Generalized matching from theory to application *

Patrick Vuillod† Luca Benini Giovanni De Micheli

Stanford University
Computer Systems Laboratory
Stanford, CA 94305

Abstract

This paper presents a novel approach for post-mapping optimization. We exploit the concept of generalized matching, a technique that finds symbolically all possible matching assignments of library cells to a multi-output network specified by a Boolean relation. Several objectives are targeted: area minimization under delay constraints, power minimization under delay constraints and unconstrained delay minimization.

We describe the theory of generalized matching and the algorithmic optimization required for its efficient and robust implementation. A tool based on generalized matching has been implemented and tested on large examples of the MCNC'91 benchmark 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%).

1 Introduction

Multiple-output technology-independent logic optimization based on Boolean relations (BRs) is potentially more powerful than traditional single-output optimization[1]. Unfortunately, BR-based techniques are more computationally expensive than traditional approaches such as algebraic [2] or *don't care*-based [6, 3] techniques and for this reason they have not yet succeeded as practical optimization engines. In [4] we introduced the concept of *generalized matching* (GM), a multiple-output Boolean matching technique which allows concurrent matching of two or more single-output library cells (or of one multiple-output cell) with a multi-output Boolean function. Generalized matching extends the Boolean relation-based approach to the technology dependent part of the synthesis flow.

Generalized matching, as formulated in [4], has the same drawback of its technology-independent counterpart, namely the high computational complexity. Hence, the formulation presented in [4] is very computationally demanding. In this work we introduce a new theoretical framework and algorithmic refinements that enable the application of GM to large circuits. We move from the observation that speed is usually the primary concern in synthesis. The timing budget for combinational logic is obtained from architectural specification. The first objective of synthesis is to produce an implementation that meets the timing constraints, and then to optimize secondary cost functions such as power or area.

When speed is the primary objective, two logic optimization problems have practical relevance: *unconstrained timing optimization* and *optimization of a secondary objective 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 re-design or re-partition 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 apply GM in the last stages of the synthesis process, namely we target the incremental optimization of a *mapped netlist*. Post-mapping optimization is also known as *re-mapping*. Our starting point is a netlist that has already been optimized by traditional synthesis techniques [5] for maximum speed with area recovery. Re-mapping is applied to either increase speed, or reduce area/power without decreasing speed.

The main theoretical contribution of this paper is the formulation of an algorithm for the solution of the minimum-cost (area or power) timing-constrained generalized matching problem. Additionally, 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 multi-level benchmarks in the MCNC'91 [15] suite. We obtain sizable improvements in speed, area under speed constraints and power under speed constraints. Moreover, re-mapping is most effective on larger netlists.

The paper is organized as follows. We first describe the rationale and high-level flow of our re-mapping approach (Section 2). In Section 3 we describe in detail the timing-constrained area minimization problem. In Section 4 we present experimental results. Constrained power minimization [16] and unconstrained speed optimization are not described in this paper (although we do provide experimental results) for space reasons.

2 The *re-mapping* approach

State-of-the art synthesis tools adopt a three-phase approach to logic optimization. First *technology-independent optimizations* are applied; then the network is mapped to the technology library of choice (*library binding*); finally the mapped netlist is further optimized by iterative algorithms that perform local transformations. We call *re-mapping* the last step [7, 8, 9]. Some re-mapping approaches [10] focus on changing the connectivity of the netlist in such a way that some gates either become redundant (and can be removed) or become sub-optimal (and can be replaced). Re-mapping transformations based on changes of the network connectivity are often called *re-wiring*.

We adopt a re-mapping 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. Once a target region is selected, it is optimized and, if optimization is successful, one or more cells are replaced with lower cost alternatives. Moreover, local wiring can be changed: the new cells may have different inputs. Hence, local re-wiring is performed as well.

Given a mapped network, the high level flow diagram of the re-mapping procedure is shown in Figure 1. Static timing analysis is performed and required times, arrival times and slacks

*Work partially supported by NSF contract MIP-9421129

†The author is now with Synopsys-Europe

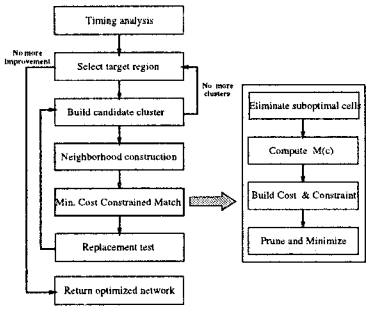

Figure 1: High-level flow of the re-mapping procedure

are computed for all nodes in the network. Then a breadth-first iteration on *target regions* of the network is started. From each target region, *candidate multi-output clusters* are generated and their neighborhood is analyzed to construct a Boolean relation representing the degrees of freedom available for the remapping of the cluster. Matching starts by eliminating library cells that are certainly sub-optimal, then the GM problem is solved by constructing a *matching function* M whose ON-set implicitly represents all possible replacements for cells in the candidate cluster.

The symbolic representation of timing constraint and area (or power) cost are built and the ON-set of M is first pruned by eliminating replacements that violate the constraints, then the minimum-cost replacements are selected from the pruned matching function. Replacement is performed if there is improvement with respect to the initial mapping. The breadth-first iteration is repeated until no further improvements are achieved.

A good choice of the target regions in the mapped netlist is paramount for the success of the re-mapping 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.

For extensive cost functions (area and power), the target regions are the *multiple fanout points* (MFPs) of the network. There are two main reasons for this choice. First, traditional library binding algorithms follow a *cone-based* paradigm [11]. A very efficient search of the optimum mapping is performed on fanout-free regions of the circuit, but the search stops when MFPs 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, it is 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.

Given a MFP, candidate clusters are generating by selecting sets of gates whose inputs or outputs are directly connected to the MFP. Although arbitrarily large clusters could be selected, the run time and memory requirements rapidly increases with the number of inputs and outputs in the cluster. For the sake of robustness and to keep run time under control we conservatively set the cluster size to two for all experiments reported in the paper. The selection of which cells to include in the clusters is decided by a *sliding window* algorithms (omitted for space reasons).

3 Min-cost Constrained Matching

In this section we focus on constrained minimization. Speed is the constraint and area is the optimization target. The clustering algorithm constructs candidate clusters starting from MFPs. Given the cluster we: i) build the Boolean relation that represents the degrees of freedom for matching created by the cluster's neighborhood; ii) perform minimum-area matching and guarantee that timing constraints are not violated. In the next subsections we describe how the two tasks are carried out. First we focus on the construction of the Boolean relation. Second, we provide the theoretical formulation of the generalized matching problem and we show how it can be solved efficiently. Third, we extend GM to take into account a cost function (area) and constraints (on timing) and we show how area can be minimized within timing constraints in computationally efficient fashion. In the following treatment we assume that the reader is familiar with Boolean algebra, BDD and ADD manipulation [6, 13].

3.1 Building the Boolean Relation

Let us consider a multi-output cluster function $\mathbf{f}$ embedded in a logic network as shown in Figure 2.

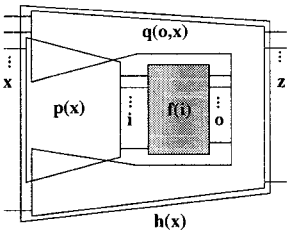

Figure 2: A multi-output cluster function embedded in its neighborhood

We adopt a formalism similar to that used by Watanabe et al. in [12]. We call $\mathbf{x}$ and $\mathbf{z}$ the vectors 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: $H = \prod_j h_j(\mathbf{x}) \overline{\oplus} z_j$, $P = \prod_j p_j(\mathbf{x}) \overline{\oplus} i_j$ and $Q = \prod_j q_j(\mathbf{o},\mathbf{x}) \overline{\oplus} z_j$. The characteristic functions fully describe the neighborhood around the multi-output function $\mathbf{f}$. In particular, they enable the computation of a Boolean relation representing the maximum 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 [12]:

$$\mathcal{F}(\mathbf{i},\mathbf{o}) = \forall_{\mathbf{x},\mathbf{z}}[(P(\mathbf{x},\mathbf{i}) \cdot Q(\mathbf{o},\mathbf{x},\mathbf{z})) \Rightarrow H(\mathbf{z},\mathbf{x})] \qquad (1)$$

In words, $\mathcal{F}$ represents the set of values of $\mathbf{i}$ and $\mathbf{o}$ such that if Q is true and P is true, then H is true for all possible values of $\mathbf{x}$ and $\mathbf{z}$. Formula (1) allows us to find all functions $\mathbf{f}$ that, when composed with $\mathbf{p}$ and $\mathbf{q}$, produce exactly function $\mathbf{h}$. 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.

Having shown how $\mathcal{F}$ can be computed when the neighborhood $\mathbf{h}$ is given, we need to clarify how $\mathbf{h}$ is selected. 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 [12]. Unfortunately this is computationally infeasible but for the smallest networks. Thus, the neighborhood has to be a small subset of the logic network, 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 $\mathbf{f}$ that gives more degrees of freedom than computing separately their *don't cares*.

Consider a two-output cluster function. To obtain an advantageous Boolean relation, we need to find nodes in the fanout cone and fanin cone of both outputs of $\mathbf{f}$. 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 $\mathbf{f}$. If fanin and fanout cones of the components of $\mathbf{f}$ are disjoint, $\mathcal{F}$ represents the same degrees of freedom that can be expressed by *don't cares*. Since we consider clusters starting from MFPs, 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.

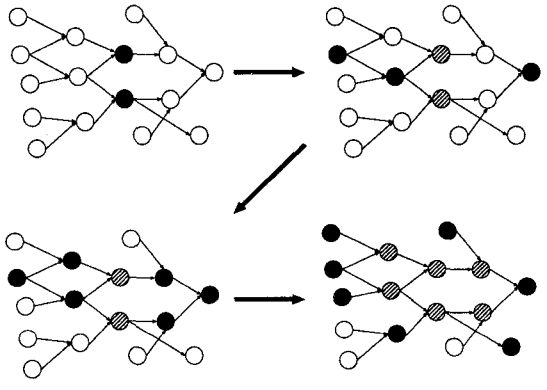

Figure 3: Building the neighborhood of a cluster.

Example 1 We show on Figure 3 how the neighborhood construction 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 Figure 3. The parameter depth is set to 2. 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 Figure 3.

Given the neighborhood, the Boolean relation $\mathcal{F}$ is obtained by Equation 1. We build the BDDs of the Boolean relations P, H, and Q by traversing the neighborhood. $\mathcal{F}$ is then computed using Equation 1. The overall complexity depends on the computation of the relation H. Our neighborhood construction algorithm has been designed to minimize the complexity of the computation of H, while at the same time to obtain a final $\mathcal{F}$ expressing useful degrees of freedom.

3.2 Generalized Matching

The GM problem consists of finding all possible sets of n library cells that can implement one of the functions represented by $\mathcal{F}$ [4]. To accomplish this task we define the concept of *quotient function* $L(\mathbf{i},\mathbf{c})$ for our technology library. The pictorial representation of the quotient function is shown in Figure 4 for a simple library with $N_{lib} = 3$ cells, g_1, g_2 and g_3.

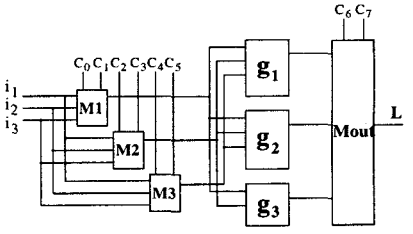

Figure 4: Quotient function of the target library.

In the figure, the blocks $M1$, $M2$, $M3$ and $Mout$ represent multiplexers, with control inputs $\mathbf{c} = [c_0, c_1, ..., c_7]^T$. 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 $Mout$ controls cell selection: it selects which cell is connected to the output. For the sake of simplicity, we assume single-output library cells (as in Figure 4) in the following treatment. GM is applicable to multi-output cells as well [4].

In order to perform generalized matching, we need to check if a two-output cluster function $\mathbf{f}(\mathbf{i})$ can be replaced by two library cells. 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 in L and $\mathcal{F}$ [4]:

$$M(\mathbf{c}) = \forall_{\mathbf{i}}\exists_{\mathbf{o}} \left(\mathcal{F}(\mathbf{i},\mathbf{o}) \cdot (L(\mathbf{i},\mathbf{c}_1) \overline{\oplus}\ o_1)(L(\mathbf{i},\mathbf{c}_2) \overline{\oplus}\ o_2)\right) \qquad (2)$$

Where $\mathcal{F}$ is the Boolean relation for the cluster, L is the quotient function. Notice that for each output, we have distinct sets of control variables, 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 *matching function* and can in principle be computed by simply implementing Equation 2 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 neighborhood boundary. In other words, Equation 2 allows us to compute *all* cell selections and input assignments compatible with $\mathcal{F}$. If we replace the pre-existing implementation of **f** with any implementation in the ON-set of M, we are guaranteed that the input-output behavior of **h** is unchanged (more details on this can be found in [4]). Each minterm of $M(\mathbf{c})$ represents a solution. The assignments are easy to analyze, because they correspond to multiplexer selections.

The main practical problem in the computation of $M(\mathbf{c})$ 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 Equation 2. 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's 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 BDDs in Equation 2 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. The matching function of output o_1 can be computed by the following formula:

$$M_1(\mathbf{c}_1) = \forall_{\mathbf{i}} \exists_{\mathbf{o}} \left(\mathcal{F}(\mathbf{i}, \mathbf{o}) \cdot (L(\mathbf{i}, \mathbf{c}_1) \overline{\oplus} o_1) \right) \tag{3}$$

The same formula holds for output o_2 (after a change of indices from 1 to 2). Computing M_1 and M_2 separately can be much easier than computing M, because the BDDs 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 a function that cannot be implemented by any cell in the library are in the ON-set of M_1 (M_2) but are *not* in the ON-set of M. A match can be in the ON-set of M only if it is valid for *both* outputs, while a match is in the ON-set of M_1 (M_2) simply if it is valid for f_1 (f_2), no matter what happens to f_2 (f_2). 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* 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 Equation 2. In other words we can replace $L(\mathbf{i}, \mathbf{c}_1)$ in Equation 2 with $L_{res}(\mathbf{i}, \mathbf{c}_1)$ defined as follows (the same can be done for $L(\mathbf{i}, \mathbf{c}_2)$):

$$L_{res}(\mathbf{i}, \mathbf{c}_1) = \begin{cases} L(\mathbf{i}, \mathbf{c}_1) & \text{if } M_1(\mathbf{c}_1) = 1 \\ \textit{don't care} & \text{otherwise} \end{cases} \tag{4}$$

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. Our bound is conservative, and the matching function is computed exactly: $M(\mathbf{c})$ still gives us all possible assignments for the cluster. This method can be used for more than two input clusters. ,

3.3 Cost Function and Constraints

Until now, we have proposed a solution to the basic GM problem. Now we focus on its useful extension by considering cost functions and constraints. We assume that *area* is the cost function and *timing* is the constraint. Although we have a way to compute all possible legal replacements for **f** , we want get the minimum-area matchings 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.

Our minimum-cost constrained matching algorithm merges GM with cost minimization and constraint enforcement. We exploit the existence of a previous implementation (since we are re-mapping) 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. We will first describe how bounding can improve the performance of our algorithm, then we will show how to find all minimum area matches, finally we describe how to prune solutions that violate the timing constraints.

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. 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 $A < A_{old} - A_{MIN}$, where A_{old} is the area of the current implementation of **f** and A_{MIN} is the smallest area of any cell in the library.

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 Equation 2. Consider area first. We call $A_{MIN,1}$ the minimum area of any match in M_1. All matches in M_2 such that area $A_2 \geq A_{old} - A_{MIN,1}$ can be pruned, because the total area of a solution involving them is certainly larger than A_{old}. The same reasoning can be done for M_1, A_1 and $A_{MIN,2}$.

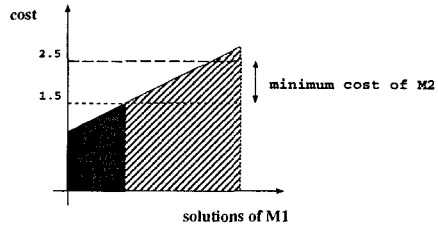

Figure 5: Bound on the partial solutions.

Example 2 Consider the example in Figure 5. 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 re-mapping. We assume that the minimum cost for M_2 (not shown) is 1. 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, the line of reasoning is similar, but more involved. Although we will discuss timing computation in greater detail, we just observe for now 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 cause on the fanin gates. We call $D_{Bound,2}$ 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 $D_{Bound,2}$ exceeds the timing constraint, the match can be pruned. The same reasoning holds for M_1 and $D_{Bound,1}$.

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 to 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 Equation 2.

Once $M(\mathbf{c})$ has been computed, we want to find the minimum-area assignment that satisfies the timing constraints. One solution would be to enumerate all the minterms of $M(\mathbf{c})$ and evaluate them for area and timing. This is not acceptable because the number of minterms of M can be large and an enumerative solution would be unacceptably slow.

A more efficient solution is to use ADDs [13] to build symbolically the cost function and the constraints. ADDs are appropriate because they represent in a compact way discrete functions, and they interface seamlessly with BDDs (they have the same structure, the only difference being that leaves can have any value).

To compute minimum-area 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 area of the cell identified by the values of control variables encountered on the path. Once the ADD $A(\mathbf{c})$ of the cost function is build, 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). Since the area of a cell is not affected by input assignment, the support of $A(\mathbf{c})$ contains only variables controlling the cell selection in the quotient function.

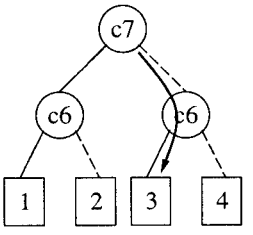

Figure 6: ADD cost function for area for 4 variables

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 Figure 6. The control variables for cell selection are c_7 and c_6. For instance, $c_6\bar{c_7}$ selects the NAND3 gate. In the ADD of the cost function, we see how the path with c_6 and $\bar{c_7}$ leads to the cost of the NAND3, i.e. 3. 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 3, and the value $\mathbf{c}^*$ of the control variables for which $A(\mathbf{c}) \cdot M(\mathbf{c})$ is minimized is $\mathbf{c}^* = c_0c_1c_2c_3'c_4'c_5c_6c_7$.

The cost function for area can be computed once for all. Its number of nodes is very small, it's bounded by $2 \times |N_{lib}|$. 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 $O(|M(\mathbf{c})| \cdot |A(\mathbf{c})|)$. $|M(\mathbf{c})|$ is the number of nodes in the BDD of the matching function, $|A(\mathbf{c})|$ 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 in 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.

After finding the minimum area matches, the last step of the algorithm is to enforce the timing constraints. Constraints can be manipulated in a symbolic fashion as well. Before describing their 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.

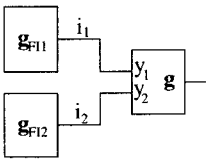

Figure 7: Delay computation for a gate

We use the *real delay model* as in SIS [5]. Consider a gate g with input pins $\mathbf{y} = [y_1, y_2, ..., y_n]^T$ shown in Figure 7. The pins are connected with inputs $\mathbf{i} = [i_1, i_2, ..., i_n]^T$. We assume that pin y_i is connected to input i_i. The arrival time at the output of the gate is:

$$t_{arr} = \max_{i=1,2,...,n} (t_{arr_i} + \alpha \times C_L + \beta_i) \tag{5}$$

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_{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_i:

$$t_{arr_i} = K_i + \eta_i(C_{other_i} + C_{in_i}) \tag{6}$$

η_i is the effective output resistance of the fanin gate g_{FIi}, K_i is the part of t_{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_{arr_i} changes,
- the intrinsic delay β_i changes because it is pin-dependent as well

In the quotient function $L(\mathbf{i},\mathbf{c})$, input assignments are set by the control variables. The quotient function represents a set of cells and input assignments. Each one of such assignments is characterized by an arrival time to the output of the gate it represents. Hence, t_{arr} for a quotient function is a function of the control variables. In other words, for a cluster output that we want to match we can build the ADD $T_{arr}(\mathbf{c})$ 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 $T_{arr}(\mathbf{c})$ 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.

The computation of T_{arr} is complicated by the fact that we are concurrently matching a multi-output cluster function using multiple quotient functions. The complication arises when we compute the arrival time t_{arr_i} at the inputs i_i of the cluster. Remember that t_{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: $T_{arr_i}(\mathbf{c}) = K_i + R_i(C_{other,i} + C_i(\mathbf{c}))$, where K_i, R_i and $C_{other,i}$ 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_{arr_i} depends on the entire $\mathbf{c}$. If we are matching a two-output cluster, T_{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 Equation 2). The computation of the arrival time at each output of the cluster is done with the following symbolic formula.

$$T_{arr}(\mathbf{c}) = \max_i(T_{arr_i}(\mathbf{c}) + \alpha(\mathbf{c}) \times C_L + \beta(\mathbf{c})) \qquad (7)$$

Where T_{arr_i}, α and β are ADDs 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.

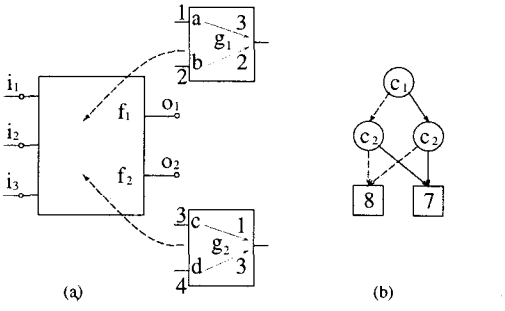

Figure 8: Symbolic representation of timing constraints

Example 4 Consider the situation shown in Figure 8: we want to compute the ADDs $T_{arr}(\mathbf{c})$ for the two outputs of the cluster of Figure 8 (a). We make several simplifying assumption for the sake of clarity. First, we assume that $K_i = 0$ and $C_{other_i} = 0$ 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 Figure 8 (a). 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 $c_1 = 0$, $c_2 = 0$. The arrival times at the inputs are $T_{arr_i}(0,0) = C_{in_1}(0,0) = 1$, $T_{arr_i}(0,0) = C_{in_2}(0,0) = 2 + 3 = 5$, $T_{arr_i}(0,0) = C_{in_3}(0,0) = 4$. For output o_1 the arrival time is $T_{arr}(0,0) = max\{(3+1),(2+5)\} = 7$ this is one leaf of the ADD $T_{arr}(c_1,c_2)$ representing the arrival time at o_1 for every combination of control variables. The complete ADD is shown in Figure 8 (b).

The algorithm for the computation of the constraint ADD is quite involved and it is not described for space reasons. The ADD of timing constraints is used to prune the matching function. This is done three times during GM. The first two times the timing constraint ADDs for M_1 and M_2 are computed and used for bounding, as discussed at the beginning of this subsection. The last time, after M has been computed, with the purpose of eliminating infeasible solutions.

3.4 The Complete Matching Algorithm

Having described all sub-tasks involved in performing minimum-cost constrained matching, we conclude our analysis by describing the complete matching algorithm. The pseudo-code of the algorithm for minimum-area matching under timing constraints is shown in Figure 9.

```
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 ∈ outputs (clusterF))
    bdd_m_only[o] = matching_func (o, bdd_quot);
  /* bounding with area and timing constraints */
  foreach (o ∈ 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 = matching_func (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);
```

Figure 9: Algorithm of the matching step

The algorithms first finds the neighborhood of the cluster function, and computes the Boolean relation `bdd_rel`, as seen in Subsection 3.1. Then the quotient functions are constructed disregarding the library cells whose area cannot improve the current solution, as seen in Subsection 3.3. The single-output matching functions are then computed by `matching_func`, and pruned using area bounds and timing constraints, as discussed in the second part of Subsection 3.3. 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 violat-

ing the constraint are eliminated. Finally, the cost function is applied and the subset of the ON-set of $M(c)$ 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 Figure 9 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 $M(c)$ after bounding and several corner conditions are flagged to speedup the computation of trivial cases.

4 Results

We have implemented a post-mapping 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 0 to infinity. Specifying a depth of 0 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 of a cluster can be also controlled. We made experiments with up to four outputs. The number of inputs `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 re-wiring.

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 which 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 tools uses the `Cudd` BDD package [14] which provides a rich set of operators on BDDs and ADDs 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 BDDs 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 greater than 8. 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 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. For clusters with four or more outputs, memory blowup is too frequent to be acceptable. In order to improve the chances of success for 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 matches. We do not describe the implementation of such aggressive bounds for space limitations.

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. For this reason 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 it when the peak is reached.

We have experimented our tool with a set of combinational MCNC benchmarks [15] including all larger ones. The benchmarks were first optimized with SIS [5] for minimum delay with area recovery, with script `script.delay` followed by the mapping command `map -n 1 -AFG`.

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 3. We used a library based on an industrial technology file, with 75 cells, with up to five inputs. The number of inputs of the cluster was limited to 10.

We show in Table 1 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 instances, the percentage gain and the run time in minutes (on SPARC20 with 256 Mb of memory) for each kind of optimization. The 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 trade off 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 MFPs and reconvergent fanout cones. Both these characteristics increase the effectiveness of our optimization tool. Notice also that very good improvements are obtained for the larger benchmarks. We conjecture that the global optimization of SIS is less efficient for large benchmarks, and re-mapping can recover a big fraction of the optimality loss.

The run times of the re-mapping tool are shorter (but of 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 in 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 2 provides detailed information on the trade-off involved in the optimization process. The first column contains the benchmark name. The second and the 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 doing power optimization (again, speed is the constraint and no tradeoff 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 a usually a small fraction of the entire circuits. Only few 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 in 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.

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 2: Trade offs in optimizations.

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	

Table 1: Results on MCNC benchmarks.

5 Conclusions

In this paper we proposed a re-mapping 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. 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. Hence, our tool has general applicability.

References

[1] F. Somenzi et al., "Minimization of Boolean relations," in *ISCAS*, pp. 738–473, 1989.

[2] R. Brayton et al., "Multilevel logic synthesis," *IEEE Proceedings*, vol. 78, pp. 264–300, 1990.

[3] H. Savoj et al., "Extracting local don't cares for network optimization," in *ICCAD*, pp. 514–517, 1991.

[4] L. Benini et al., 'A survey of Boolean matching techniques for library binding," *TODAES*, vol. 2, n. 3, 1997.

[5] E. Sentovich et al., "Sequential Circuits Design Using Synthesis and Optimization," in *ICCD*, pp. 328-333, 1992.

[6] G. De Micheli. *Synthesis and optimization of digital circuits.* McGraw-Hill, 1994.

[7] K. Cheng et al., "Multi-level logic optimization by redundancy addition and removal," in *Euro-DAC*, pp. 373–377, 1993.

[8] W. Kunz et al., "Multi-level logic optimization by implication analysis," in *ICCAD*, pp. 6–13, 1994.

[9] B. Rohfleisch et al., "Logic clause analysis for delay optimization," in *DAC*, pp. 668–672, 1995.

[10] S. Chang et al., "Fast Boolean optimization by rewiring," in *ICCAD*, pp. 262–269, 1996.

[11] K. Keutzer, "DAGON: technology binding and local optimization by DAG matching," in *DAC*, pp. 341–347, 1987.

[12] Y. Watanabe et al., "Permissible functions for multioutput components in combinational logic optimization," *IEEE TCAD ICAS*, vol. 15, no. 7, pp. 734–744, 1996.

[13] R. Bahar et al., "Algebraic Decision Diagrams and their Applications," in *ICCAD*, pp. 188-191, 1993.

[14] F. Somenzi. *The CUDD package User's guide. Version 1.0.5* 1995.

[15] S. Yang, "Logic Synthesis and Optimization Benchmarks User Guide Version 3.0," *Tech. Rep. MCNC*, 1991.

[16] P. Vuillod et al., "Re-mapping for low power under tight timing constraints," *ISLPED*, 1997.