a set of buffers and inverters of $\mathcal{L}$, the required times at the primary outputs, find trees of buffers and inverters that distribute each signal to the sinks and satisfy the objective function without violating the drive capacities of any gates or buffers.

Once again, depending upon whether the net topology is fixed, there are two versions of GFO: GFO–NTF and GFO–NTU. It is now known that both GFO–NTU and GFO–NTF are NP-complete under LDDM [6], [39].

Many approaches for GFO use LFO as a subroutine. Singh and Vincentelli [53] apply LFO to a cutset of critical nets. Hoover *et al.* [23] and Touati [57] apply LFO to each net in a reverse topological (RT) order starting from the primary outputs. Berman *et al.* [6] use integer linear programming to select nets where LFO should be applied along with the best buffering solution for each net. Liu *et al.* [34] use a Lagrangian relaxation formulation to solve the GFO problem for minimum buffer area subject to delay constraints.

The RT algorithm is perhaps the most promising GFO algorithm for delay minimization under LDDM. It works as follows. The nodes (gates) of the network are traversed in a reverse topological order: from primary outputs to primary inputs. At a node $v$, the LFO algorithm is applied to the net rooted at $v$ and the new fanout tree is constructed. As a result of the new fanout tree, the required time at $v$, $R(v)$, might have changed. So it is recomputed and propagated to the fanin nodes. Note that when the LFO algorithm is applied at $v$, the required times used for the sink nodes of the net at $v$ are the updated ones. The algorithm terminates when all the network nodes have been processed.

For example, consider the network of Fig. 13. The RT algorithm visits all the network nodes, in the increasing order of the node numbers: from 1 to 10. After buffering the net rooted at the output of node 1 using some LFO algorithm, it updates the required times at node 1's four input pins, which are sinks of the nets rooted at nodes 2, 3, 4, and 6. These updated required times will be used when later these four nodes are visited and the nets rooted at their outputs are considered for buffering.

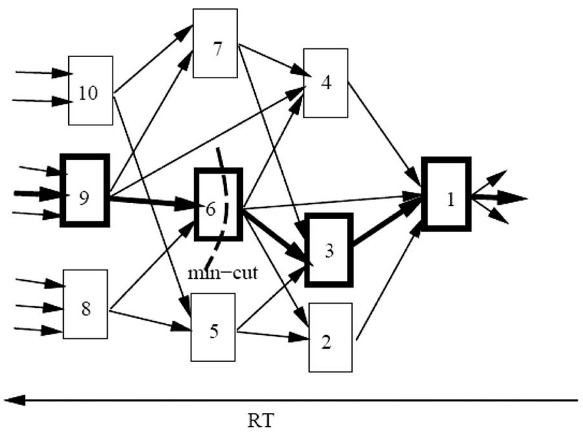

**Fig. 13.** ***Applying various GFO algorithms to a network.***

The RT algorithm has the following nice property. If for each gate $s$ in the circuit the load coefficient $\beta(i,s) = \beta(j,s)$ for all inputs $i$ and $j$ of $s$ (the "identical load coefficients condition"), and if the LFO algorithm is delay optimum, the RT algorithm is delay optimum for the GFO problem under LDDM [39]. It turns out that the RT algorithm yields good delay improvements under both LDDM and ISDDM in circuits implemented with industrial cell libraries, where several gates do not satisfy the identical load coefficients condition.

The main drawbacks of the RT algorithm are that since it applies the LFO algorithm to all the nodes of the circuit—critical or not—it can lead to large area penalty and runtime. By applying the RT algorithm only to critical nodes (shown in bold in Fig. 13) and only propagating the updated required times through noncritical nodes, one can get rid of these problems.

There is also a body of work using gain-based synthesis [56], which works well in the presence of near-continuous libraries. Kung [30] and Rezvani *et al.* [45] have proposed algorithms to construct fanout trees under such a scenario. Using gain-based analysis, these algorithms first build fanout-free chains of buffers and inverters, one chain for each sink. Then, they merge these chains to recover area.

## C. Gate Replication

Gate replication, like buffer optimization, is a technique for speeding up a design by redistributing the fanout load. The main idea here is to make $k$ copies of a gate $g$: $g_1, g_2, \ldots, g_k$, $(k \geq 2)$, partition the fanout gates of $g$ among the $k$ copies, and replace $g$ with $g_1$ through $g_k$. Each copy drives smaller capacitive load than $g$, possibly reducing the delay through the circuit.

This speedup, however, comes at a cost. Replication results in an area penalty. Also, the fanin gates of $g$ drive a larger load now and are slowed down. The speedup of $g$ has to be compared against the slowing down of the fanin gates of $g$, before allowing replication. Gate replication therefore typically uses only small values of $k$, either 2 or 3.

Fig. 14 shows an example of gate replication with $k = 2$. This is the same example as in Fig. 11, except that $f$, the fanin gate of $g$, is also shown, since it becomes relevant. Duplicating $g$ into $g_1$ and $g_2$ and intelligently distributing the fanouts of $g$ to $g_1$ and $g_2$ improves the required time at the input of $f$ by 0.9 units (from $-1.6$ to $-0.7$).

We now describe an algorithm for gate duplication $(k = 2)$ [54]. By repeatedly applying it, general replication can be obtained. The algorithm has three main phases. In the first phase, the gates are traversed in a reverse topological order from primary outputs to primary inputs. When a gate $g$ is being visited, the following questions are asked for each input pin $p$ of $g$.

1) If $g$ is not duplicated, what is the best required time at $p$? This considers possibilities that some

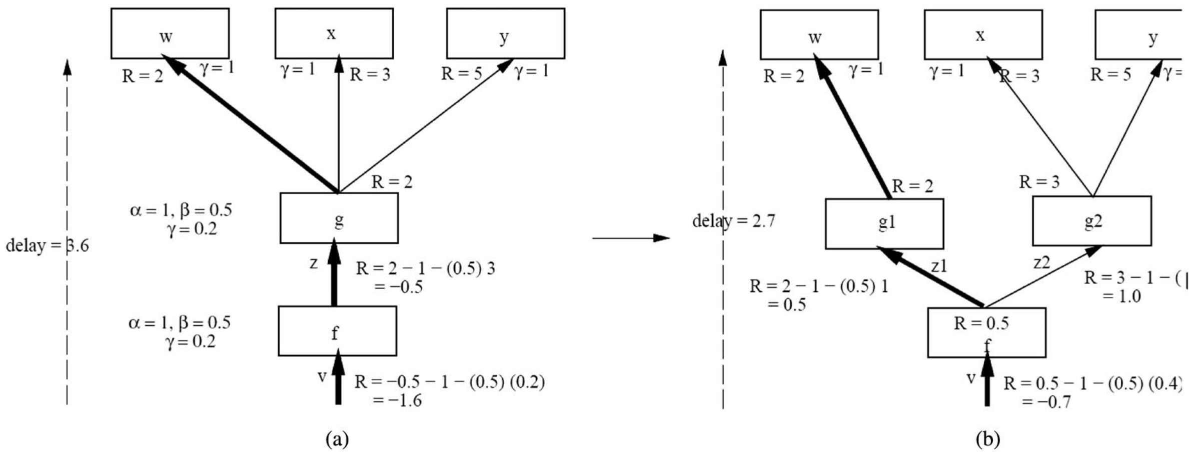

**Fig. 14.** *Replicating gate g into $g_1$ and $g_2$.*

fanouts of $g$ may be duplicated and others not. For each possibility, $R(p)$ is recomputed. The possibility that maximizes $R(p)$ is stored.

2) If $g$ is duplicated into $g_1$ and $g_2$, what is the best required time at the duplicated input pins $p_1$ and $p_2$? The goal is to maximize the minimum of $R(p_1)$ and $R(p_2)$. Once again, the possibilities of some fanouts being duplicated are considered first. Then, for each such possibility, the new fanout set $T$ of $g$ is temporarily created. $T$ is partitioned into two disjoint sets as follows. First, $T$ is sorted in increasing order of required times (the required times used here are the updated ones, after appropriate original fanouts of $g$ have been temporarily duplicated). Then, the best location $j^*$ in this sorted list is determined. All the fanouts before $j^*$ become fanouts of $g_1$, and the remaining are assigned to $g_2$. The best location $j^*$ is that index in the sorted list which maximizes $\min\{R(p_1), R(p_2)\}$.

During the second phase, a set of gates is chosen for duplication. The network is traversed from PIs to POs in a topological order. When visiting gate $g$, the two required times computed in the first phase for the most critical input pin $p$ of $g$ are compared. If option 1) corresponds to a higher required time, $g$ is not duplicated, otherwise it is.

In the third and final phase, duplication of the chosen set of gates is carried out from POs to PIs. The fanout partitioning for each duplicated gate is done as described in option 2) of the first phase.

Although buffering and gate replication target the same problem of redistributing and optimizing the fanout loads, there are some differences.

1) Unlike gate replication, buffering does not change the capacitive loads driven by the fanin gates of the source gate $g$.
2) In gate replication, additional nets are added to the inputs of the replicated copies, possibly increasing the routing complexity.
3) In the context of layout-driven optimization, if the gate being replicated is large, enough space needs to be available for placing the copies, and if the copies cannot be placed close to the original gate, the routing of the fanout net may also be affected.
4) Buffering adds additional logic stages, mostly on noncritical paths. If all the fanouts of $g$ are critical, it may be more beneficial to replicate $g$ instead of buffering it.
5) Whereas buffering can optimize a long wire (i.e., a net with one sink), gate replication cannot.

## D. Simple Gate Decomposition/Collapsing

A simple gate is an AND, OR, NOR, NAND, XOR, or XNOR gate. We consider decomposition of multiple-input (i.e., with at least three input pins) simple gates into two or more simple gates present in the gate library. The advantage of simple gate decomposition is that a mapped design remains mapped after the transform. This transform identifies delay-critical multiple-input simple gates in the design and considers them one by one for decomposition into two simpler gates. For instance, it may decompose a four-input AND gate $g$ into a two-input AND gate $g_1$ followed by a three-input AND gate $g_2$, assuming both $g_1$ and $g_2$ are in the library. For a candidate gate $g$, all possible choices of simpler gates $g_1$ and $g_2$ (with $g_1$ feeding $g_2$) that can replace $g$ are considered. The pin assignment for $g_1$ and $g_2$ (i.e., how the fanin signals of $g$ are partitioned between $g_1$ and $g_2$) may be chosen greedily. The choice of gates $g_1$ and $g_2$ that yields largest reduction in the local delay at the output pin of $g_2$ is chosen to replace $g$. Area constraints can be easily incorporated. As with gate replication, decomposition into more than two gates can be achieved by repeating the procedure on the new design.

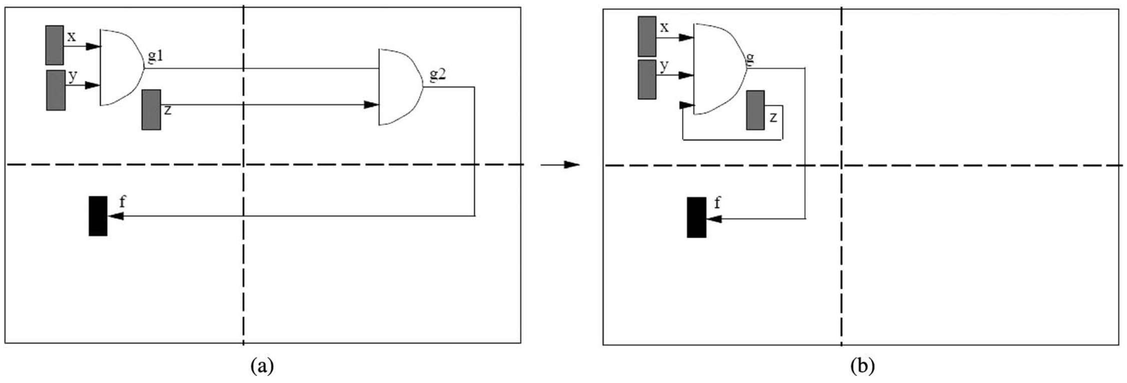

**Fig. 15.** *Simple gate collapsing.*

In the context of layout-driven decomposition, the new locations of $g_1$ and $g_2$ need to be determined. A force-directed incremental placement procedure or a greedy move-based local algorithm [58] can be used.

There can be situations where simple gate collapsing, which is the inverse of the decomposition transform, can be useful. Simple gate collapsing searches for two simple gates $g_1$ and $g_2$ of the same type, with $g_1$ directly feeding $g_2$, and considers to replace them with their composite gate $g$ if the circuit timing or routing congestion improve. One situation where both delay and congestion improve is shown in Fig. 15. The circuit delay improves because the long wire from $g_1$ to $g_2$ disappears and the one from $z$ to $g_2$ is replaced by a much shorter wire to $g$. Similarly, the long wire from $g_2$ to $f$ is replaced by a shorter wire from $g$ to $f$. Also, two stages of gate delays are replaced by one. An algorithm similar to the one for decomposition can be used for collapsing.

## E. Resynthesis and Remapping

The reason we considered simple gate decomposition and collapsing separately is because in order to extend them to complex gates, we need a general mapping algorithm. If we have such an algorithm at our disposal, we can do powerful restructuring.

In remapping, first we identify regions of the design that are critical. Then, each region is resynthesized using techniques such as tree height reduction [19], critical path resynthesis [52], logic minimization [18], etc. Finally, we remap the region and compare the combined cost (such as delay, area, etc.) of the remapped region with the original region. If it has improved, the original region is replaced with the new one.

In the context of an already placed (and possibly routed) design, the newly mapped region needs to be incrementally placed (and routed) after deleting the gates and nets associated with the old region. Depending on the number and size of regions resynthesized, this process can perturb the layout significantly and should be carried out earlier in the physical flow.

## F. Pin Permutation

Consider a two-input AND gate $g$ with inputs $x$ and $y$ embedded in a circuit. Let the intrinsic delays from the inputs to the output $g$ be $\alpha(x, g) = 2$ and $\alpha(y, g) = 1$. Assume the circuit configuration results in the arrival time at $x$, $A(x) = 3$, and $A(y) = 2$, as in Fig. 16(a) (i). Then, under the load-independent delay model, $A(g) = \max\{3 + 2, 2 + 1\} = 5$. Since $g$ is symmetric with respect to $x$ and $y$, we can switch the incoming signals without changing the logic functionality, as shown in Fig. 16(a) (ii). Now $A(g) = \max\{3 + 1, 2 + 2\} = 4$. By exchanging the signals at symmetric input pins of a gate such that the late-arriving signal is reconnected to an input pin with smaller arc delay, we are able to reduce the arrival time at $g$ by one unit. This is the basic idea behind pin permutation.

Pin permutation (called REPLUGGING in [20]) makes sense because the technology mapping tool may not have incorporated the best possible pin matching. Even if it had, the arrival times change during layout and application of logic transformations. Then, pin permutation can come handy. Moreover, it is simple and layout friendly: it does not disturb any gate placement and affects routing minimally—only the connections to pins of the same gate need to be permuted.

The goal of pin permutation is as follows: Given a circuit $\eta$, apply pin permutation to the gates of $\eta$ such that $\eta$'s delay is minimized. Before addressing the problem for the entire circuit, we briefly outline the solution for a single gate $g$. Assume $g$ has a single output. Two inputs $a$ and $b$ are in a symmetry class for $g$ if $g$ is symmetric with respect to them, i.e., $g(a, b, \ldots) = g(b, a, \ldots)$. In general, a gate can have several symmetry classes. For instance, the symmetry classes of $g = ab + cde$ are $\{a, b\}, \{c, d, e\}$. The symmetry classes of a gate can be determined efficiently using ROBDD/s by noting that $g$ is symmetric with respect to $a$ and $b$ if and only if $g_{ab'} = g_{a'b}$.[4] For the load-independent

[4]The function $g_a$ is the cofactor of $g$ with respect to $a$ and consists of those on-set min terms of $g$ that have $a = 1$.

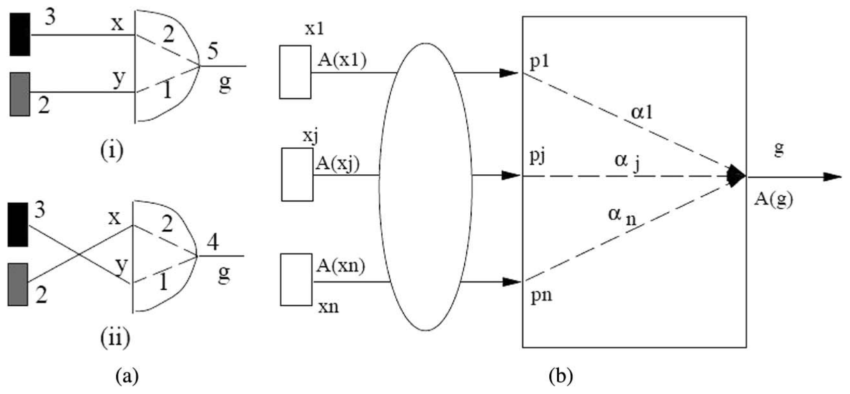

**Fig. 16.** *Pin permutation: (a) exploiting asymmetric delays through symmetric pins; and (b) problem formulation.*

delay model, the problem for a single gate $g$ can be stated as follows: Given a symmetry class $\{p_1, \ldots, p_n\}$ of input pins of $g$. The pin $p_j$ is connected to signal $x_j$, which has arrival time $A(x_j)$. Let $\alpha_j$ be the arc delay from pin $p_j$ to the output of $g$. Connect signals $\{x_i\}$ to pins $\{p_j\}$ such that the arrival time at the output of $g$, $A(g)$, is minimized. If $g$ has two or more symmetry classes, they can be treated sequentially. See Fig. 16(b). The simple algorithm that sorts $\{A(x_i)\}$ in increasing order, sorts $\{\alpha_j\}$ in decreasing order, and matches the corresponding entries in the two sorted lists [i.e., if $A(x_i)$ matches $\alpha_j$, connect signal $x_i$ to pin $p_j$], yields the minimum value of $A(g)$.

For the load-dependent delay model, since the input pin capacitance value can be different for different pins, the arrival time of signal $x_i$ may change depending on which input pin of $g$ it is connected to. Then, the previous algorithm is no longer optimum. However, assuming that $x_i$'s are mutually independent (i.e., they are not in the transitive fanin or fanout of each other), one can still propose an exact, polynomial-time algorithm to minimize $A(g)$ via pin permutation. This is based on solving a bottleneck matching problem in a related graph [20]. If $g$ is a multiple-output gate, even that formulation needs to be modified [8].

For the entire circuit, one could apply the single-gate technique to all the critical gates with symmetric inputs. Although pin permutation is not as powerful as gate resizing or net buffering, it yields timing improvements quickly at no cost in area [8].

## G. Merging Multiple Optimizations

One area that has received significant attention in the research community is executing a group of design and optimization steps simultaneously rather than sequentially, for example, simultaneous technology mapping and placement [26], [36], [44]; combined floorplanning, technology mapping, and linear placement [47]; combined routing tree construction and fanout optimization [41], [48]; and combined logic restructuring and placement [35].

To provide a flavor of this body of work, we describe the main idea of the layout-driven technology mapping algorithm of [44]. This was among the first works to combine technology mapping with layout. The main objective was to incorporate interconnect length and wire delay during mapping. Before mapping, input/output (I/O) pin assignment (for primary inputs and outputs) is done, followed by an initial placement $P$ of the nodes of the subject graph. Then, a dynamic programming algorithm, similar to the one described in Section III-B, is used for mapping. For each match at a given vertex, the mapped location of the match is computed based on the mapped locations of the match inputs and the locations of the unmapped fanout gates in the initial placement $P$. The area cost is modified to include both the gate area and the wirelength of the nets incident on the gate. The placement locations are used to estimate the wirelength cost. For the delay-oriented mapper, the arrival time calculation takes into account the wire delays as well.

In theory, these algorithms can result in better design quality. However, the runtime increase is significant. The applicability of such techniques on large industrial designs and design flows, and extensibility to arbitrary cost functions are yet to be proven.

## H. Comparing Postmapping Transforms

Table 1 shows the scope and potential impact of various postmapping transforms. The scope covers different stages in the design flow where the transform can be applied, such as postmap preplacement, postplacement, etc. For instance, local structuring and remapping can be applied postmapping, both before and after placement. It could potentially be applied post global route too, but not after detailed routing. Since it is the most general transform, its quality of results (QoR) impact is large. Gate resizing and buffer optimization can be applied after global routing, and their QoR impact is also significant. Gate resizing can even be applied after detail routing, since its impact on routing is local. Pin permutation is

Table 1 Scope and Relative Impact of Postmapping Transforms

| Transform | Post-map pre-placement | Post-placement | Post-global-route | Post-detail-route | QoR impact |
|---|---|---|---|---|---|
| Local restructuring/remapping | Yes | Yes | Possible | No | Large |
| Gate resizing | Yes | Yes | Yes | Yes | Large |
| Fanout optimization | Yes | Yes | Yes | No | Large |
| Gate replication | Yes | Yes | Yes | No | Medium |
| Pin permutation | Yes | Yes | Yes | Yes | Small |

layout friendly and can be applied postdetail routing. However, its QoR impact is small.

## V. CONGESTION OPTIMIZATION

Due to increased design complexity, routing congestion has become a major problem in the design of very large scale integration (VLSI) circuits. Congestion manifests when the routing demand exceeds the routing resources available on the chip or parts of the chip. This can happen due to high chip utilization (i.e., too many cells in the core area), bad floorplanning (e.g., poor macroplacement, narrow routing channels), poor synthesis (e.g., excessive logic sharing, poor multiplexor design), or their combination. As a result, we end up with either an unroutable design or a design that fails to achieve timing closure due to routing detours. In either case, the design process has to be restarted from an earlier stage, such as RTL redesign, logic optimization, floorplanning, or placement. This can increase the design turnaround time significantly.

To fix congestion, the designer needs to first understand the root cause of congestion and then modify the culprit design modules and/or flow steps. During redesign, a congestion-prediction metric is required. Given different design choices, this metric helps select the choice that will most likely lead to zero or minimum congestion post routing. For logic resynthesis, a purely structural metric for congestion has been proposed in [28]. It is based on the concept of adhesion, a property related to the connectivity of the network graph that implements the logic function. Adhesion of a logic network represented by an undirected graph $G(V,E)$ is the minimum number of edges between all vertex pairs $s,t \in V$ which when removed from $E$ will disconnect $G$. One way of measuring adhesion is by computing the sum of all pairs min-cut for the graph.[5] Empirically, it is shown that networks with smaller adhesion have better routability post placement. The objective function of logic optimization transforms such as kernel extraction is modified, in that both literal savings and adhesion are incorporated. It is shown that the maximum congestion is reduced as a result of this change. One drawback of the adhesion metric is the large computation runtime, which limits its scalability.

The other body of work focuses on congestion-driven logic optimization and mapping, such as [43], [51], and [55]. In [55], Stok and Kutzchebauch create an initial placement of the technology-independent netlist, and use it for congestion-aware decomposition and technology mapping. Gates are decomposed based on the locations of the drivers of the gate inputs. In technology mapping, gates that are located close to each other are preferred for covering by a library gate during match generation. To reduce congestion, Pandini *et al.* [43] use wirelength as a metric to be minimized during technology mapping. Shelar *et al.* [51] propose a dynamic-programming algorithm to generate probabilistic congestion maps for all matches, which are then used to minimize congestion during the backward phase in mapping. Congestion metric is used in regions with congestion, whereas traditional area or delay metrics are used in sparsely congested regions.

To avoid congestion, congestion-driven optimization and mapping need to be applied preemptively in the design flow before place and route (i.e., before congestion picture becomes available). However, this may lead to area or timing penalty. Applying them postcongestion may require a fresh placement, depending on the amount of netlist change. However, there is no guarantee that the new placement will not lead to congestion. Further research is required to address the problems in both scenarios.

## VI. CONCLUSION

In this paper, we have presented a survey of state-of-the-art technology mapping algorithms for minimum area and delay, as well as postmapping optimizations such as gate resizing, buffering, gate replication, resynthesis and remapping, and pin permutation. Given the suboptimality of mapping algorithms and lack of physical information during mapping, postmapping and postplacement optimizations have assumed great importance and are an integral part of industrial design flows. A qualitative comparison of postmapping optimizations with respect to their relative scope and QoR impact was also shown. Finally, an overview of structural congestion metrics and techniques to avoid or alleviate congestion through logic resynthesis was given. We believe congestion avoidance and alleviation will continue to be a fertile and exciting area of research in the near future. ■

[5]A min-cut for a given pair of vertices $s$ and $t$ is the partition of $V$ into two sets $S$ and $T$ with $s \in S$, $t \in T$ such that the sum of the weights of the edges from $S$ to $T$ is minimum.

## REFERENCES

[1] C. J. Alpert and A. Devgan, "Wire segmenting for improved buffer insertion," in *Proc. Design Autom. Conf.*, 1997, pp. 588–593.

[2] C. J. Alpert, A. Devgan, and S. T. Quay, "Buffer insertion for noise and delay optimization," in *Proc. Design Autom. Conf.*, 1998, pp. 362–367.

[3] C. J. Alpert, A. Devgan, and S. T. Quay, "Buffer insertion with accurate gate and interconnect delay computation," in *Proc. Design Autom. Conf.*, 1999, pp. 479–484.

[4] K. Bartlett, W. Cohen, A. J. De Geus, and G. D. Hachtel, "Synthesis of multi-level logic under timing constraints," *IEEE Trans. Comput.-Aided Design*, vol. CAD-5, no. 4, pp. 582–596, Oct. 1986.

[5] M. Berkelaar and J. Jess, "Gate sizing in MOS digital circuits with linear programming," in *Proc. Eur. Conf. Design Autom.*, 1990, pp. 217–221.

[6] C. L. Berman, J. L. Carter, and K. F. Day, "The fanout problem: From theory to practice," in *Proc. Decennial Caltech Conf. Adv. Res. VLSI*, Mar. 1989, pp. 69–99.

[7] J. R. Burch and D. E. Long, "Efficient boolean function matching," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 1992, pp. 408–411.

[8] R. Carragher *et al.*, "Layout-driven logic optimization," in *Proc. Int. Workshop Logic Synthesis*, 2000, pp. 270–276.

[9] D. Chai and A. Kuehlmann, "Building a better boolean matcher and symmetry detector," in *Proc. Design Autom. Test Eur. Conf.*, 2006, DOI: 10.1109/DATE.2006.243959.

[10] S. Chatterjee, A. Mischenko, R. K. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2005, pp. 2894–2903.

[11] K. Chaudhary and M. Pedram, "Computing the area versus delay trade-off curves in technology mapping," *IEEE Trans. Comput.-Aided Design*, vol. 14, no. 12, pp. 1480–1489, Dec. 1989.

[12] J. Ciric and C. Sechen, "Efficient canonical form for boolean matching of complex functions in large libraries," *IEEE Trans. Comput.-Aided Design*, vol. 22, no. 5, pp. 535–544, May 2003.

[13] J. Cong and Y. Ding, "An optimal technology mapping algorithm for delay optimization in lookup table based FPGA designs," in *Proc. Int. Conf. Comput.-Aided Design*, 1992, pp. 48–53.

[14] O. Coudert, R. Haddad, and S. Manne, "New algorithms for gate sizing: A comparative study," in *Proc. Design Autom. Conf.*, 1996, pp. 734–739.

[15] J. Darringer, D. Brand, J. Gerbi, W. Joyner, and L. Trevillyan, "LSS: A system for production logic synthesis," *IBM J. Res. Develop.*, vol. 28, no. 5, pp. 326–328, Sep. 1984.

[16] D. Debnath and T. Sasao, "Efficient computation of canonical form for boolean matching in large libraries," in *Proc. Asia South Pacific Design Autom. Conf.*, 2004, pp. 591–596.

[17] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Technology mapping in MIS," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 1987, pp. 116–119.

[18] E. M. Sentovich *et al.*, "SIS: A system for sequential circuit synthesis," Electron. Res. Lab., Univ. California, Berkeley, Berkeley, CA, USA, Memo. UCB/ERL M92/41, May 1992.

[19] J. P. Fishburn, "A depth-decreasing heuristic for combinational logic," in *Proc. Design Autom. Conf.*, 1990, pp. 361–364.

[20] J. P. Fishburn, "LATTIS: An iterative speedup heuristic for mapped logic," in *Proc. Design Autom. Conf.*, 1992, pp. 488–491.

[21] J. P. Fishburn and A. E. Dunlop, "TILOS: A posynomial programming approach to transistor sizing," in *Proc. Int. Conf. Comput.-Aided Design*, 1985, pp. 326–328.

[22] M. C. Golumbic, "Combinatorial merging," *IEEE Trans. Comput.*, vol. 25, no. 11, pp. 1164–1167, Nov. 1976.

[23] H. J. Hoover, M. M. Klawe, and N. J. Pippenger, "Bounding fan-out in logical networks," *J. Assoc. Comput. Mach.*, vol. 31, no. 1, pp. 13–18, Jan. 1984.

[24] Z. Huang, L. Wang, Y. Nasikovskly, and A. Mischenko, "Fast boolean matching for small practical functions," in *Proc. Int. Workshop Logic Synthesis*, 2013.

[25] J. Ishikawa *et al.*, "A rule-based reorganization system lores/EX," in *Proc. Int. Conf. Comput. Design*, Oct. 1988, pp. 262–266.

[26] Y. Jiang and S. Sapatnekar, "An integrated algorithm for combined placement and libraryless technology mapping," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 1999, pp. 102–105.

[27] K. Keutzer, "Dagon: Technology binding and local optimization by DAG matching," in *Proc. Design Autom. Conf.*, 1987, pp. 341–347.

[28] P. Kudva, A. Sullivan, and W. Dougherty, "Metrics for structural logic synthesis," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2002, pp. 551–556.

[29] Y. Kukimoto, R. K. Brayton, and P. Sawkar, "Delay-optimal technology mapping by DAG covering," in *Proc. Design Autom. Conf.*, 1998, pp. 348–351.

[30] D. Kung, "A fast fanout optimization algorithm for near-continuous buffer libraries," in *Proc. Design Autom. Conf.*, 1998, pp. 352–355.

[31] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, "Logic decomposition during technology mapping," *IEEE Trans. Comput.-Aided Design*, vol. 16, no. 8, pp. 813–833, Aug. 1997.

[32] W. N. Li, A. Lim, P. Agarwal, and S. Sahni, "On the circuit implementation problem," in *Proc. Design Autom. Conf.*, 1992, pp. 478–483.

[33] J. Lillis, C. K. Cheng, and T. T. Y. Lin, "Optimal wire sizing and buffer insertion for low power and a generalized delay model," in *Proc. Int. Conf. Comput.-Aided Design*, 1995, pp. 138–143.

[34] I. Liu, A. Aziz, and D. F. Wong, "Buffering large networks by Lagrangian relaxation," in *Proc. Int. Workshop Logic Synthesis*, 1999, pp. 6–11.

[35] J. Lou, W. Chen, and M. Pedram, "Concurrent logic restructuring and placement for timing closure," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 1999, pp. 31–35.

[36] J. Lou, A. Salek, and M. Pedram, "An exact solution to simultaneous technology mapping and linear placement problem," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 1997, pp. 671–675.

[37] R. Murgai, "Delay constrained area recovery via layout-driven buffer optimization," in *Proc. Int. Workshop Logic Synthesis*, Jun. 1999, pp. 217–221.

[38] R. Murgai, "On the complexity of minimum-delay gate resizing/technology mapping under load-dependent delay model," in *Proc. Int. Workshop Logic Synthesis*, 1999, pp. 209–211.

[39] R. Murgai, "On the global fanout optimization problem," in *Proc. Int. Conf. Comput.-Aided Design*, 1999, pp. 511–515.

[40] R. Murgai, "Performance optimization under rise and fall parameters," in *Proc. Int. Conf. Comput.-Aided Design*, 1999, pp. 185–190.

[41] T. Okamoto and J. Cong, "Interconnect layout optimization by simultaneous Steiner tree construction and buffer insertion," in *Proc. ACM/SIGDA Phys. Design Workshop*, 1996, pp. 1–6.

[42] L. P. P. P. van Ginneken, "Buffer placement in distributed RC-tree networks for minimum Elmore delay," in *Proc. Int. Symp. Circuits Syst.*, 1990, pp. 865–868.

[43] D. Pandini, L. T. Pileggi, and A. J. Strojwas, "Congestion-aware logic synthesis," in *Proc. Design Autom. Test Eur. Conf.*, 2002, pp. 664–671.

[44] M. Pedram and N. Bhat, "Layout driven technology mapping," in *Proc. Design Autom. Conf.*, 1991, pp. 99–105.

[45] P. Rezvani, A. Ajami, M. Pedram, and H. Savoj, "LEOPARD: A logical effort-based fanout optimizer for area and delay," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 1999, pp. 516–519.

[46] R. Rudell, "Logic synthesis for VLSI design," Ph.D. dissertation, Electr. Eng. Comput. Sci. Dept., Univ. California Berkeley, Berkeley, CA, USA, Apr. 1989, UCB/ERL M89/49.

[47] A. Salek, J. Lou, and M. Pedram, "A DSM design flow: Putting floorplanning, technology mapping and gate placement together," in *Proc. Design Autom. Conf.*, Jun. 1998, pp. 287–290.

[48] A. Salek, J. Lou, and M. Pedram, "A simultaneous routing tree construction and fanout optimization algorithm," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 1998, pp. 625–630.

[49] S. S. Sapatnekar, V. B. Rao, P. M. Vaidya, and S. M. Kang, "An exact solution to the transistor sizing problem for CMOS circuits using convex optimization," *IEEE Trans. Comput.-Aided Design*, vol. 12, no. 11, pp. 1621–1634, Nov. 1993.

[50] H. Savoj, K. Xiang, K. Pan, and A. Domic, "Technology dependent timing optimization," in *Proc. Int. Workshop Logic Synthesis*, 1997.

[51] R. S. Shelar, P. Saxena, X. Wang, and S. S. Sapatnekar, "A near-optimal technology mapping algorithm targeting routing congestion under delay constraints," in *Proc. ACM Int. Symp. Phys. Design*, 2005, pp. 137–144.

[52] K. J. Singh, "Performance optimization of digital circuits," Ph.D. dissertation, Electr. Eng. Comput. Sci. Dept., Univ. California Berkeley, Berkeley, CA, USA, Dec. 1992.

[53] K. J. Singh and A. S. Vincentelli, "A heuristic algorithm for the fanout problem," in *Proc. Design Autom. Conf.*, 1990, pp. 357–360.

[54] A. Srivastava, R. Kastner, and M. Sarrafzadeh, "Timing driven gate duplication: Complexity issues and algorithms," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2000, pp. 447–450.

[55] L. Stok and T. Kutzchebauch, "Congestion aware layout driven logic synthesis," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2001, pp. 216–223.

[56] I. Sutherland and R. Sproull, "The theory of logical effort: Designing for speed on the back of an envelope," in *Proc. Adv. Res. VLSI*, Santa Cruz, CA, USA, 1991, pp. 1–16.

[57] H. Touati, "Performance-oriented technology mapping," Ph.D. dissertation, Electr. Eng. Comput. Sci. Dept., Univ. California Berkeley, Berkeley, CA, USA, Nov. 1990, UCB/ERL M90/109.

[58] X. Yang, M. Wang, and M. Sarrafzadeh, "DRAGON2000: Standard-cell placement tool for large industry circuits," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2000, pp. 260–263.

## ABOUT THE AUTHOR

**Rajeev Murgai** received the B.Tech. degree in electrical engineering from Indian Institute of Technology, Delhi, India, in 1987, winning the President of India Gold Medal for standing first in the class, the M.S. degree in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, PA, USA, in 1989, and the Ph.D. degree in electrical engineering and computer sciences from the University of California at Berkeley, Berkeley, CA, USA, in 1993.

He joined the Advanced CAD Research Department, Fujitsu Laboratories of America, Sunnyvale, CA, USA, in 1994 and became a Research Fellow in 2004. In 2008, he joined Magma Design Automation, where he was V.P. Product Development for Magma's synthesis product, Talus Design. Currently, he is Principal Engineer at Synopsys India Pvt. Ltd., Noida, India, working on their synthesis product. He has coauthored a book *Logic Synthesis for Field-Programmable Logic Arrays* (New York, NY, USA: Springer-Verlag, 1995). His primary research interest is logic synthesis.