Iterative Remapping for Logic Circuits

Luca Benini, *Member, IEEE*, Patrick Vuillod, and Giovanni De Micheli, *Fellow, IEEE*

***Abstract*—This paper presents an aggressive optimization technique targeting combinational logic circuits. Starting from an initial implementation mapped on a given technology library, the network is optimized by finding optimal replacements to clusters of two or more cells at the same time. We leverage a *generalized matching* algorithm that finds symbolically all possible matching assignments of library cells to a multioutput network specified by a Boolean relation and automatically selects the minimum-cost replacement. The remapping technique can be applied to area minimization under delay constraints, power minimization under delay constraints, and unconstrained delay minimization.**

Our remapping tool is based on a fully symbolic algorithm geared toward flexibility and robustness. The tool has been tested on a large set of benchmark circuits. The quality of the results proves the practical relevance of the technique. We obtain sizable improvements in i) speed (6% in average, up to 20.7%), ii) area under speed constraints (13.7% in average, up to 29.5%), and iii) power under speed constraints (22.3% in average, up to 38.1%).

***Index Terms*— Boolean algebra, circuit optimization, circuit synthesis, logic design.**

I. Introduction

LOGIC synthesis and optimization are evolving in response to the challenges of larger designs, tighter constraints, and aggressively scaled submicrometer technologies. The classical two-phases logic-synthesis approach [8] based on technology-independent optimization followed by technology-dependent library binding has been augmented by a third phase, often called remapping [9]. Remapping consists of a set of local transformations applied to a gate-level mapped netlist. Such transformations can leverage accurate back-annotation from placement and routing to direct the optimization effort toward the most critical regions of the netlist. Precisely targeted transformations on mapped netlists are becoming more a need than a choice for meeting design constraints in submicrometer designs where the cost functions employed in the early phases of logic synthesis are increasingly inaccurate.

In this paper, we describe a remapping approach for iterative optimization of combinational logic networks. Multiple-output subnetworks are iteratively selected and optimized by replacing the original implementations with lower cost and functionally compatible subnetworks. Our approach is based on the concept of *generalized matching* (GM) [5], a multiple-output Boolean matching technique that supports concurrent matching of two or more single-output library cells (or of one multiple-output cell) with a multioutput Boolean function. Generalized matching extends the Boolean relation-based approach to the technology-dependent part of the synthesis flow. It is a well-known fact that multiple-output technology-independent logic optimization based on Boolean relations [1] (BR's) is potentially more powerful (but much more computationally intensive) than traditional single-output optimization approaches such as the algebraic [2] or *don't care*-based approach [3], [4]. We propose a BR-based remapping approach that is powerful but also computationally efficient.

We move from the observation that speed is usually the primary concern in logic synthesis. The timing budget for combinational logic is obtained from architectural specification. When speed is the primary objective, two logic-optimization problems have practical relevance: *unconstrained timing optimization* and *optimization of a secondary cost function (area/power) under tight timing constraints*. The solution to the first problem is useful for the designer to test the feasibility of the constraints. If the timing budget is exceeded after unconstrained timing optimization, the designer must redesign or repartition the specification. The second problem is probably the most frequent in practice: the designer wants to obtain the minimum-area or minimum-power implementation that satisfies the timing constraint.

We target the incremental optimization of a *mapped netlist*. Our starting point is a netlist that has already been optimized by traditional synthesis techniques [6] for maximum speed with area recovery. Remapping is applied to either increase speed or reduce area/power without decreasing speed. Moreover, the remapping engine can take full advantage of the presence of multioutput cells (such as full-adders) in the technology library. Such cells are usually suboptimally exploited in traditional logic-synthesis tools.

The main theoretical contribution of this work is the formulation of a fully symbolic algorithm for finding the minimum-cost replacement for a multioutput cluster of cells under tight timing constraints. From the implementation point of view, we have made several efforts to achieve efficiency and robustness, obtaining satisfactory results. We demonstrate the robustness of our approach by reporting results for all largest benchmarks in the MCNC'91 [21] suite. We obtain sizable improvements in i) speed (6% in average, up to 20.7%), ii) area under speed constraints (13.7% in average, up to 29.5%), and iii) power under speed constraints (22.3% in average, up to 38.1%). Moreover, remapping is very effective on larger netlists.

Manuscript received December 10, 1997; revised April 24, 1998. This work was supported by the National Science Foundation under Contract MIP-9421129.

L. Benini is with the Dipartimento di Informatica, Elettronica e Sistemistica, Universita di Bologna, Bologna 40136 Italy.

P. Vuillod was with the Computer Systems Laboratory, Stanford University, Stanford, CA 94305 USA. He is now with Synopsys EPIC Tools Group, Gieres 38610 France.

G. De Micheli is with the Computer Systems Laboratory, Stanford University, Stanford, CA 94305 USA.

Publisher Item Identifier S 0278-0070(98)08483-8.

This paper is organized as follows. In Section II, we provide basic background information and review the formulation of generalized matching as a decision problem. In Section III, we outline the optimization flow and describe the routines for choosing the target regions and computing degrees of freedom for optimization. In Section IV, we introduce the new formulation of generalized matching as a constrained optimization problem, which is the core theoretical contribution of this work. In Section V, we focus on unconstrained speed optimization. Section VI presents experimental results. Conclusions are drawn in Section VII.

II. Background

We assume that the reader is familiar with Boolean functions, discrete functions, binary decision diagram (BDD)-based manipulation of Boolean functions and algebraic decision diagram (ADD)-based manipulation of discrete functions (see [7], [8], [10], and [19] for a review). We denote vectors and matrices in bold, i.e., $\mathbf{x} = [x_1, x_2, \cdots, x_n]^T$. We use the symbols $\forall_x f = f_x \cdot f'_x$ and $\exists_x f = f_x + f'_x$ to designate, respectively, the *consensus* and the *smoothing* of Boolean function f with respect to variable x. Remember that the consensus operation corresponds to universal quantification, while smoothing corresponds to existential quantification. Consensus (smoothing) with respect to an array of variables can be computed by repeated application of single-variable consensus (smoothing).

Consider Boolean functions that model a portion (or cluster) of the circuit. They are called *cluster functions*. We denote by $\mathbf{f} = [f_1, f_2, \cdots, f_n]^T$ a generic multioutput cluster function. We call *pattern function* a combinational function modeling a library cell, and we use g to represent a generic single-output pattern function.

When considering the minimization of multioutput Boolean functions, the degrees of freedom provided by the environment can be expressed by a *Boolean relation* [1]. If we call X the input space and Y the output space, a Boolean relation $\Re$ can be represented by its *characteristic function* $\mathcal{X}$: $X \times Y \rightarrow \{1, 0\}$ such that $\mathcal{X}(\mathbf{x}, \mathbf{y}) = 1$ if and only if $\mathbf{y} \in Y$ is one of the possible outputs of $\Re$ for the input $\mathbf{x} \in X$.

Matching a cluster function with one (or more) pattern functions means finding a way of assigning the inputs of the cluster function to the inputs of the pattern function such that the pattern function becomes a correct implementation of the cluster function. Notice that this requirement is weaker than functional equivalence. A pattern function is a correct implementation if and only if it can replace the cluster function without changing the functionality of the circuit *at the primary outputs*. Although several types of matching have been defined, such as NPN-matching [11], in the following sections we will use the above definition of matching based on replaceability.

A. Generalized Matching

In [5], we introduced the concept of *generalized matching*. Generalized matching extends the Boolean relation-based approach [1], [18] to the technology-dependent part of the synthesis flow. GM has two key advantages with respect to traditional single-output Boolean matching techniques, namely, i) expressing the degrees of freedom for matching with a Boolean relation that is more powerful than *don't cares* [1] and ii) concurrently matching multiple single-output cells. As a result, GM finds matches that *cannot be found* with any traditional Boolean matching technique.

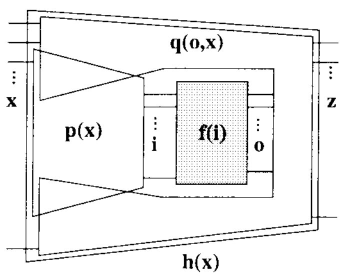

Fig. 1. A multioutput cluster function embedded in its neighborhood.

Let us consider a multioutput cluster function $\mathbf{f}$ with N_i inputs and N_o outputs embedded in a logic network. It is represented in Fig. 1. We adopt a formalism similar to that used by Watanabe *et al.* [18]. We call $\mathbf{x}$ (with dimension N_x) and $\mathbf{z}$ (with dimension N_z) the arrays of Boolean variables at the inputs and the outputs of the network that embeds the cluster function $\mathbf{f}$. The functionality of such network is represented by the Boolean function $\mathbf{h}(\mathbf{x})$. We call it the *neighborhood* of $\mathbf{f}$. The inputs of the cluster function can be seen as a function $\mathbf{p}(\mathbf{x})$ of the inputs $\mathbf{x}$. The function $\mathbf{q}(\mathbf{o}, \mathbf{x})$ describes the behavior of the outputs $\mathbf{z}$ when the outputs of the cluster functions are seen as additional primary inputs.

From $\mathbf{h}$, $\mathbf{p}$, and $\mathbf{q}$, we obtain three characteristic functions

$$H(\mathbf{x}, \mathbf{z}) = \prod_{j=1}^{N_z} h_j(\mathbf{x}) \overline{\oplus} z_j \tag{1}$$

$$P(\mathbf{x}, \mathbf{i}) = \prod_{k=1}^{N_i} p_k(\mathbf{x}) \overline{\oplus} i_k \tag{2}$$

$$Q(\mathbf{o}, \mathbf{x}, \mathbf{z}) = \prod_{j=1}^{N_z} q_j(\mathbf{o}, \mathbf{x}) \overline{\oplus} z_j. \tag{3}$$

These characteristic functions enable the computation of a Boolean relation representing the complete set of *compatible functions* of $\mathbf{f}$, i.e., functions that can implement $\mathbf{f}$ without changing the input–output behavior of $\mathbf{h}$. Watanabe *et al.* showed that the characteristic function $\mathcal{F}$ of the Boolean relation can be obtained with the following formula [18]:

$$\mathcal{F}(\mathbf{i}, \mathbf{o}) = \forall_{\mathbf{x}, \mathbf{z}}[(P(\mathbf{x}, \mathbf{i}) \cdot Q(\mathbf{o}, \mathbf{x}, \mathbf{z})) \Rightarrow H(\mathbf{x}, \mathbf{z})]. \tag{4}$$

Equation (4) allows us to find all functions $\mathbf{f}$ that, when composed with $\mathbf{p}$ and $\mathbf{q}$, produce exactly function $\mathbf{h}$.[1] There are generally many functions with this property. These functions are represented by a Boolean relation, and $\mathcal{F}$ is the characteristic function of such relation.

[1] Roughly speaking, (4) expresses the inclusion of the intersection of the characteristic equations for $p(\mathbf{x})$ and $q(\mathbf{o}, \mathbf{x})$ into the characteristic equation of $h(\mathbf{x})$, enforced for every value of $\mathbf{x}$ and $\mathbf{z}$.

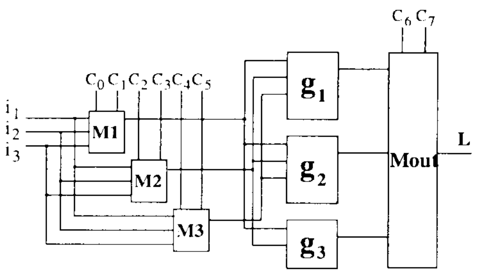

Fig. 2. Quotient function of the target library.

The GM problem consists in finding all possible sets of n library cells that can implement one of the functions represented by $\mathcal{F}$. To accomplish this task, we define the concept of *quotient function* $L(\mathbf{i}, \mathbf{c})$ for our technology library [5]. The pictorial representation of the quotient function is shown in Fig. 2 for a simple library with $N_{lib} = 3$ cells, g_1, g_2, and g_3.

In the figure, the blocks $M1$, $M2$, $M3$, and M_{out} represent virtual multiplexers, with control inputs $\mathbf{c} = [c_0, c_2, \cdots, c_7]^T$. The first three multiplexers control the input pin assignments. By changing the control inputs, we can control how the external inputs are connected to the pins of the cells. Multiplexer M_{out} controls cell selection: it selects which cell is connected to the output. Roughly speaking, the quotient function represents all possible functions that can be implemented by a technology library (refer to [5] for a detailed explanation).

Example 1: Referring to Fig. 2, we consider a simple library containing three cells. A value $\mathbf{c}^*$ of control variables $\mathbf{c}$ uniquely identifies a cell and its input assignment. For instance, $\mathbf{c}^* = [0, 0, 0, 1, 1, 0, 0, 0]^T$ selects cell g_1, with the top-most pin connected to input i_1, the second pin from the top connected to input i_2, and the bottom pin connected to input i_3. Note that input polarity assignments can be represented as well, with the introduction of one control variable for each virtual multiplexer [5].

To perform generalized matching, we need to check if an n-output cluster function $\mathbf{f}(\mathbf{i})$ can be replaced by n library cells. For the sake of simplicity of description, we restrict $n = 2$ even though our method is fully general. Remember that the cluster function and its degrees of freedom are represented by a Boolean relation $\mathcal{F}(\mathbf{i}, \mathbf{o})$. We can express GM with a Boolean formula [5]

$$M(\mathbf{c}) = \forall_{\mathbf{i}}\, \exists_{\mathbf{o}}(\mathcal{F}(\mathbf{i}, \mathbf{o}) \cdot (L(\mathbf{i}, \mathbf{c}_1) \overline{\oplus} o_1)(L(\mathbf{i}, \mathbf{c}_2) \overline{\oplus} o_2)). \quad (5)$$

Where $\mathcal{F}$ is the Boolean relation for the cluster, L is the quotient function. Notice that for each output, we have a different quotient function with distinct sets of control variables, and hence $\mathbf{c} = [\mathbf{c}_1, \mathbf{c}_2]$. This is because each output of $\mathbf{f}$ can be matched by a different cell with different input assignments. $M(\mathbf{c})$ is called the *matching function* and can in principle be computed by simply implementing (5) with standard BDD operators. The ON-set of $M(\mathbf{c})$ denotes all possible assignments of the cluster to two library cells with the property that the new implementation of the cluster function can replace the old one without changing the behavior observed at the output of $\mathbf{h}$. In other words, (5) allows us to compute *all* cell selections and input assignments compatible with $\mathcal{F}$ [5].

Unfortunately, the generality of (5) has a cost in terms of computational complexity. In practice, only very small instances of GM can be solved in a reasonable time by a straightforward implementation of the computations expressed by (5). Several theoretical insights and algorithmic optimizations will be introduced in the following sections with the main purpose of extending the practical usefulness of GM. Roughly speaking, two strategies will be exploited to manage complexity, namely, decomposition and bounding. With decomposition, the solution of a complete GM instance will be split in several simpler instances. The final solution can be computed as the intersection of the solutions for the simpler instances. With bounding, the search space for candidate solutions will be restricted by rapidly eliminating regions that cannot contain optimal solutions.

Concluding the section, notice that both traditional Boolean matching and generalized matching have been formulated as decision problems, where the solution consists of finding a *yes* or *no* answer. Such decision problems arise sometimes in practice, for example, when checking the equivalence of two logic circuits with unknown input assignment [12]. However, variations of the matching problem, namely, *minimum-cost matching* and *minimum-cost constrained matching* have much wider practical relevance.

III. The Remapping Approach

In the recent past, numerous logic-synthesis tools have been developed in academia and industry. Most implementations follow a two-phase approach. In the first phase, *technology-independent optimizations* are performed: the initial description (written in a generic hardware description language) is optimized using transformations and cost functions that do not depend on the particular technology library chosen for the final implementation. Then, in the second phase, technology-dependent optimizations are applied as the generic logic description is mapped to the technology library. This step is often called *library binding*.

Recently, the two-phase synthesis flow has been augmented by a third phase. Several algorithms have been developed that operate on a mapped netlist and attempt to further optimize it [13]–[15]. We call *remapping* the postprocessing step. Some remapping approaches [16] focus on changing the connectivity of the netlist in such a way that some gates either become redundant (and can be removed) or become suboptimal (and can be replaced). Remapping transformations based on changes of the network connectivity are often called *rewiring*.

We adopt a remapping approach. Starting from an optimized and mapped netlist, we apply our optimization engine to specific regions of the mapped netlist where local improvements are more likely. The high-level flow of the remapping procedure is shown in Fig. 3. First, the initial mapped network is analyzed. Power dissipation, arrival times, required times, and slacks are computed for all nodes. This information drives the selection of the target regions for remapping. Clusters of cells in the target regions are constructed and remapping is attempted. The degrees of freedom extracted by examining the portion of the logic network around the

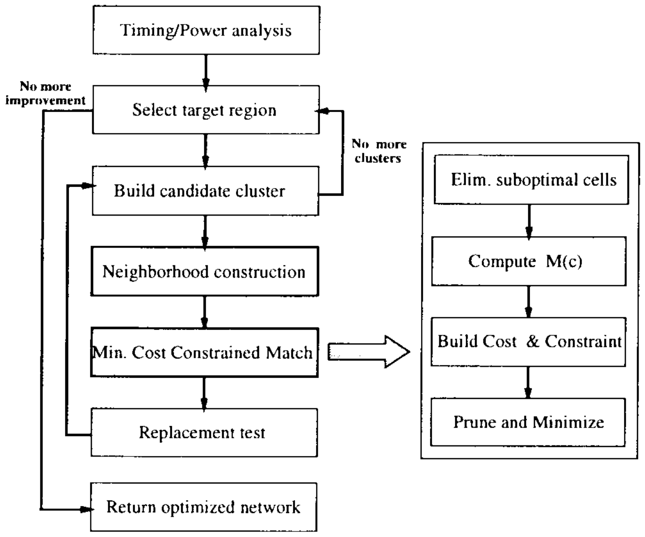

Fig. 3. High-level flow of the remapping procedure.

```
ReMapMFP (Network)
  foreach (node ∈ BackBFirstTrav (Network)) {
    if (node is MFP) {
      fo_list = fanout (node);
      foreach (node_1, node_2 ∈ fo_list) {
        clusterF = [node_1, node_2];
        if (best_match = ComputeBRelOptimize (clusterF, Network)) {
          replace (clusterF, best_match, Network);
        }
      }
      foreach (node_1 ∈ fo_list) {
        clusterF = [node_1, node];
        if (best_match = ComputeBRelOptimize (clusterF, Network))
          replace (clusterF, best_match, Network);
      }
    }
  }
```

Fig. 4. Sliding window algorithm for cluster function selection.

cluster (i.e., the neighborhood) are exploited. Cluster selection and neighborhood construction are described in Sections III-A and III-B, respectively. The core remapping task is based on an efficient implementation of minimum-cost-constrained generalized matching, which is described in Section IV. The procedure is iterated until convergence.

The choice of the remapping approach is dictated by practical reasons. Current technology mapping tools are robust and efficient. We leverage their capabilities to obtain an initial optimized implementation, then we apply our powerful but computationally demanding optimization engine to portions of the circuit where traditional mapping algorithms do not produce optimal results. The advantages of this choice are: i) it merges seamlessly with preexisting tools and design flows and ii) it allows us to put more effort in local optimizations, since we perform a reduced number of them. The main drawback is that our technique performs only incremental improvements; thus, if the starting point is a local optimum very far from the global optimum, we may not be able to move out of it.

A good choice of the target regions in the mapped netlist is paramount for the success of the remapping strategy. Different choice criteria are applied depending on the nature of the cost function that we want to optimize. Power and area are *extensive* cost functions, i.e., they depend on the entire circuit, while speed is an *intensive* cost function, i.e., it is determined only by a critical portion of the circuit (the slowest paths). When we optimize an extensive cost measure, we want to distribute the optimization effort on the entire netlist, while the optimization of an intensive cost function can be better achieved by focusing only on the critical portion.

A. Target-Region Selection

In this subsection, we describe a target-selection strategy tailored for optimization of extensive cost functions (area and power). Our approach to the optimization of intensive cost functions (speed) is described in Section V. We focus on *multiple fanout points* (MFP's). There are two main reasons for this choice. First, traditional library binding algorithms follow a *cone-based* paradigm [17]. A very efficient search of the optimum mapping is performed on fanout-free regions of the circuit, but the search stops when MFP's are reached. As a result, the final implementation consists of highly optimized fanout-free regions connected by multiple fanout points. Roughly speaking, we target the loss of optimality caused by the interruption of the cone-based search when a MFP is reached.

Second, since our optimization strategy is based on the computation of a Boolean relation expressing the degrees of freedom for the implementation of a multiple-output subnetwork, we are more likely to find degrees of freedom when the output functions of the subnetwork share some support variables. This is generally true when two or more gates driven by a MFP are considered as candidates for optimization.

The enumeration of the MFP's is done by traversing the network in a backward *breadth-first* fashion starting from the output and moving toward the inputs. The pseudocode of the algorithm for selection of candidate networks for remapping is shown in Fig. 4. Several corner cases and limit conditions are not shown for the sake of simplicity.

The outermost loop implements the backward breadth-first traversal. Whenever a MFP is reached, its fanout gates are inserted in list `fo_list`. The first inner loop selects multioutput clusters consisting of set of elements in `fo_list`. The second inner loop selects clusters that include the gate with multiple fanout and one or more of its fanout gates. In the simplified pseudocode, two-output clusters are selected. The actual implementation can generate multioutput clusters with any number of outputs. In practice, three outputs is usually the maximum for efficiency reasons because the complexity of the computations involved in matching rapidly increases with the number of cluster outputs, and the number of clusters that can be generated is $O(|FO|^{n_{\text{out}}})$ (where n_{out} is the number of cluster outputs and $|FO|$ is the number of fanout stems).

For each candidate cluster generated by the internal loops, the function `ComputeBRelOptimize` is called. It computes minimum-cost constrained matching and is the core procedure in the algorithm. It will be analyzed in the following section. It returns `best_match` if a match has been found that improves the cost and satisfies the constraints. If this is the case, the original network is modified accordingly by function `replace`.

Procedure `ReMapMFP` is usually applied several times to the target network. A single pass is not enough for getting the best results. For instance, when we operate a replacement, we may create new multiple fanout points. Additionally, incremental modifications of the network may cause changes in the degrees of freedom available in the neighborhood of the optimization point and enable further improvement. The main reason for performing a backward traversal is that replacements can create new MFP's on the inputs of the clusters being replaced. Moving backward, we traverse the newly created MFP's in the same pass, and we accelerate convergence (*convergence* is reached when a call of `ReMapMFP` does not decrease the cost function).

In practice, convergence is reached in two or three passes. Even if convergence is rapid, we implemented several optimizations to reduce the run time in successive iterations. For instance, MFP's whose neighborhood is not changed in iteration n are skipped in iteration $n+1$. The formal definition of neighborhood will be given in the next subsection.

The algorithm in Fig. 4 constructs candidate clusters starting from MFP's. Once a cluster is created, the function `ComputeBRelOptimize` is called. The two main tasks of the function are the following.

- Build the Boolean relation that represents the degrees of freedom for matching created by the cluster's neighborhood.
- Perform minimum-cost matching and guarantee that timing constraints are not violated.

In the next subsection, we describe how the first task is carried out, while the second task is described in Section IV.

B. Building the Boolean Relation

Boolean relation $\mathcal{F}$ can be computed using (4) once the neighborhood $\mathbf{h}$ is specified. Ideally, we would like to compute $\mathcal{F}$ by considering as $\mathbf{h}$ the entire logic network, from primary inputs to primary outputs. This choice would give us the maximum degrees of freedom for the implementation of the cluster function [18]. Unfortunately, this is computationally infeasible except for the smallest networks. Thus, the neighborhood has to be a small subset of the logic network, like a "bubble" around the cluster function. Notice that GM relies on finding a Boolean relation that gives the most degrees of freedom to the chosen cluster. Intuitively, we want to establish a relation between the outputs of `f` that gives more degrees of freedom than computing separately their *don't cares*.

We consider two-output cluster functions for the sake of explanation. The neighborhood construction algorithm is shown in Fig. 4. Our purpose is to compute Boolean relations expressing many degrees of freedom. Thus, we look for nodes in the fanout cone and fanin cone of both outputs of `f`. Intuitively, a common fanout node within the neighborhood is an indication that functionality at the neighborhood outputs is controlled by the interaction of both outputs. Similarly, a common fanin node implies that there is some sharing of information among the inputs of `f`. If fanin and fanout cones of the components of `f` are disjoint, $\mathcal{F}$ represents the same degrees of freedom that can be expressed by *don't cares*.

Since we consider clusters starting from MFP's, at least one common fanin exists. To build the neighborhood, we explore the transitive fanout and fanin of the cluster. We control complexity by limiting the search to a given depth. The algorithm generating the neighborhood is shown in Fig. 5. Its inputs are the cluster function `f` (`clusterF` in the pseudocode) and the maximum depth of exploration (parameter `depth`). We call o_1 and o_2 the outputs of `clusterF`. First, we mark the fanout cones of o_1 and o_2. We take the intersection of the fanout cones (set `intersec`) and get the paths from o_1 and o_2 to nodes in `intersec`. All nodes in such paths are collected in `pathnodes`. From `pathnodes`, we create a new set, `outnodes`, by extending `pathnodes` with their fanout nodes. They correspond to the part of the neighborhood in the fanout of the cluster. A similar procedure is applied to the fanin cones, and the set `innodes` is produced. They are the fanin part of the neighborhood.

The *external* nodes of `outnodes` are picked up to produce `xnodes` (input nodes) and `znodes` (output nodes). The input nodes of `innodes` are put in `xnodes`. We do not need the output nodes of `innodes` because they are not affected by any change of the cluster. The neighborhood is defined as the union of `xnodes` and `znodes`. The rationale of this algorithm is to include in the neighborhood the maximum number of reconvergence regions containing `f` (constrained by the depth of the exploration). Including such regions in the neighborhood increases the probability that $\mathcal{F}$ expresses degrees of freedom that cannot be captured by *don't cares*. It may be observed that the same result is achieved by a straightforward algorithm that computes the neighborhood by simply traversing the transitive fanin and fanout of `f` with depth `depth`. Experimentally, we observed that the straightforward approach is not practical because the neighborhood gets very large even for small depths and the computation for $\mathcal{F}$ becomes too expensive.

Example 2: We show on Fig. 6 how the algorithm works. The picture represents a portion of a logic network, the vertices being logic gates and the arrows the connections between them. We start from a two-node cluster, marked in black in the top-left part of the Fig. 6. The parameter `depth` is set to two. To build the neighborhood, we first select the reconvergent nodes in the transitive fanout and fanin of the clusters (with depth 2) from the cluster. These nodes are marked in black on the top right. The nodes on paths connecting the cluster with reconvergent nodes are marked in black on the bottom left. Finally, we take the "envelope" of these nodes to get the neighborhood. The neighborhood is the set of nodes marked in black in the bottom-right part of Fig. 6.

Given the neighborhood, the Boolean relation $\mathcal{F}$ is obtained by (4). We build the BDD's of the Boolean relations P, H, and Q by traversing the neighborhood. We apply the corresponding BDD's operators and universal quantification to compute $\mathcal{F}$. Note that $\mathcal{F}$ depends only on a few BDD variables, namely, the variables for the inputs and the outputs of the cluster; therefore, the resulting BDD of the Boolean relation $\mathcal{F}$ is very small. However, the overall complexity depends on the computation of the relation H. To build the relation H, we need to build the BDD's of each neighborhood output z_i with respect to the inputs and compute their conjunction. This operation can be

```
ComputeNeighborhood (clusterF, depth)
  /* 1. Moving forwards for building outnodes */
  coneout1 = make_fanout_cone ($o_1$, depth);
  coneout2 = make_fanout_cone ($o_2$, depth);
  intersec = get_intersec (coneout1, coneout2);
  /* pathnodes are the nodes on the reconvergent paths */
  pathnodes = NIL;
  foreach (node $\in$ intersec) pathnodes = pathnodes $\cup$ path ($o_1$, node) $\cup$ path ($o_2$, node);
  /* outnodes are the envelope of path nodes */
  outnodes = pathnodes;
  foreach (node $\in$ pathnodes) outnodes = outnodes $\cup$ (fanout (node));
  /* Take only the external nodes of outnodes */
  znodes = NIL;
  foreach (node $\in$ outnodes)
    if (fanout (nodes) $\notin$ outnodes)
      znodes = znodes $\cup$ node;

  /* 2. Move backwards for building innodes */
  conein1 = make_fanin_cone ($o_1$, depth);
  conein2 = make_fanin_cone ($o_2$, depth);
  intersec = get_intersec (conein1, conein2);
  / * pathnodes are the nodes on the reconvergent paths */
  pathnodes = NIL;
  foreach (node $\in$ intersec) pathnodes = pathnodes $\cup$ path ($o_1$, node) $\cup$ path ($o_2$, node);
  /* innodes are the envelope of path nodes */
  innodes = pathnodes;
  foreach (node $\in$ pathnodes) innodes = innodes $\cup$ (fanin (node));
  /* Take only the external nodes of innodes and outnodes */
  xnodes = NIL;
  foreach (node $\in$ (innodes $\cup$ outnodes))
    if (fanin (node) $\notin$ (innodes $\cup$ outnodes))
      xnodes = xnodes $\cup$ node;
  neighborhood = (xnodes, znodes);
  return (neighborhood);
```

Fig. 5. Algorithm for neighborhood construction.

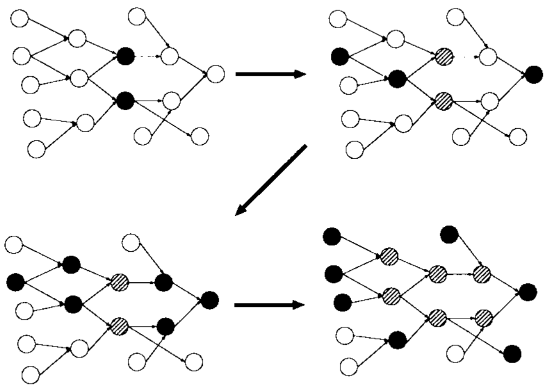

Fig. 6. Building the neighborhood of a cluster.

computationally infeasible if $\mathbf{z}$ are primary outputs and $\mathbf{x}$ are primary inputs. The algorithm of Fig. 5 has been designed to minimize the complexity of the computation of H, and at the same time to obtain a final $\mathcal{F}$ expressing useful degrees of freedom.

IV. Minimum-Cost Constrained GM

Generalized matching is formulated and solved in Section II-A as a decision problem. In this section, we first describe an efficient algorithm for the computation of the matching function $M(\mathbf{c})$. We introduce a *pruning* procedure for speeding up the computations involved in (5). Then, we extend GM from a simple decision algorithm to a complete constrained symbolic optimization procedure. We describe the symbolic computation of cost functions and constraints, and we introduce a *bounding* procedure for further improving the efficiency of symbolic optimization.

A. Pruning

The main practical problem in the computation of $M(\mathbf{c})$ by (5) is that, although the BDD representation of $M(\mathbf{c})$ is generally very compact, the same is not true for the intermediate results of the computation in (5). Experimentally, we observed that BDD blowup was very common while computing the conjunction of $\mathcal{F}$ with the quotient functions and while computing the quantifications. We can express the final result, but there is a peak BDD size to overcome. To avoid going up to this peak, we partition the problem. We compute the matching function for each output separately, and use the partitioned solutions to reduce the size of the BDD's in (5) before universal quantification. Notice that the procedure *does not* compromise the global optimality of the final solution. This claim will be clarified in the following discussion.

Again, we discuss the case of two outputs for the sake of simplicity (although the approach is general). The matching function of output o_1 can be computed with the following formula:

$$M_1(\mathbf{c}_1) = \forall_{\mathbf{i}}\, \exists_{\mathbf{o}} (\mathcal{F}(\mathbf{i}, \mathbf{o}) \cdot (L(\mathbf{i}, \mathbf{c}_1) \overline{\oplus} o_1)). \tag{6}$$

The same formula holds for output o_2 (changing indexes from 1 to 2). Computing M_1 and M_2 separately can be much easier than computing M because the BDD's have fewer support variables, and only one conjunction has to be computed before quantification. This observation is confirmed in practice. The computation of M_1 and M_2 requires much less memory than the computation of M.

It is easy to see that $M_1(\mathbf{c})$ and $M_2(\mathbf{c})$ are less constrained than $M(\mathbf{c})$: $M_1 \geq M$ and $M_2 \geq M$. $M_1(\mathbf{c})$ expresses all possible matches for output o_1, assuming that o_2 can be implemented by an arbitrary function of the inputs. In general, the ON-set of M_1 (M_2) contains solutions that are not valid: all matches of f_1 (f_2) that are admissible only when f_2 (f_1) is not representable by $L(\mathbf{i}, \mathbf{c}_2)$ ($L(\mathbf{i}, \mathbf{c}_1)$) belong to the ON-set of M_1 (M_2) but are in the OFF-set of M. We can use the M_1 and M_2 as conservative bounds for pruning the search space of $M(\mathbf{c})$ because we know that if a value $\mathbf{c}^*$ is not in the ON-set of *both* M_1 *and* M_2, it will be in the OFF-set of M, and we do not need to take it into account when matching M.

The simplest way to exploit this property is to compute the *restriction* [10] of $L(\mathbf{i}, \mathbf{c}_1)$ and $L(\mathbf{i}, \mathbf{c}_2)$ with respect to $M_1(\mathbf{c}_1)$ and $M_2(\mathbf{c}_2)$, respectively, and then compute M with (5). In other words, we can replace $L(\mathbf{i}, \mathbf{c}_1)$ in (5) with $L_{res}(\mathbf{i}, \mathbf{c}_1)$ defined as follows:

$$L_{res}(\mathbf{i}, \mathbf{c}_1) = \begin{cases} L(\mathbf{i}, \mathbf{c}_1), & \text{if } M_1(c_1) = 1 \\ \text{don't care}, & \text{otherwise.} \end{cases} \tag{7}$$

The same can be done for $L(\mathbf{i}, \mathbf{c}_2)$. In practice, we augmented the basic algorithm by introducing *reencoding* of the control variables $\mathbf{c}$ with a reduced set of new control variables $\boldsymbol{\gamma}$. The number of new control variables is equal to $\lceil \log_2 N_{mint,1} \rceil$, where $N_{mint,1}$ ($N_{mint,2}$) is the number of minterms in the ON-set of M_1 (M_2). With the new encoding[2] $L_{res}(\mathbf{i}, \boldsymbol{\gamma}_1)$ and $L_{res}(\mathbf{i}, \boldsymbol{\gamma}_2)$ have typically a much more compact BDD representation, and (5) can be efficiently computed. We do not describe the reencoding algorithm in detail because it is quite complex and not essential for the understanding of the complete algorithm.

By computing M_1 and M_2 we prune the search space, and the computation of $M(\mathbf{c})$ is much faster. Notice that we do not make any approximation here. $M(\mathbf{c})$ still gives us all possible assignments for the cluster. The bound is conservative, and the matching function is computed exactly. This method can be used for more than two input clusters. In practice, the method breaks down when (6) cannot be computed with the available memory resources.

B. Cost Function and Constraints

Although we have a way to compute all possible legal replacements for $\mathbf{f}$, we want get the minimum-cost matches satisfying the timing constraints. Hence, we need to apply a cost function to $M(\mathbf{c})$ and find at least one assignment $\mathbf{c}^*$ minimizing it. Moreover, we need to enforce the satisfaction of the constraints.

The most straightforward approach is the explicit enumeration of the ON-set of $M(\mathbf{c})$. For each minterm (that corresponds to a valid replacement), the cost function (power or area) is computed and the timing constraints are checked. The minimum-cost solution satisfying the constraints is then selected. Unfortunately, the enumerative approach is excessively slow. Generally, $M(\mathbf{c})$ has a large number of minterms whose enumeration would take an unreasonable amount of time.

To overcome this limitation, we solve the minimum-cost constrained GM problem in a symbolic fashion. We employ ADD's [19] to build an abstract representation of cost functions and constraints. ADD's are appropriate because they represent discrete functions in a compact way, and they interface seamlessly with BDD's (they have the same structure, the only difference being that leaves can have any value).

[2] The cardinality of the ON-set of M_1 (M_2) is generally much smaller than the total number of values of $\mathbf{c}$; thus, the number of variables needed to encode it is generally smaller than the number of control variables $\mathbf{c}$.

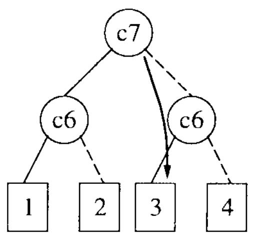

Fig. 7. ADD cost function for area for four variables.

To compute minimum-cost matching, we build the ADD for the cost function with the same support variables as the BDD for $M(\mathbf{c})$. A path in the ADD leads to a leaf containing the cost of the cell identified by the values of control variables encountered on the path. Once the ADD of the cost function is built, we can compute the *product* with the BDD of the matching function and select the minterm pointing to the minimum value of it (product and minimum selection are standard ADD operators). A similar line of reasoning holds for constraints, with the only difference that the constraint ADD is used to prune from the ON-set of M all solutions that violate the constraint.

1) Area Cost: Since the area of a cell is not affected by the input assignment, the support of the ADD representing the area cost, $A(\mathbf{c})$, contains only variables controlling the cell selection in the quotient function. The cost function for area can be computed once and for all. Its number of nodes is very small, bounded by $2 \times |N_{lib}|$ (if no two cells have the same area, the ADD reduces to a binary tree). All ADD operators involved in the construction of the symbolic representation of the cost function and its minimization over the ON-set of M have complexity $O(|M(\mathbf{c})| \cdot |A(\mathbf{c})|)$. The number of nodes in the BDD of the matching function is $|M(\mathbf{c})|$, while $|A(\mathbf{c})|$ is the number of nodes in the ADD of the cost function. Since usually both $|M(\mathbf{c})|$ and $|A(\mathbf{c})|$ are small [at most on the order of 100 BDD nodes for $|M(\mathbf{c})|$], the computation of the area cost is very fast compared to an enumeration of the minterms of M.

Example 3: Consider a library $L = \{$NAND2, AND2, NAND3, AND3$\}$ with area costs, respectively, $\{1, 2, 3, 4\}$. The ADD $A(\mathbf{c})$ of the area cost function for this library is represented in Fig. 7. The ELSE edges are represented by dashed arrows, while THEN edges are represented by solid arrows. The control variables for cell selection are c_7 and c_6. For instance, $c_6\overline{c}_7$ selects the NAND3 gate. In the ADD of the cost function, we see how the path with c_6 and $\overline{c}_7$ leads to the cost of the NAND3, i.e., three. Assume that the matching function is $M(\mathbf{c}) = c_0c_1c_2c_3'c_4'c_5c_7'$. Taking the minimum of the product $A(\mathbf{c}) \cdot M(\mathbf{c})$, we obtain the value three, and the value $\mathbf{c}^*$ of the control variables for which $A(\mathbf{c}) \cdot M(\mathbf{c})$ is minimized is $\mathbf{c}^* = c_0c_1c_2c_3'c_4'c_5c_6c_7'$.

2) Power Cost: The computation of the power ADD is much more involved than that of the area ADD because power consumption depends on both cell assignment and input assignment. In this work, we consider only the power due to output switching activity (also known as *external power* or *switching power*). This is motivated by two facts. First, our model of the power consumed within the cells (*internal power*) is quite complex, and it would unnecessarily complicate the explanation of the basic technique. Second, there is no agreement in the literature on what the best internal power model is. For instance, ours is completely different from the one presented in [22]. On the contrary, there is an agreement on how to compute the external power. Moreover, switching power is usually the most important contribution.

External power at a node o of a Boolean network is given by

$$\mathcal{P}_{ext}(o) = KP_t(o)C(o)\,\frac{V_{dd}^2}{2} \tag{8}$$

where $P_t(o)$ is the transition probability of node o, $C(o)$ is the capacitance at node o, V_{dd} is the supply voltage, and K is a constant factor.

Consider a cluster with two outputs. The two outputs are bound by the Boolean relation $\mathcal{F}$. A target library function can be placed at one of the outputs o if it satisfies $\mathcal{F}(\mathbf{i}, \mathbf{o})$. The same holds for the other output. We want to analyze the gain in power of the replacement at one of the outputs o of the cluster. External power can change at the node o because the transition probability $P_t(o)$ depends on the function at o. Observe that the variation is possible because, by exploiting the degrees of freedom expressed by $\mathcal{F}$, we may modify the function at o, and consequently the transition probability. The capacitance $C(o)$ is independent of the function at output o because it depends only on the fanouts of o, which are not modified by matching.

The computation of the transition probability is done symbolically as follows. Output node o is represented by the function $o = f(\mathbf{i})$. We call $\mathbf{i}$ and $\mathbf{i}^+$ two consecutive input patterns, their response at the output being $o = f(\mathbf{i})$ and $o^+ = f(\mathbf{i}^+)$. The transition probability of o, $P_t(o)$, is the probability that o switches; therefore, it is $P(o \neq o^+)$. Consider now two consecutive input patterns $\mathbf{i}$ and $\mathbf{i}^+$. If the responses to these patterns o and o^+ are different, we observe a transition at the outputs. This event contributes to $P(o \neq o^+)$. It happens with a probability $P(\mathbf{i}, \mathbf{i}^+)$. The formula for the transition probability is the sum of all such events at the inputs. We obtain the following formula:

$$P(o \neq o^+) = \exists_{\mathbf{i}, \mathbf{i}^+}(f(\mathbf{i}) \oplus f(\mathbf{i}^+))P(\mathbf{i}, \mathbf{i}^+). \tag{9}$$

If we assume spatial independence of the inputs and we neglect high-order temporal correlations (i.e., correlations between patterns with more than one cycle-time difference), the probability of two consecutive input vectors $P(\mathbf{i}, \mathbf{i}^+)$ is the product of the probability of each bit sequence of the vector

$$P(\mathbf{i}, \mathbf{i}^+) = \prod_{i \,\in\, \text{Inputs}} p(i, i^+) \tag{10}$$

where

$$p(i, i^+) = ii^+p_{11} + i\overline{i^+}p_{10} + \bar{i}i^+p_{01} + \bar{i}\overline{i^+}p_{00} \tag{11}$$

and where p_{ab} is the probability of the sequence ab. These values are expressed below with the static probability P_s of the signal i and the transition probability P_t:

$$p_{10} = \frac{P_t}{2} \qquad p_{11} = P_s - \frac{P_t}{2}$$
$$p_{01} = \frac{P_t}{2} \qquad p_{00} = 1 - P_s - \frac{P_t}{2}.$$

	$i_1^+ i_2^+$			
$i_1 i_2$	00	01	11	10
00	00	00	01	00
01	00	00	01	00
11	10	10	11	10
10	00	00	01	00

(a)

	$i_1^+ i_2^+$			
$i_1 i_2$	00	01	11	10
00	.15	.15	.02	.02
01	.15	.15	.02	.02
11	.02	.02	.05	.05
10	.02	.02	.05	.05

(b)

Fig. 8. (a) Table of the transitions for a two-input AND. (b) Table of the probabilities of each transition pair.

Example 4: Consider a two-input AND gate. The truth table in Fig. 8(a) shows the pairs of values (o, o^+) for all the pairs (i, i^+). The rows represent one value of the inputs $(i_1 i_2)$, and the columns a following value $(i_1^+ i_2^+)$. For example, if we apply the pattern 00 and then 11 to the AND gate, the response is 01. In this example, the input static probabilities P_s are (0.3, 0.5), and transition probabilities P_t are (0.2, 0.5). The probabilities for (i_1, i_1^+) are

$$p_{10} = 0.2/2 = 0.1 \qquad p_{11} = 0.3 - 0.2/2 = 0.2$$
$$p_{01} = 0.2/2 = 0.1 \qquad p_{00} = 1 - 0.3 - 0.2/2 = 0.6.$$

In the same way, we obtain, for (i_2, i_2^+), $p_{10} = p_{11} = p_{01} = p_{00} = 0.25$.

In Fig. 8(b), we have represented the matrix of the probabilities of two consecutive input patterns. The indexing of the matrix is the same as the truth table; the rows represent $\mathbf{i}$ and the columns $\mathbf{i}^+$. We see that, for example, the probability of the pair of inputs (01, 10) is 0.02.

In the truth table [Fig. 8(a)], we see six points where o switches. To obtain the transition probability, we sum these six points weighted by the matrix [Fig. 8(b)]: $P_t(o) = 0.025 + 0.025 + 0.025 + 0.025 + 0.05 + 0.05 = 0.2$.

The consequence of the transition probability change is the modification of power consumption at o and at the fanout nodes of o. If the transition probability changes at o, its fanout nodes will have a new transition probability. However, we know from Section II-A that the output nodes $\mathbf{z}$ of the neighborhood have the same behavior regardless of the match. Therefore, they have a constant transition probability and, consequently, all the nodes in their fanout have a constant probability for this given match. So the impact in the fanout nodes does not go further than $\mathbf{z}$. The impact of the change on the fanout nodes has to be taken in consideration, but it has only a limited scope in the circuit, namely, the nodes inside the neighborhood. This issue is discussed in the Appendix. For now, we assume that changes in transition activity of o do not sensibly affect the transition activity even for nodes within the neighborhood.

We analyzed the impact of a replacement at the outputs of the cluster. However, external power is modified at the inputs

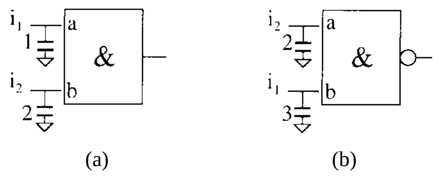

Fig. 9. (a) Assignment of $\{i_1, i_2\}$ with AND2 and (b) assignment of $\{i_2, i_1\}$ with NAND2.

as well. Consider i, one of the inputs $\mathbf{i}$. The power variation at the input is not due to the transition probability because this one may change only at the gate output. It is caused instead by the variations of the load $C(i)$ at the input when we change cell and pin assignment during remapping. The load at input i can be expressed as

$$C(i) = C_i^{\text{other}} + C_i^1 + C_i^2 \tag{12}$$

where C_i^1 (C_i^2) is the input capacitance of the cell connected to o_1 (o_2) and to input i. [If i is not connected to the cell at o_1 (o_2), $C_i^1 = 0$ $(C_i^2 = 0)$.] C_i^{other} is a constant, but C_i^1 and C_i^2 depend on the cell selection and the pin assignments.

Example 5: Consider a single output cluster o and two candidate matches. The input probabilities of $\{i_1, i_2\}$ are, respectively, $P_s = (0.3, 0.5)$ and $P_t = (0.2, 0.5)$, as in Example 4. The first match connects o with an AND2 gate to i_1 and i_2. The input capacitances are one and two, as shown in Fig. 9(a) The input power of this choice is $\mathcal{P}_{\text{ext}}(i) = 0.2 * 1 + 0.5 * 2 = 1.2$.

The second choice connects o with a NAND2 gate to i_2 and i_1. The input capacitances are two and three, as shown in Fig. 9(b). The input power of this choice is $\mathcal{P}_{\text{ext}}(i) = 0.2 * 3 + 0.5 * 2 = 1.6$.

Obviously, the first match is better than the second one. The cost function will tend to find the lower cost pin permutation for a given cell and find the cells that have a low input capacitance.

To evaluate the power cost, we need to compute the input capacitance and the output transition probability for each cell and pin assignment. Power cost is a function of variables $\mathbf{c}$. For each minterm of the Boolean space of $\mathbf{c}$, we compute a power value. As we have shown before, we need to find the output transition probability for any cell configuration and the capacitance at each input for any input assignment.

For a particular value of the control variables $\mathbf{c} = \mathbf{c}^*$, the restriction of the quotient function $L(\mathbf{i}, \mathbf{c}^*)$ represents a Boolean function of the inputs $\mathbf{i}$ alone: it is the function implemented by the library cell and the input pin assignment expressed by $\mathbf{c}^*$. For such a function, we can compute P_t. Thus, we can compute P_t [as defined in (8)] for any possible value of $\mathbf{c}$ in a symbolic fashion using ADD's

$$P_t(\mathbf{c}, o) = \exists_{\mathbf{i}, \mathbf{i}^+} \big((L(\mathbf{i}, \mathbf{c})) \oplus L(\mathbf{i}^+, \mathbf{c})\big) P(\mathbf{i}, \mathbf{i}^+) \tag{13}$$

with $P(\mathbf{i}, \mathbf{i}^+)$ being the ADD computing the probabilities of $(\mathbf{i}, \mathbf{i}^+)$ and L the BDD of the quotient function. $P_t(\mathbf{c}, o)$ represents how the transition probability of output o changes as a function of the control variables (that select different implementations for o).

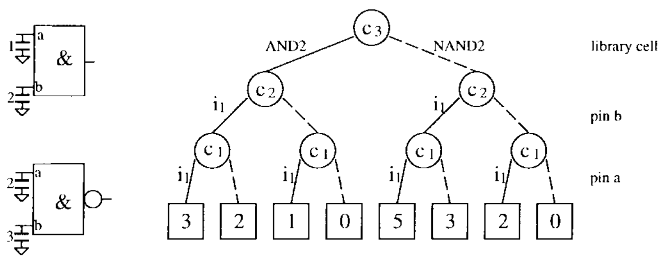

Fig. 10. ADD of the capacitance at i_1 for any selection.

The second part of the cost function is to compute the capacitance. The ADD formulation has the same form as (12), but C_i^1 and C_i^2 are ADD's function of $\mathbf{c}$. The ADD of the capacitance is then $C_i(\mathbf{c}) = C_i^1(\mathbf{c}) + C_i^2(\mathbf{c}) + C_i^{\text{other}}$. To compute $C_i(\mathbf{c})$, we need to build an ADD with the leaves containing the input capacitances of the pin selected by each ADD path.

Example 6: Consider again the cells AND2 and NAND2 of Example 5. To simplify, we consider a cluster with one output and only two inputs $\{i_1, i_2\}$. We need three control variables: c_1 controlling where (i.e., on which input) pin a of the cells is connected, c_2 controlling the connection of pin b, and c_3 making the library cell selection. We consider that input i_1 has no other fanout in the circuit. The ADD of the capacitance at input i_1 is represented on Fig. 10. When $c_1 = 0$, $c_2 = 1$, and $c_3 = 0$, cell NAND2 is selected ($c_3 = 0$), pin b is connected to input i_1, and pin a is not connected to input i_2. Hence, the total capacitive load on i_1 is three (the load of pin b of the NAND cell).

The ADD of the power cost function is given by the following equation:

$$\mathcal{P}(\mathbf{c}) = \sum_{o \in \text{Outputs}} P_t(\mathbf{c}, o) C(o) + \sum_{i \in \text{Inputs}} P_t(i)(C_i(\mathbf{c})). \tag{14}$$

The first sum of the equation is the weighted switching activity at the outputs of the cluster, and the second sum represents the power consumed at the inputs of the cluster.

3) Timing Constraint: Similarly to power, timing depends on input assignment as well as on cell assignment. Before describing the ADD-based representation, we describe how timing constraints are computed. For each cluster output, arrival time and required time are computed. We can replace a cluster by an alternative implementation if the new arrival times at *all* cluster outputs do not exceed the required times. For timing constraints, we use the critical path of the circuit as the maximum delay that can exist from the primary inputs to the primary outputs. From this constraint, we compute the arrival and required times for all nodes.

We use the *mapped delay model* as in [6]. Consider a gate g with input pins $\mathbf{y} = [y_1, y_2, \cdots, y_n]^T$ shown in Fig. 11. The pins are connected with inputs $\mathbf{i} = [i_1, i_2, \cdots, i_n]^T$. We assume that pin y_i is connected to input i_i. The arrival time at the output of the gate is

$$t_{\text{arr}} = \max_{i=1, 2, \cdots, n} (t_{\text{arr}_i} + \alpha \times C_L + \beta_i). \tag{15}$$

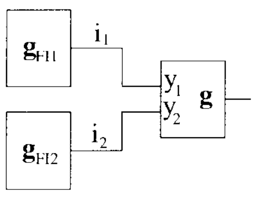

Fig. 11. Delay computation for a gate.

α is the effective output resistance of the gate, C_L is the effective load capacitance at the output, and β_i is the pin-dependent intrinsic delay of the gate. Finally, $t_{\mathrm{arr_i}}$ is the arrival time at input i_i. It is a function of both i_i and y_i because it depends on the pin of gate g and the fanin gate g_{FIi} driving input i

$$t_{\mathrm{arr_i}} = K_i + \eta_i(C_{\mathrm{other}_i} + C_{\mathrm{in}_i}). \tag{16}$$

η_i is the effective output resistance of the fanin gate g_{FIi}, K_i is the part of $t_{\mathrm{arr_i}}$ depending on previous stages (and the intrinsic delay of the fanin gate), C_{other_i} is the load capacitance of the fanin gate that does not depend on g, and C_{in_i} is the input load capacitance of pin y_i.

Observe that the arrival time at the output of a gate t_{arr} depends on the input assignment. If we change the assignment of pins to inputs, the arrival time may change for two reasons.

- The arrival time at the inputs $t_{\mathrm{arr_i}}$ changes.
- The intrinsic delay β_i changes because it is pin dependent as well.

Remember that in the quotient function $L(\mathbf{i}, \mathbf{c})$, input assignments are set by the control variables; hence, t_{arr} for a quotient function is a function of the control variables. For a cluster output that we want to match, we build the ADD $T_{\mathrm{arr}}(\mathbf{c})$ of the arrival time. It represents the arrival time at the output for any input and cell assignment. A value $\mathbf{c}^*$ of the control variables selects a path in $T_{\mathrm{arr}}(\mathbf{c})$ that leads to a leaf containing the value of the arrival time at the output when the cell and input assignment corresponding to $\mathbf{c}^*$ are chosen.

Similarly to power, the computation of T_{arr} is complicated by the fact that we are concurrently matching a multioutput cluster function using multiple quotient functions. The complication arises when we compute the arrival time $t_{\mathrm{arr_i}}$ at the inputs i_i of the cluster. Remember that $t_{\mathrm{arr_i}}$ depends on the output resistance and the load capacitance. The gates in the fanin of the cluster are loaded with a capacitance that depends on how the pins of gates in the cluster are connected to them. In symbols, $T_{\mathrm{arr_i}}(\mathbf{c}) = K_i + R_i(C_{\mathrm{other},\,i} + C_i(\mathbf{c}))$, where K_i, R_i, and $C_{\mathrm{other},\,i}$ are constants, while $C_i(\mathbf{c})$ is an ADD representing how the load capacitance on input i changes with the input assignments of the cells in the quotient functions.

It is important to notice that $T_{\mathrm{arr_i}}$ depends on the entire $\mathbf{c}$. If we are matching a two-output cluster, $T_{\mathrm{arr_i}}$ is an ADD whose support includes both $\mathbf{c}_1$ and $\mathbf{c}_2$ [the control variables of quotient functions L_1 and L_2 in (5)]. The computation of the arrival time at each output of the cluster is done with the following symbolic formula:

$$T_{\mathrm{arr}}(\mathbf{c}) = \max_i(T_{\mathrm{arr_i}}(\mathbf{c}) + \alpha(\mathbf{c}) \times C_L + \beta(\mathbf{c})) \tag{17}$$

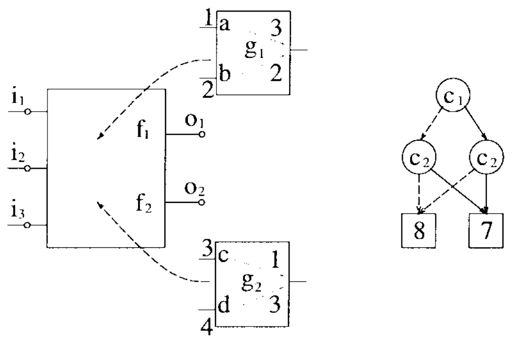

Fig. 12. Symbolic representation of timing constraints.

where $T_{\mathrm{arr_i}}$, α, and β are ADD's in the control variables and all operators involved in the computation are standard ADD operators. The leaves of T_{arr} contain all possible arrival times for the output.

Example 7: Consider the situation shown in Fig. 12. We want to compute the ADD's $T_{\mathrm{arr}}(\mathbf{c})$ for the two outputs of the cluster of Fig. 12(a). We make several simplifying assumptions for the sake of clarity. First, we assume that $K_i = 0$ and $C_{\mathrm{other}_i} = 0$ for all inputs. The driving resistances have the same value for all inputs $\eta_i = 1$, and the load on o_1 and o_2 is null. Second, we assume that f_1 can be matched only by cell g_1 and f_2 can be matched only by cell g_2. The input loads and intrinsic propagation delays for the cells are shown in Fig. 12(a). Input load values are displayed close to the input pins of the cells. Intrinsic delays are close to the arrows representing propagation paths. Moreover, we assume that g_2 can be connected only to inputs i_2 and i_3, while we can connect the input pins of g_1 to i_1 and i_2. The connection of both pins of a cell to the same input is not allowed.

With these simplifying assumptions, we just need two control variables to express the degrees of freedom in the input assignments. Control variable c_1 controls the connection of g_1: $c_1 = 0$ means that pin a is connected with input i_1 and pin b is connected with input i_2. The opposite connection is chosen when $c_1 = 1$. Similarly, when $c_2 = 0$, pin c is connected with input i_2 and pin d is connected with input i_3.

Assume, for example, that $c_1 = 0$ and $c_2 = 0$. The arrival times at the inputs are $T_{\mathrm{arr_1}}(0, 0) = C_{\mathrm{in_1}}(0, 0) = 1$, $T_{\mathrm{arr_2}}(0, 0) = C_{\mathrm{in_2}}(0, 0) = 2 + 3 = 5$, and $T_{\mathrm{arr_3}}(0, 0) = C_{\mathrm{in_3}}(0, 0) = 4$. For output o_1, the arrival time is $T_{\mathrm{arr}}(0, 0) = \max\{(3 + 1), (2 + 5)\} = 7$; this is one leaf of the ADD $T_{\mathrm{arr}}(c_1, c_2)$ representing the arrival time at o_1 for every combination of control variables. The complete ADD is shown in Fig. 12(b).

Once the ADD of the arrival times has been built, it can be used to prune the ON-set of the matching function. All assignments of $\mathbf{c}$ that lead to leaves with value of the arrival time larger than the required time violate the timing constraint and are discarded from the ON-set of M. If timing is minimization target (as opposed to constraint), the minimum delay matching solution is easily found by selecting the minimum leaf in the ADD obtained by restricting $\mathbf{c}$ to the ON-set of M.

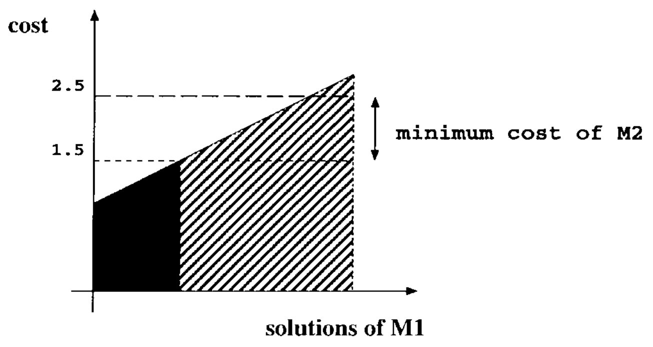

Fig. 13. Bound on the partial solutions.

C. Bounding

Our minimum-cost constrained matching algorithm merges GM with cost minimization and constraint enforcement. We exploit the existence of a network implementation (since we are remapping) and of constraints to obtain *bounds* on the cost function and tight constraints. In this way, we drastically prune the search space and further increase computational efficiency without giving up optimality.

It is not necessary to include in the quotient function cells and assignments for which we are certain that the global costs will be higher than that of the original implementation. Additionally, it is useless to express assignments that would violate the timing constraints. These assignments can be suppressed when building the quotient function.

Let us consider area minimization under delay constraints first. When computing $L(\mathbf{i}, \mathbf{c}_1)$ and $L(\mathbf{i}, \mathbf{c}_2)$, some library cells are not even included because their area is too large. The area of a cell included in the quotient function must be $A < A_{\rm old} - A_{\rm MIN}$, where $A_{\rm old}$ is the area of the current implementation of $\mathbf{f}$ and $A_{\rm MIN}$ is a lower bound to the area of any cell in the library. If a library cell has area A larger than $A_{\rm old} - A_{\rm MIN}$, it is not a viable candidate for replacing any cell in the original target cluster because the new implementation will have area larger than or equal to $A + A_{\rm MIN}$, which is larger than $A_{\rm old}$.

After M_1 and M_2 have been computed, we can use both area and timing to further reduce the solution space that has to be explored by (5). We call $A_{\rm MIN,\,1}$ the minimum area of any match in M_1. All matches in M_2 such that area $A_2 \geq A_{\rm old} - A_{\rm MIN,\,1}$ can be pruned because the total area of a solution involving them is certainly larger than $A_{\rm old}$. The same reasoning can be done for M_1, A_1, and $A_{\rm MIN,\,2}$.

Example 8: Consider the example in Fig. 13. The graph shows the cost of all solutions in M_1. The x-axis corresponds to the solutions, and the y-axis to the area cost. The cost 2.5 is the original cost of the cluster (for both outputs) before remapping. We assume that the minimum cost for M_2 (not shown) is one. We can prune all solutions in M_1 that have a higher cost than $2.5 - 1 = 1.5$ without loss of optimality because these solutions will lead to a higher cost than the original implementation of the cluster. The gray area corresponds to the solutions that we can discard. We keep only the solutions of M_1 in the black area.

For timing constraints, the line of reasoning is similar but more involved. Observe that delays depend on input loads; therefore, when we concurrently match two or more cells, we need to take into account the load that the cells causes on the fanin gates. We call $D_{\rm Bound,\,2}$ the delay for a match in M_2 (i.e., a cell implementing f_2), assuming that the load caused by the cell implementing f_1 is the minimum among all matches in M_1. If $D_{\rm Bound,\,2}$ exceeds the timing constraint, the match can be pruned. The same reasoning holds for M_1 and $D_{\rm Bound,\,1}$.

Notice that both timing and area bounds are conservative and do not prune any match that can improve the current mapping of the cluster. Roughly speaking, we use area and timing to prune solutions in M_1 (M_2) that could not be optimal even if they were coupled with the best possible match in M_2 (M_1). The bounds are very useful in further decreasing the number of candidate matches for M and the efficiency of the computation of (5).

1) Power-Based Bounding: When performing constrained power minimization, power bounds must be computed. Power-based bounding is conceptually the same as area-based bounding, but its implementation is more involved. To suppress assignments that have a cost higher than the original implementation, we compute a power bound for each cell. This bound is the *minimum possible cost* that this cell has in any implementation in the cluster. If the bound is higher than the original cost in power, we discard the cell.

To compute the bound, we have to analyze two factors: the probability of transition at the outputs and the capacitance at the inputs. The value of the probability of transition is not completely free at the outputs. The outputs are bound by the Boolean relation $\mathcal{F}(i, o)$. This relation defines a *range* on the transition probability at one of the outputs o_1. Like in (13), we express the *minimum* transition probability at o_1 as

$$\begin{aligned} P_{t\,\min}(o_1) &= \exists_{\mathbf{i},\,\mathbf{i}^+}\big[\big(\exists_{\mathbf{o},\,\mathbf{o}^+}\mathcal{F}(\mathbf{o},\,\mathbf{i})\mathcal{F}(\mathbf{o}^+,\,\mathbf{i}^+)\big(o_1 \neq o_1^+\big)\big) \\ &\qquad \wedge \big(!\exists_{\mathbf{o},\,\mathbf{o}^+}\mathcal{F}(\mathbf{o},\,\mathbf{i})\mathcal{F}(\mathbf{o}^+,\,\mathbf{i}^+)\big(o_1 = o_1^+\big)\big)\big]P(\mathbf{i},\,\mathbf{i}^+). \end{aligned} \tag{18}$$

The bound at output o_2 is obtained by changing the index 1 to 2.

Given two consecutive input patterns $\mathbf{i}$ and $\mathbf{i}^+$, the formula inside the outer parenthesis expresses that o_1 switches regardless of the target function, as long as it satisfies the Boolean relation. The Boolean relation gives such degrees of freedom that o_1 could switch or not switch under the *same* input patterns, depending on the choice of the target function. To get the minimum transition probability given by the Boolean relation, we express that o_1 may switch under two given input patterns (first term), and under these same patterns, there is no representation where it could *not* switch (second term).

This value makes a bound on the transition probability at the outputs, regardless of the cell representatives.

Example 9: Boolean relation is displayed in the truth table on Fig. 14. A valid representative for o_1 is $g_1 = \overline{i_1 i_2}$. But the function $g_2 = \bar{i}_1$ may also be a representative. We assume that the input probabilities are $P_s = \{0.5, 0.5, 0.5\}$ and $P_t = \{0.5, 0.5, 0.5\}$. The transition probability at o_1 using g_1

$i_1i_2i_3$	o_1o_2
000	$\{10, 01, 00\}$
001	$\{11\}$
010	$\{11\}$
011	$\{11\}$
100	$\{10, 01, 00\}$
101	$\{11, 01\}$
110	$\{10, 01, 00\}$
111	$\{01, 00\}$

Fig. 14. Boolean relation for a cluster with three inputs and two outputs.

is 0.375, and using g_2 is 0.5. We see how the transition may be different for o_1 according to the representative. For example, under the two consecutive input patterns (001, 100), o_1 may stay at one or switch. The bound on the transition activity is reached here for $o_1 = \overline{i_1 i_2 i_3}$. With this implementation, the transition activity is 0.218, as computed with (18). No cell representation can yield a lower transition activity at o_1.

The bound takes into account the input capacitance as well. Without considering the functionality of the cell, we want to express a lower bound on the power consumed to drive the inputs of any cell in the library. We assume that for each cell in the library, the functionality that can be obtained by shorting two inputs of the cell is already more efficiently implemented by another cell with less inputs. Consider a cell g with inputs capacitances sorted in *decreasing* order $\{C_1, C_2, \cdots, C_n\}$. If cell pins could be connected with any cluster input, regardless of functionality, the minimum-power connection would be $C_1P_{t1} + C_2P_{t2} + C_3P_{t3} + \cdots$, where $\{P_{t1}, P_{t2}, \cdots, P_{tk}\}$ are the transition probabilities of the inputs sorted in *increasing* order. This is the *minimum possible* power consumed at the inputs of this cell for any input permutation.

We add to this bound the minimum power at both outputs as computed in (18). We do not know what is the other cell selected for the cluster, and therefore what is the power consumed at its inputs. So we sum to the bound the best case, usually an inverter because it has only one connection, and its power contribution is very low. If this bound is larger than the original power value, i.e., the power of the original implementation, we can discard the cell. We see here that the effectiveness of the bound depends highly on the quality of the original implementation. However, if the original implementation is of bad quality, we can use a tighter value as the original value in a first pass, get a suboptimal solution, and run again by releasing the tighter value.

The algorithm for computing the power bound is represented in Fig. 15. We first compute (in `other_pow`) the minimum power contribution at the outputs and at the inputs for capacitances C^{other} of cells going out of the cluster. Then, for each cell, we compute the minimum power that we can find at the inputs of this cell, `min_pow_in`. We first sort the capacitances, then `sum_products` computes the sum $C_1P_{t1} + C_2P_{t2} + C_3P_{t3} + \cdots$ as described above. We add to the minimum power for a cell the minimum power that we can find at the input of the other cell composing the cluster. The latter is the minimum capacitance found in the library times the next transition probability in the order defined. We then compare this bound to the power of the original implementation, `orig_pow`. If the bound is higher than the original implementation, we can discard the cell.

```
EliminateExpensiveCells (clusterF, bddBR)
  other_pow = ADD_ZERO;
  /* Adds the minimum power at the outputs */
  foreach (o $\in$ outputs (clusterF))
    other_pow += get_min_tp (bddBR, o) * cap (o);
  /* Adds the power contribution of $C^{other}$ */
  foreach (i $\in$ inputs (clusterF))
    other_pow += get_cap_other (i, clusterF)*get_node_tp(i);
  sorted_tps = sort_increasing (input_tps);
  foreach (cell $\in$ library) {
    min_pow = other_pow;
    sorted_caps = sort_decreasing (cell_cap);
    min_pow_in = sum_products (sorted_tps, sorted_caps, &next_tp);
    /* min_input_cap is the smallest input capacitance in the library */
    min_pow = min_pow_in + min_input_cap (library) * sorted_tps[next_tp];
    if (min_pow > orig_pow)
      eliminate_cell (cell);
  }
```

Fig. 15. Algorithm for computing the bound on library cells.

```
ComputeBRelOptimize(clusterF, Network)
  neighborhood = ComputeNeighborhood(clusterF, Network)
  bdd_rel = make boolean relation (neighborhood, clusterF)
  c = compute control variables according size of clusterF
  bdd_quot = compute_quotient_function_with_area_bounds( c )
  /* pass of matching output per output */
  foreach (o $\in$ outputs (clusterF))
    bdd_m_only[o] = compute_matching_function (o, bdd_quot);
  /* bounding with area and timing constraints */
  foreach (o $\in$ outputs (clusterF))
    bdd_m_only[o] = bound_with_area (clusterF, bdd_m_only[]);
    bdd_m_only[o] = bound_with_timing_constraint (clusterF, bdd_m_only)
    if (bdd_m_only[o] == NIL)
      return (NIL);
    bdd_red_quot[o] = compress_quotient_function (bdd_quot, bdd_m_only[o])
  }
  bdd_m = compute_matching_function (clusterF, bdd_red_quot[])
  bdd_m = timing_constr (bdd_m)
  bdd_best = get_best_area (bdd_m, add_area)
  /* In the case of one the three function fails:
  no match, or violation of the timing, or no best area
  than the current implementation */
  if (bdd_best == NIL)
    return (NIL);
  /* Puts the bdd_best in ''readable'' form, i.e.  returns in best_match
  the cell number and the pin connections rather than a bdd */
  best_match = analyze_best_minterm (bdd_best)
  return (best_match);
```

Fig. 16. Algorithm of the matching step.

Notice that this bound is very conservative because we do not want to compromise the optimality of the solution. Fortunately, it gives the opportunity for discarding on average 38% of the cells in the library, at a very low computational cost. This result is important because it improves the overall run time of the algorithm.

The cost of `EliminateExpensiveCells` is mostly due to the computation of `get_min_tp`, which is itself in the order of the size of the Boolean relation `bddBR`. Since the BDD representation of this relation is very small (the BDD has approximately ten variables), the overall cost of the procedure is small.

D. The Complete Matching Algorithm

Having described all subtasks involved in performing minimum-cost constrained matching, we conclude our analysis by describing the complete matching algorithm. The pseudocode of the minimum-area constrained GM algorithm is shown in Fig. 16. The minimum-power version of the algorithm has similar structure.

The algorithm first finds the neighborhood of the cluster function and computes the Boolean relation `bdd_rel`, as seen in Section III. Then the quotient functions are constructed disregarding the library cells whose cost cannot improve the current solution, as seen in Section IV-C. The single-output matching functions are then computed by `compute_matching_function` and pruned using area bounds and timing constraints, as discussed in Section IV. The quotient functions are then compressed using the conservative bounds, and the new, smaller `bdd_red_quot` are used for performing full generalized matching on the reduced search space.

The resulting matching function is then pruned using timing constraints: in function `timing_constr`, all solutions violating the constraint are eliminated. Finally, the cost function is applied and the subset of the ON-set of $M(\mathbf{c})$ containing minimum-cost solutions is obtained. The matching algorithm then "decodes" one of the minimum solutions and returns the cells and the input assignments for replacement. NIL is returned if there are no solutions improving the current mapping of the cluster.

Several performance-enhancing features complicate the algorithm. What is shown in Fig. 16 is a simplified version. For example, caching of previous ADD and BDD computations is heavily exploited (not only the simple caching mechanisms provided by BDD packages), an advanced algorithm has been implemented for the compression of $M(\mathbf{c})$ after bounding, and several corner conditions are flagged to speed up the computation of trivial cases.

V. Timing Optimization

The speed of a circuit is an intensive quantity. In timing optimization, the goal is to minimize the critical path of the circuit. Thus, we need to consider only the gates that are in the critical path. The simplest optimization strategy is to build clusters with sets of gates in the critical path (i.e., *critical gates*). However, we want to consider also gates that are not in the critical path. In this case, what ideally we would like to do is to "borrow" some delay from noncritical gates that have some delay slack.

A good search strategy is to define clusters including at least one critical gate and one or more noncritical ones. The cost function tries to minimize the delay of the gate(s) in the clusters that are on the critical path while satisfying the timing constraints on the other gates (i.e., ensuring that the noncritical gates do not become critical).

The algorithm for traversal and cluster generation is briefly outlined. We first mark all critical gates. Then, for each multiple fanout point among either the inputs or the outputs of critical gates, we construct clusters as in Section III and we apply GM. Last, we construct all possible clusters containing pairs of critical gates and apply GM.

After each successful matching (i.e., a matching that reduces the delay for a gate on the critical path), we do not recompute the critical path, but we move to other critical nodes. The critical path is recomputed after all critical gates have been traversed. If at least one successful replacement has been performed, we continue the optimization on the new critical path. Otherwise, remapping terminates.

Since the total number of clusters generated (and of attempted GM's) in one pass on the critical path is usually not very large, we can afford to attempt aggressive optimization: the size of the neighborhood for extracting Boolean relation $\mathcal{F}$ is increased, and clusters with more than two outputs can be attempted.

During generalized matching, we use only the bound obtained from the delay of the initial implementation. There is no other bound on the quotient function like for area or power (where we could bound on constraints and cost) because we do not have any secondary cost function. Fortunately, computation times are not degraded because i) we perform a smaller number of matches and ii) the netlists we start with are already optimized for speed; hence, the delay bounds are usually tight.

The goal of timing optimization is to increase the maximum slack at the outputs of the cluster. The flow of timing optimization can be summarized as follows: we first compute the arrival times of each output with the procedure employed in the previous section for the computation of timing constraints. Slacks are computed by subtracting arrival times to required times. The computation is performed symbolically using ADD's. Given the ADD $SL(\mathbf{c})$ of the minimum output slacks as a function of the inputs and cell assignments, the maximum speed assignments are the values of $\mathbf{c}$ that lead to the maximum leaf of $SL(\mathbf{c})$.

VI. Results

We have implemented a postmapping optimization tool based on generalized matching. The tool reads a mapped circuit described in `blif` (or `slif`) and a library file and runs the optimization. Several user-controlled parameters can be specified. The depth of the neighborhood can range from zero to infinity. Specifying a depth of zero reduces the neighborhood to the cluster, while depth of infinity means that the entire logic network is taken as neighborhood. The latter choice is of course only conceivable for small circuits.

The number of outputs N_o of a cluster can be also controlled. We made experiments with up to four outputs. The number of inputs N_i of a cluster can be controlled as well. Usually they are assumed to be the inputs of the cells implementing the cluster in the original mapped netlist. However, additional input can be added taken from nodes in the neighborhood. With this simple modification, we can exploit the power of generalized matching to perform local rewiring.

We can also change the cluster selection algorithm to select arbitrary sets of nodes as clusters. Experimentally, we observed that this is much less effective than starting from multiple fanout points, mainly because traditional logic optimization is already effective on fanout-free cones.

A generic cost function has to be a function returning an ADD whose support are the control variables and leaves are the cost values. Of course, a new bounding function may be integrated with the new cost function. All the experimental statements in this paper rely on the fact that we can easily tune cost functions and bounds.

Memory optimization is the primary concern in the implementation. The tool employs the `Cudd` BDD package [20], which provides a rich set of operators on BDD's and ADD's and powerful memory management and caching features. We set up a memory limit of 1 000 000 BDD and ADD nodes. When this limit is reached, the matching function exits with the value `NIL` and the traversal continues. When the BDD's exceed the memory limit, the program simply frees the memory and moves on.

Extensive tests demonstrated that bounding is necessary and effective. Without any bounding, the memory threshold is often reached when the number of inputs of the cluster is larger than eight. With bounding, we are below the threshold for up to 12 inputs. The compression of the matching functions using single-output matching as a conservative bound is probably the most useful algorithmic optimization. The size of the uncompressed quotient functions makes it very difficult even to match two-output clusters, but the algorithm using separate matching and compression greatly increases the percentage of matchings that can be successfully carried out.

By using the aforementioned optimization, two-output matching can always be carried out, whereas three-output and four-output matching succeed in 60 and 40% of the cases, respectively. Obviously, for clusters with four or more outputs, the gain brought by extending the size of the cluster is lost because of the frequent memory blowup. To improve the chances of success for three or four outputs, we implemented tighter bounds that allow further compression of the quotient functions but imply the loss of some potentially advantageous matchings. We do not describe the implementation of such aggressive bounds.

To further improve efficiency, we use special caching of the intermediate results. We cache the quotient function and, in the cases where the matching fails, the intermediate BDD's of the neighborhood. This caching speeds up the overall matching and marginally affects memory consumption. Caching is useful to speed up both the network traversal and the matching itself. Traversal is faster because cases were GM is not productive are not retried. Matching is faster because several partial results are often found in cache and can be reused.

The BDD variable ordering has been set after extensive experimentation. We use a fixed variable order that minimizes the BDD peak size, regardless the intermediate results size. The order is the following. The control variables of the input multiplexers in the quotient function pin are at the bottom, preceded by the input variables, the output variables, and the library selection variables. Different orders lead to BDD blowup with high probability. Automatic reordering is not a good solution because it can destroy the good ordering to reduce the size of intermediate results and it often cannot recover a good ordering when the peak in BDD size is reached.

Our goal was to be able to show a practical realization of the remapping procedure. Therefore, we implemented the algorithm targeting robustness and conservatively set the parameters to avoid failure even in corner cases and produce results in a reasonable amount of time. Needless to say, with this choice we gave up some optimality.

TABLE I
RESULTS ON MCNC BENCHMARKS

Bench	gates	speed		area		power	
		%	CPU	%	CPU	%	CPU
z4ml	48	1.97	2	4.35	1	13.24	1
b9	110	7.04	5	6.95	7	15.50	5
term1	179	0.47	12	8.42	27	14.12	26
C432	181	2.69	11	6.60	14	13.35	27
9symml	204	3.72	17	9.18	18	21.80	12
alu2	347	4.15	31	11.68	29	18.96	87
x4	364	1.36	5	4.29	10	15.98	10
C499	365	20.76	31	25.96	69	32.04	10
C880	377	6.34	45	6.59	29	12.42	64
C1908	508	5.48	46	14.68	88	15.88	54
C1355	524	6.48	89	16.97	15	15.77	53
too_large	573	1.22	21	5.31	37	16.03	27
x3	639	1.04	11	5.39	20	12.53	23
rot	671	2.18	30	7.81	31	14.22	22
apex6	691	3.84	33	4.19	24	13.63	30
alu4	697	6.66	72	8.58	45	16.15	149
frg2	774	5.78	24	8.47	25	21.53	29
vda	781	10.59	41	21.25	104	34.78	108
t481	863	4.39	40	10.42	64	22.75	51
C2670	943	1.39	46	10.85	111	14.69	40
dalu	966	1.96	24	7.30	76	20.28	75
k2	1212	4.88	45	6.04	117	10.33	142
C3540	1344	5.54	92	11.15	142	19.20	156
pair	1480	3.70	64	5.88	96	18.48	74
C5315	2039	9.61	218	8.74	271	13.43	187
des	3621	5.06	241	5.05	289	12.54	495
C7552	3716	9.45	306	29.55	417	38.14	268
C6288	4373	9.23	341	24.79	557	31.85	279
Total		6.05		13.71		22.39	

We have experimented our tool with a set of combinational MCNC'91 benchmarks [12] including all largest ones. The benchmarks were first optimized with Berkeley's Sequential Interactive Synthesis (SIS) system [6] for minimum delay with area recovery, with script `script.delay` followed by the mapping command `map -n 1 -AFG`.

We used a library based on an industrial technology file, with 75 cells, with up to five inputs. All two-input functions are available, as well as NAND and NOR gates with higher fanin. Some *and-or-invert*, *or-and-invert* and multiplexer gates are included. Two different sizes are provided for each two-input cell. Three sizes are available only for inverter and buffer. The library is rich in functionality but relatively poor in sizing options. This choice was made for stressing the capability of GM for finding new functional matches more than choosing good resizing options (specifically targeted tools are available for this task).

Our tool was run with the same parameter settings on all benchmarks in an effort to demonstrate robustness and generality. We ran the matching algorithm on clusters of two outputs, with the neighborhood search limited to a depth of three. The number of inputs of the cluster was limited to ten. Although our technique is ideally suited for matching multioutput cells, typical libraries do not include many such cells. Hence, we decided to study the impact of our technique for a library containing only single-output cells. The quality of the results should increase if a library with many multioutput cells is employed.

We show in Table I the results on all the MCNC benchmarks for speed, area, and power optimization. The starting point was the same for all optimizations, namely, the circuits mapped by SIS. The table gives for each benchmark the number of gates, the percentage gain, and the run time in minutes (on SPARC20 with 256 Mb of memory) for each kind of optimization. The

TABLE II
TRADEOFFS IN OPTIMIZATIONS

Bench	timing opt		area opt	power opt
	power	area	power	area
z4ml	2.22	3.30	11.50	0.40
b9	3.28	4.57	5.38	4.01
term1	0.14	0.51	5.28	2.08
C432	0.71	-0.48	-5.46	2.89
9symml	-0.84	0.40	10.48	5.04
alu2	1.35	2.69	9.63	6.27
x4	-0.09	-0.00	11.66	2.27
C499	5.85	13.59	12.81	12.68
C880	0.60	1.06	-7.25	2.74
C1908	0.54	3.57	-5.67	3.34
C1355	4.06	5.06	12.14	10.78
too_large	0.18	0.41	3.91	1.03
x3	0.07	0.07	0.34	1.14
rot	0.27	0.35	1.72	3.90
apex6	0.35	0.17	1.68	0.62
alu4	1.36	2.39	6.29	4.74
frg2	0.87	1.28	17.18	4.41
vda	0.65	2.15	25.39	12.22
t481	0.76	2.40	10.15	5.03
C2670	0.39	0.36	-1.82	3.76
dalu	0.14	0.12	9.22	2.36
k2	0.40	1.55	-1.95	-2.00
C3540	1.50	2.38	6.35	4.54
pair	0.30	0.84	2.32	1.22
C5315	0.37	0.69	-9.80	0.92
des	0.29	1.02	4.98	2.20
C7552	0.69	0.94	27.67	20.19
C6288	6.70	7.27	17.53	14.73
Total	1.60	2.32	9.70	7.05

TABLE III
RESULTS FOR MULTIPLE-OUTPUT CELL MATCHING

Bench	area ($\leq$ 3 inp.)	area ($\leq$ 5 inp.)
z4ml	3.21	3.85
b9	0.0	2.62
term1	1.82	2.72
C432	0.0	2.06
9symml	0.0	5.34
alu2	0.0	4.21
x4	0.21	2.30
C499	1.85	5.53
C880	0.0	2.91
C1908	2.81	4.19
C1355	4.30	5.14
too_large	1.02	1.34
x3	1.51	2.39
rot	1.11	2.04
apex6	0.0	1.72
alu4	0.0	4.57
frg2	1.35	4.75
vda	3.61	5.26
t481	1.61	2.71
C2670	0.0	5.46
dalu	1.79	3.65
k2	2.0	2.53
C3540	2.08	3.47
pair	1.57	3.03
C5315	1.13	2.31
des	1.02	1.98
C7552	4.93	6.38
C6288	4.33	6.60
Total	1.50	3.70

last line gives the average gain for each optimization. In the average, the gain is weighted by the size of the circuit.

We observe an average gain of 6% in speed, 13.7% in area, and more than 22.3% in power. For area and power optimization, the critical path of the circuit has been constrained to remain the same as the original circuit: no tradeoff has been allowed between the delay constraint and the (area/power) cost function.

When looking at the results on a benchmark-by-benchmark basis, we observe that the quality of the optimization achieved is consistent when the cost function is changed. This phenomenon can be explained by the fact that some benchmarks have many MFP's and reconvergent fanout cones. Both these characteristics increase the effectiveness of our optimization tool. Notice also that very good improvements are obtained for the largest benchmarks. We conjecture that the global optimization of SIS is less efficient for large benchmarks, and remapping can recover a big fraction of the optimality loss.

The run times of the remapping tool are shorter (but on the same order) than those spent by SIS in technology-independent and technology-dependent optimization. Most of the time is spent in building the matching function and in universal quantification of the variables. The average time of a single match is on the order of the tenth of a second with this machine configuration. The percentage of successful matches, i.e., matches that find a better solution, range from 5 to 10%.

Table II provides detailed information on the tradeoff involved in the optimization process. The first column contains the benchmark name. The second and third columns give the percentage change (positive if gain, negative if loss) in area and power, respectively, when doing unconstrained delay optimization. The fourth column gives the percentage change in power for area optimization (no change is allowed in speed). The last column gives the change in area when power is targeted (again, speed is the constraint and no tradeoffs are allowed with it).

We observe that delay optimization leaves area and power almost unchanged. This result is intuitive, since delay optimization focuses only on the critical path, which is usually a small fraction of the entire circuits. Only two benchmarks have an increase in area or power, and for these two benchmarks, the delay is only marginally reduced. We observe also that, in general, area is not traded off for power and vice versa. In general, area decreases when doing power optimization, and power decreases when doing area optimization. This is expected, since power is in first approximation the product between area and switching activity; hence, it is related to area.

One last set of experiments was run to estimate the potential of GM for matching multiple-output cells. Industrial libraries do not usually include many multiple-output cells; hence, we built two test libraries with the following characteristics. The first library includes two-output cells with two or three inputs. These cells are functionally equivalent to pairs of single-output cells with one or two inputs that share one input. The second library contains two-output cells with five to three inputs, which are functionally equivalent to pairs of single-output cells with three or less inputs that share one input. Each multiple-output cell is assumed to occupy 20% less area than the total area occupied by the two equivalent single-output cells. The area reduction represents the expected savings achievable by careful layout of two merged cells.

Remapping was performed with the new libraries. Results are summarized in Table III. Column 2 reports the percentage area savings obtained with the first library with respect to the minimum-area implementation of Table I. Column 3 reports the savings obtained using the second library. Apparently, our tool is capable of fruitfully exploiting multiple-output cells. On the other hand, the first library contained 50% more cells than the original one, while the second library contained approximately twice the number of cells as the original one. This is a significant overhead if cells have to be hand designed one by one. If this is the case, the additional area savings may not be sufficient to justify the investments required to develop the library. In summary, our results indicate that GM can make good use of multiple-output cells, but the advantages are incremental.

VII. CONCLUSIONS

In this paper, we proposed a remapping approach that exploits the power of Boolean relations to optimize a mapped netlist under tight constraints. Our main objective is to build a powerful, robust, and efficient optimization tool that can be applied to large circuits. We have presented the theoretical foundation of our approach and several algorithmic improvements that are needed to achieve the targeted robustness and speed.

The core remapping engine is based on generalized matching, a Boolean matching technique that enables concurrent mapping of multiple-output logic networks specified by Boolean relations, a problem that has no previously known solution. We have extended basic GM to deal with cost function minimization and constraint satisfaction, and we proposed an efficient algorithm for the exact solution of the extended GM problem.

We tested the effectiveness of our approach on a large set of benchmarks. The results show that our optimization tool can reduce the area by more than 13.7% in average or reduce power by more than 22.3% without any speed penalty. Unconstrained speed optimization is effective as well (more than 6% average speed improvement is achieved). The optimization is performed starting from mapped circuits that have been optimized using traditional technology-independent and technology-dependent techniques.

APPENDIX
POWER-ESTIMATION ENGINE

Power estimation is obviously a prerequisite for power optimization. Unfortunately, there is no complete agreement in the literature on what type of power-estimation engine should be used during logic optimization for driving the choice of transformations and checking the final results. There is some consensus on using zero-delay power estimates during optimization. Although glitch power is neglected, it has been observed that zero-delay estimates provide reliable *relative power information*. In other words, they can be used to choose among alternative implementations.

On the contrary, the choice of which approach is to be used to obtain zero-delay estimates is still a controversial issue. Two classes of methods have been proposed. Probabilistic methods avoid multiple simulations by propagating probability values in one single pass through the network [24]–[26]. Statistical methods rely on traditional simulation and define stopping criteria based on sampling theory to decide when the average power estimates are within a user-specified interval of confidence [28]. While there is no dispute in recognizing the better accuracy of statistical method, the most discordant claims have been made about efficiency issues.

We take an intermediate position based on the observation that probabilistic methods are not robust enough to deal with general and large networks, but they are substantially faster than statistical methods on small circuits. Our power-estimation engine is based on a hybrid approach. We use a statistical technique to obtain accurate power estimates on the entire network, then we employ fast local estimates based on a probabilistic method in the inner optimization loop.

Power estimation is needed during two phases of the remapping process. First, it is needed for computing the signal probabilities and transition activities for the entire network. Second, it is required to test the validity of the local remappings performed during optimization. In Section IV-B2, we computed the power cost function assuming that the inputs of the cluster were uncorrelated and a change in the transition activity of the cluster outputs does not have significant consequences in the fanout gates within the neighborhood. Neither of these assumptions is verified in the general case; consequently, we need to check a remapping with more accurate power estimation. We call this step *resimulation*.

The computation of signal probabilities and transition activities for the entire network is performed using a Monte Carlo approach [28] based on bit-parallel simulation (BPS) [27]. The efficiency of BPS is high: we could simulate thousands of patterns for our largest benchmarks in a few seconds. Simulation time grows linearly with network size. Patterns or, alternatively, probabilities and transition activities can be specified for the inputs. The Monte Carlo stopping criterion can be overridden by the user, and full simulation of a pattern file can be performed.

Resimulation was implemented using BDD-based probabilistic techniques. Remember that the functionality of the network is unchanged outside the neighborhood of a cluster when a remapping has been performed. Hence, we can resimulate the neighborhood alone to check that the power saved in the remapping is not swamped by the effect of remapping on fanout gates within the neighborhood. Since the neighborhood is a small fraction of the entire network, BDD-based probabilistic power estimation is performed in a very short time [24]. It is important to notice that resimulation does not take into account the correlation at the inputs of the neighborhood; thus, it is not as accurate a global simulation in estimating the effects of a remapping (but it is much faster).

After implementing the resimulation engine, our experiments revealed that the estimated power savings computed by the cost function were extremely close to those given by resimulation. This was a surprising discovery, since it has been claimed that local modifications may have effects on the power of the transitive fanouts [29]. Suspecting a bug in our resimulations, we actually ran a set of computationally expensive tests. We applied remapping to several benchmarks, and after each successful cell replacement, we simulated *the entire network* using BPS. The surprising results are shown in Fig. 17. The y-axis shows the power differences ($\mathcal{P}_{\text{after}} - \mathcal{P}_{\text{before}}$), while the

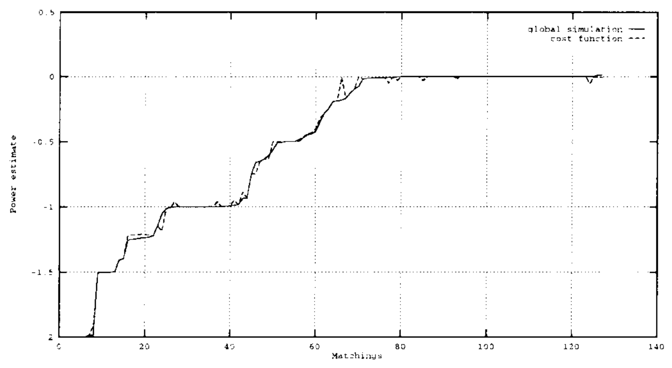

Fig. 17. Power cost function versus simulation.

x-axis shows the cumulative number of matches. Matches are ordered for increasing differences. Two curves are plotted: the power differences predicted by the cost function and those actually measured by full BPS simulation. The reader can observe that there is almost perfect agreement. This plot was obtained performing a large number of remappings, and it is consistent across our benchmarks.

It appears that the cost function is very accurate in estimating power savings and losses, and resimulation is actually not needed. Although it is implemented and functional, it was not used in our runs. Nevertheless, to protect ourselves against pathological cases, we perform global BPS simulation every few remappings (usually ten). If power is increased, the remappings can be undone. We conjecture that this event is extremely rare: it never happened in our tests.

Acknowledgment

The authors would like to thank F. Somenzi for providing the `Cudd` package.

References

[1] F. Somenzi and R. K. Brayton, “Minimization of Boolean relations,” in *Proc. IEEE Int. Symp. Circuits and Systems*, May 1989, pp. 738–473.

[2] R. Brayton, G. Hachtel, and A. Sangiovanni-Vicentelli, “Multilevel logic synthesis,” *Proc. IEEE*, vol. 78, pp. 264–300, Feb. 1990.

[3] M. Damiani and G. De Micheli, “Don’t care set specifications in combinational and synchronous logic circuits,” *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 365–388, Mar. 1993.

[4] H. Savoj, R. Brayton, and H. Tuati, “Extracting local don’t cares for network optimization,” in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1991, pp. 514–517.

[5] L. Benini and G. De Micheli, “A survey of Boolean matching techniques for library binding,” *ACM Trans. Design Automat. Electron. Syst.*, vol. 2, no. 3, pp. 193–226, July 1997.

[6] E. Sentovich, K. Singh, C. Moon, H. Savoj, R. Brayton, and A. Sangiovanni-Vincentelli, “Sequential circuits design using synthesis and optimization,” in *Proc. Int. Conf. Computer Design*, Oct. 1992, pp. 328–333.

[7] F. Brown, *Boolean Reasoning*. Norwell, MA: Kluwer Academic, 1990.

[8] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.

[9] W. Kunz and D. Stoffel, *Reasoning in Boolean Networks*. Norwell, MA: Kluwer Academic, 1997.

[10] R. Bryant, “Graph-based algorithms for Boolean function manipulation,” *IEEE Trans. Comput.*, vol. C-35, pp. 677–691, Aug. 1986.

[11] F. Mailhot and G. De Micheli, “Algorithms for technology mapping based on binary decision diagrams and on Boolean operations,” *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 599–620, May 1993.

[12] J. Mohnke and S. Malik, “Permutation and phase independent Boolean comparison,” *Integration, VLSI J.*, pp. 109–129, Dec. 1993.

[13] K. Cheng and L. Entrena, “Multi-level logic optimization by redundancy addition and removal,” in *Proc. Eur. Conf. Design Automation*, Feb. 1993, pp. 373–377.

[14] W. Kunz and P. Menon, “Multi-level logic optimization by implication analysis,” in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1994, pp. 6–13.

[15] B. Rohileisch, B. Wurth, and K. Antreich, “Logic clause analysis for delay optimization,” in *Proc. Design Automation Conf.*, June 1995, pp. 668–672.

[16] S. Chang, L. Van Ginneken, and M. Marek-Sadowska, “Fast Boolean optimization by rewiring,” in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1996, pp. 262–269.

[17] K. Keutzer, “DAGON: Technology binding ant local optimization by DAG matching,” in *Proc. Design Automation Conf.*, June 1987, pp. 341–347.

[18] Y. Watanabe, L. M. Guerra, and R. K. Brayton, “Permissible functions for multioutput components in combinational logic optimization,” *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 734–744, July 1996.

[19] R. Bahar, E. Frohm, G. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi, “Algebraic decision diagrams and their applications,” in *Proc. Int. Conf. Computer Aided Design*, Nov. 1993, pp. 188–191.

[20] F. Somenzi, *The CUDD Package User’s Guide*, version 1.0.5, Nov. 1995.

[21] S. Yang, “Logic synthesis and optimization benchmarks user guide version 3.0,” Microelectronics Center of North Carolina, Research Triangle Park, NC, Tech. Rep., Jan. 1991.

[22] C. Tsui, M. Pedram, and A. Despain, “Power efficient technology decomposition and mapping under an extended power consumption model,” *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 1110–1122, 1994.

[23] Synopsys, Inc., *Power Compiler User Manual*, version 3.5, 1996.

[24] R. Marculescu, D. Marculescu, and M. Pedram, “Logic level power estimation considering spatiotemporal correlations,” in *Proc. Int. Conf. Computer Aided Design*, 1994, pp. 294–299.

[25] T. Chou, K. Roy, and S. Prasad, “Estimation of circuit activity considering signal correlations and simultaneous switching,” in *Proc. Int. Conf. Computer Aided Design*, 1994, pp. 300–309.

[26] P. Schneider, U. Schlichtmann, and B. Wurth, “Fast power estimation of large circuits,” *IEEE Design Test Comput. Mag.*, vol. 13, pp. 70–78, 1996.

[27] P. Schneider, “PAPSAS: A fast switching activity simulator,” in *Proc. Workshop Power and Timing Modeling, Optimization and Simulation*, 1995, pp. 350–360.

[28] R. Burch, F. Najm, P. Yang, and T. Trick, “A Monte Carlo approach for power estimation,” *IEEE Trans. VLSI Syst.*, vol. 1, no. 1, pp. 63–71, 1993.

[29] C. Lennard and A. Newton, “On estimation accuracy for guiding low-power resynthesis,” *IEEE Trans. Computer-Aided Design*, vol. 15, no. 6, pp. 644–664, 1996.

Luca Benini (M’98), for a photograph and biography, see p. 232 of the March 1998 issue of this Transactions.

Patrick Vuillod received the computer science engineering degree from Ingénieur ENSIMAG, Grenoble, France, in 1993 and the master’s and Ph.D. degrees in computer science from INPG, Grenoble, in 1994 and 1997, respectively.

He has worked in Grenoble in research and development for IST in cooperation with INPG-CSI. He currently is with Synopsys-Epic, Gieres, France. His current research interests are in logic synthesis and synthesis for low-power systems. His previous works were on high-level description languages and synthesis for FPGA’s.

Giovanni De Micheli (F’94), for a photograph and biography, see p. 232 of the March 1998 issue of this Transactions.