A Methodology and Algorithms for Post-Placement Delay Optimization

Lalgudi N. Kannan*
Escalade Corp.
1210 East Arques Avenue
Sunnyvale, CA 94086

Peter R. Suaris*
Interconnectix, Inc.
10220 SW Nimbus Avenue
Portland, OR 97223

Hong-Gee Fang
Cadence Design Systems, Inc.
2655 Seely Road MS 6B1
San Jose, CA 95134

Abstract

In this paper we present a placement-intelligent resynthesis methodology and optimization algorithms to meet post-layout timing constraints while at the same time reducing the interconnect congestion. We begin with the synthesized design netlist after initial placement and make incremental modifications – taking placement into account – to generate a final netlist and placement that meets the delay constraints after place and route. The algorithms described have been implemented as part of a tool for placement based resynthesis. The tool has been used with a number of pre-optimized designs from industry and to obtain improvements in post-placement delays ranging from 13 to 22% with improved routability.

1 Introduction

The goal of the ASIC design process is to realize a design as a gate array or standard cell array which satisfies a set of user/technology defined constraints such as area, delay, maximum load/maximum fanout, and hold time. Conventional ASIC design methodologies typically consist of a synthesis phase wherein the design is synthesized, optimized, and mapped to a target library, followed by placement and routing phase.

During traditional logic synthesis, circuits are optimized to meet user-defined timing and area constraints. The timing measures used during this stage of the process consist mainly of gate delays and rough approximations for interconnect delay - usually a fanout based value, with the wiring parasitic capacitance related to the fanout of the net using a piecewise linear model. As illustrated in Fig. 1, such an approximation can be grossly inadequate to predict the real interconnect delay after layout. Cells X and Y both drive nets with 3 fanouts. However it is easy to see that the actual loads are very different, since cell X drives a much longer net.

*Work done while at Cadence Design Systems, Inc., 2655 Seely Road, San Jose, CA 95134

With today's submicron design processes the wiring parasitic becomes a dominant factor in the total delay of a timing path [1]. It is predicted that for a 0.5 micron process, the wiring can contribute as much as 60% of the total delay. When the wiring effect is dominant, traditional synthesis tools that use a fanout-based model may be optimizing a timing value which is significantly different from the actual post layout value. Another problem with traditional synthesis is in area estimation. Typically, the tools try to optimize the total gate area, and the interconnect area and the routability of the chip are not taken into account. As a result, although the total gate area of the synthesized netlist is quite small, it may not fit into the assigned die area after layout.

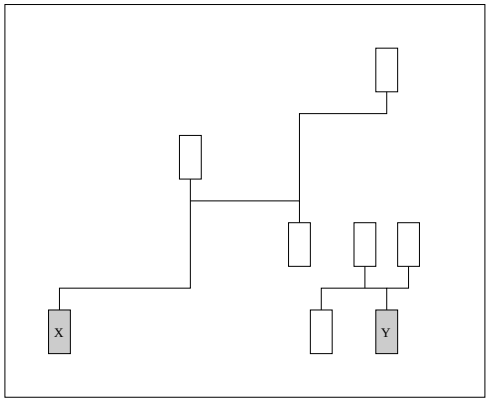


Figure 1: Limitations of fanout-based wiring models

1.1 Previous/Related Work

The problem of fusing layout considerations into the logic synthesis process has been addressed in a few papers in recent years [2], [3], [4], [5]. Abouzeid et al. [2] describe an algebraic factorization method based on lexicographic expressions of boolean functions to reduce both gate and wiring areas. It attempts to express the set of logic functions in a form that will result in layered structured cones with ordered input injection after technology mapping. Pedram and Bhat [3] describe a method in which a fast global placement is done of the logic network (decomposed into a canonical graph of 2-input NAND gates and inverters), and this placement information is used to guide the wiring estimation as well as the technology mapping. The placement information is updated dynamically so that the nodes processed later can use more accurate information. This

approach is extended in [4] to include the logic restructuring and the technology decomposition phases of logic synthesis. This is further extended to the post-mapping phase in [5]. Here, a fanout optimization algorithm based on alphabetic trees is presented that generates fanout trees free of internal edge crossings thus improving routing area.

However, all these methods suffer from the fact that they work on layout measures that can be far removed from the actual layout. It is extremely difficult to accurately predict the post-layout interconnect lengths during the synthesis process. Realistic estimates of interconnect can be obtained only after actual placement is done. To obtain good results, synthesis tools must work on the post-layout results. Traditionally the approach has been to extract the interconnect parasitics after place and route and backannotate them to the synthesis tool. The synthesis tool then makes changes to the netlist and passes the changes to the layout tool. If these changes are large scale, major changes in the placement will result, rendering the backannotated values meaningless. Hence this approach will give rise to multiple iterations, with no guarantee that it will converge (see Fig. 2a). Partly due to this problem, permissible modifications to the netlist currently have been very restricted (usually limited to resizing of buffers). As wiring delays become more significant with higher congestion, the resynthesis process can alleviate many of the post-layout problems if it can take placement information into account.

1.2 Our Approach

In this paper we present a *placement-intelligent* resynthesis process that can improve post-layout timing while improving the interconnect congestion. Fig. 2 contrasts the conventional ASIC Design flow with our proposed approach using placement-based synthesis.

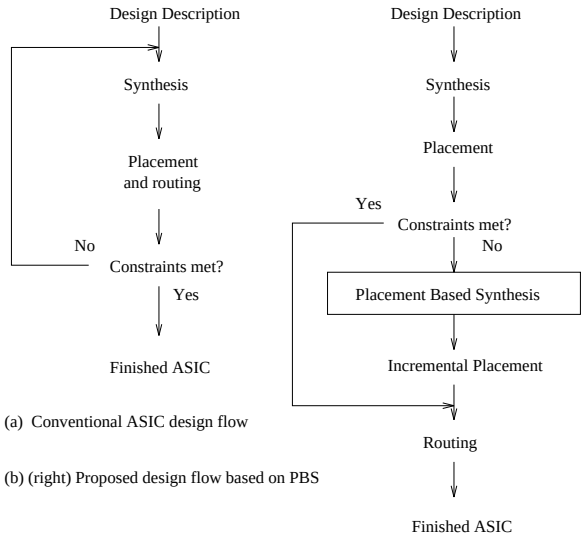


Figure 2: Comparison of ASIC Design Flows

In the proposed flow (Fig. 2b), after synthesis of the design is done, a placement is obtained. The synthesized netlist, along with this initial placement, becomes the input to the placement-based resynthesis tool. A placement-based timing analysis of the design is used to direct the resynthesis process, which attempts to improve delay, meet all design constraints, and alleviate congestion. The output from the resynthesis phase is a modified netlist and suggested locations for the newly added components. The incremental place and route tool derives a placement that conforms as much as possible to the suggested locations. The changes done by the resynthesis process are incremental in nature and the chances of converging to a placement close to the suggested locations are very high.

The techniques and algorithms described have been incorporated into a system for placement-based synthesis, and the results have been very encouraging. Our experiments show a 13-22% improvement in the post-layout delays as well as improved routability.

The rest of the paper is organized as follows. In Section 2 we present a scheme to do placement-based timing analysis along with an efficient method to compute the wiring delays. In Section 3 we describe some transformations and algorithms to do post-placement delay optimization and to improve routability. In Section 4 we describe how these algorithms have been incorporated into a complete system for doing placement-based resynthesis. We present some experimental results in Section 5 and concluding remarks in Section 6.

2 Placement-based Timing Analysis

The placement-based timing analyser computes the gate and wiring (interconnect) delay, calculates the minimum and maximum arrival/required times at all points in the network and determines the critical paths. The information from timing analysis is used to generate delay reports as well as to direct the optimization algorithms.

Our delay calculations are based on the following equation for the delay of a gate [1]:

$$D = D_I + D_L + D_W + D_S$$

D_I is the intrinsic gate delay. D_L is the load delay due to the loading at the output pin of the current gate given by

$$D_L = R_{drive}(C_{pins} + C_{wire})$$

where R_{drive} is the drive resistance of the gate, C_{pins} is the total pin capacitance, $C_{pins} = C_{out} + \sum_i C_i$, and C_{wire} is the estimated wiring capacitance. D_W is the interconnect or wiring RC delay. D_S is the input slew delay due to slowly changing input signals given by $D_S = SD_{Lprevious}$ where S is the slew sensitivity factor of the current stage and $D_{Lprevious}$ is the load delay of the previous stage.

The estimates for the wiring parasitics (C_{wire}) are computed from the placement locations of the components as described in the next section.

2.1 Wire load computation

In order to compute the parasitics for a net, it is assumed that the terminals are connected as a spanning tree.

Let S be a set of terminals that are connected by a net. Let the coordinates of terminal i be denoted by (x_i, y_i). Let d be the driving terminal of the net ($d \in S$) and let C_i be the input capacitance of terminals i. N and P are two sets of terminals. C_h and C_v are capacitances per unit length for the horizontal and vertical routing layers respectively.

The procedures COMPUTE-WIRE-LOAD and COMPUTE-NODE-LOAD are used to perform the load computation. The downstream capacitance of each node of the spanning tree, which is the capacitance of the subtree of the spanning tree rooted at the node, is also computed and stored. In fact the downstream capacitance of a node is the total wire capacitance of a net connecting that node and all its children. These downstream capacitances will be used extensively during placement-based delay optimization as described in later sections.

The procedure COMPUTE-WIRE-LOAD(see Fig. 3) also updates the wireload at each node in the spanning tree, and this is used by the subsequent procedure COMPUTE-NODE-LOAD.

```
COMPUTE-WIRE-LOAD(S)
    N ← S − {d}, P ← {d}
    L_h ← L_v ← 0, C_wire(d) ← 0
    setNearestTerm(d) ← NIL
    foreach term i in N
        cost(i) ← L_manhattan(d, i)
        setNearestTerm(i) ← d

    while |N| > 0
        find term m such that cost(m) is minimum
        n ← getNearestTerm(m)
        l_h ← |x_m − x_n|, l_v ← |y_m − y_n|
        L_h ← L_h + l_h, L_v ← L_v + l_v
        c_wire(n, m) ← l_h C_h + l_v C_v
        P ← P ∪ {m}, N ← N − {m}
        foreach i ∈ N
            if cost(i) > L_manhattan(i, m)
                cost(i) ← L_manhattan(i, m)
                setNearestTerm(i) ← m
    C_wire(d) ← L_h C_h + L_v C_v

COMPUTE-NODE-LOAD(v)
    S ← succ(v)
    if S = φ
        C(v) ← C_in(v)
        return C(v)
    C(v) = 0
    foreach i ∈ S
        C(v) ← C(v) + c_wire(i, v)+COMPUTE-NODE-LOAD(i)
    C(v) ← C(v) + C_in(v)
    return C(v)
```

Figure 3: Procedures to compute the wire load and the node load

Let us denote the MST computed by COMPUTE-WIRE-LOAD as $T(d)$. Any subtree of $T(d)$ rooted at a node v is denoted by $T(v)$. Let $C_{T(v)}$ denote the total capacitance of the subtree $T(v)$. (including the input capacitance of the node) We can then compute $C_{T(v)}$ for every node v using the procedure COMPUTE-NODE-LOAD (see Fig. 3). Here $C_{wire}(v)$ is the wire load at v computed from COMPUTE-WIRE-LOAD, and $C_{in}(v)$ is the input pin capacitance at v.

3 Placement-based delay optimization

This is done by applying a number of transformations on the critical path of the circuit to reduce the critical path delay. A number of techniques have been described in the literature [6], [7], [8]. However, none of these methods explicitly take the placement into account. In our work we choose delay-reducing transformations which take the placement into account and behave in a manner that will only slightly perturb the intial placement while considerably improving the routability.

As a preliminary step to applying the transformations we strip off all the buffers and merge all redundant inverters introduced by the initial synthesis run. This will allow us to do a placement-based buffer insertion from scratch taking true wire parasitics into account. One might wonder, if we are going to strip off all the buffers during resynthesis, why introduce them in the first place? We have found that by including buffers in the initial placement, we can minimize changes in the net area of the components in the netlist. This improves chances that the incremental placer will find a solution close to the suggested placement.

The transformations are described below in detail along with their placement and routing implications.

3.1 Fanout buffering

In this technique, the output of a cell driving a large number of fanouts is selected, and a buffer is inserted to buffer a portion of the fanouts to minimize the delay. The approach we use is based on the fast heuristic algorithm described in [8]. Fig. 4 illustrates the basic transformation. The fanouts moved over to the output of the inserted buffer are typically non-critical ones.

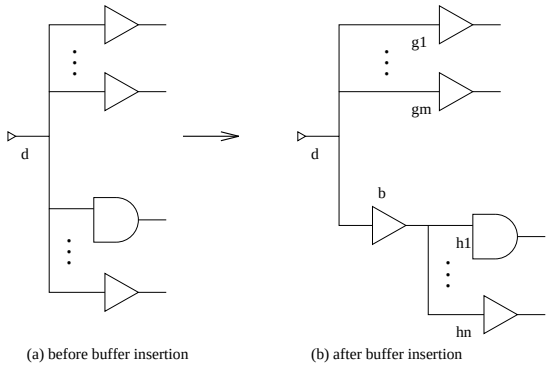


Figure 4: The fanout buffering transformation

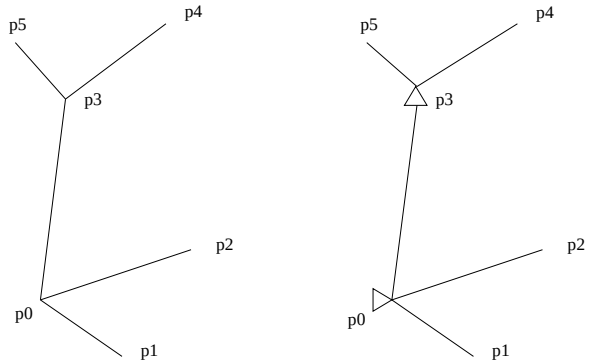
As an additional requirement, a solution is desired that would improve the routability of the circuit. Keeping in mind that the wiring delay can be quite significant when compared to the gate delays, we formulate the solution to the problem as follows.

Consider a net consisting of a source node p_0 and a set of n destination terminals $\{p_1, p_2, \ldots, p_n\}$. Let the location of each terminal p_i be given by (x_i, y_i). Let the required arrival time at terminal p_i be given by r_i and the actual arrival time be given by a_i. The problem is to find a buffer configuration for the net with locations for the buffers such that

$$\max_{i \in \{1 \ldots p_n\}} \{a_i - r_i\}$$

is minimized.

Fig. 5a shows a net composed of a source terminal p_0 and destination terminals $p_1, p_2, \ldots, p_5$. The buffer configuration that meets the timing constraints may look like the one given in Fig. 5b. In this case two additional buffers are introduced near the locations (x_3, y_3) and (x_0, y_0).



(a) before fanout buffering (b) after fanout buffering

Figure 5: Buffering the net to improve delay

We assume the terminals in the net are connected as a *minimal spanning tree* (MST). We restrict the locations of the candidate buffers to the internal vertices of the minimal spanning tree. Since any subtree of a minimal spanning tree is also a minimal spanning tree [9], any subnets generated by the fanout buffering will constitute a subtree of the original MST. The loads of such subnets can be precomputed and stored at the nodes as described in Section 2.1.

These assumptions make it possible to have a computationally efficient solution to the problem. As an added advantage, this approach does not introduce any additional routing for the net. In other words, the routing wire length is almost the same as that of a single minimum spanning net connecting the terminals together. Thus by the act of removing all the buffers initially and reinserting them in a near optimal fashion, we achieve considerable improvements in routability. This can be clearly seen in Fig 11 which shows the buffered net before and after post-placement resynthesis.

The procedure (FANOUT-BUFFERING-TRANS) to do the fanout buffering transformation is given in Fig. 6. A set of terminals, M, is partitioned into two groups G and H, where H is buffered and G is not (see Fig. 4). L is the component library and $buf(\in L)$ is the buffer that is inserted. Let i be a terminal ($i \in M$) and let s_i be the slack $(r_i - a_r)$ at i. $T(i)$ is the minimal spanning subtree rooted at i and $C_{T(i)}$ is the load at i (see Section 2.1). R_d and R_{buf} are the drive resistances of the driver term and the output term of the inserted buffer respectively. C_{buf} is the input capacitance of the buffer and D_{buf} the intrinsic delay through the buffer.

```
FANOUT-BUFFERING-TRANS(d, M)
    buf ← select_suitable_buffer(L)
    S_old ← min(s_i | i ∈ M)
    break_term ← NULL
    foreach terminal i ∈ M and i not a leaf terminal
        S_G ← S_H ← ∞
        foreach terminal j ∈ M
            if j ∈ T(i)
                S_H ← min(S_H, s_j)
            else
                S_G ← min(S_G, s_j)
        S_G ← S_G + R_d C_T(i) − R_d C_buf
        S_H ← S_H + R_d C_T(i) − R_d C_buf − R_buf C_T(i) − D_buf
        S_new ← min(S_G, S_H)
        if S_old < S_new
            S_old ← S_new
            break_term ← i
    if break_term = NULL
        return FAILURE
    else
        insert buffer buf at break_term
        return SUCCESS
```

Figure 6: Procedure to do the fanout buffering transformation

The place and route tool will resolve the overlaps introduced by the addition of the cell with only minor perturbation of the cells in the local region.

3.2 Gate Resizing

In this technique, a cell is replaced with a cell in the library that is equivalent in functionality but having a better drive strength, intrinsic delay, load or other characteristic that makes it more appropriate [6]. This substitution is basically done in place, and any resulting overlaps would be minor and resolvable by the place and route tool.

Let P be the set of all the input terminals that are driven by the fanout of the driver gate d, and let Q be the set of all the input terminals that are driven by the fanins to the driver gate (see Fig. 7).

In the example shown $P = \{p_1, p_2, \ldots, p_m\}$ and $Q = \{q_1, q_2, \ldots, q_n\}$. It can be easily seen that replacing the gate d with another gate will only affect the slacks at these two sets of terminals and hence the slacks need only be recomputed at these terminals. The procedure for doing the gate resizing transformation is shown in Fig. 8. G_d is the set of gates in the library that are equivalent in functionality to d but with different delay characteristics.

The overall procedure to do the delay optimization is shown in Fig 9. N is the network, and C is a list of terminals that constitute the critical path. F represents

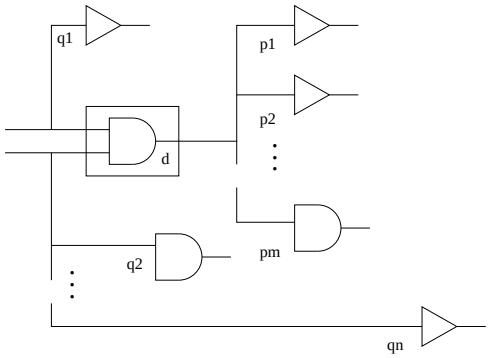


Figure 7: The gate resizing transformation

```
GATE-RESIZE(d, M)
    G_d ← get_equiv_gates(L, d)
    S1 ← min(s_i|p_i ∈ P), S2 ← min(s_i|q_i ∈ Q)
    S_old ← min(S1, S2)
    g_best ← NULL
    foreach gate g ∈ G_d(g ≠ d)
        S1' ← min(s'_i|p_i ∈ P), S2' ← min(s'_i|q_i ∈ Q)
        S_new ← min(S1', S2')
        if S_old < S_new
            S_old ← S_new
            g_best ← g
    if g_best = NULL
        return FAILURE
    else
        replace gate d with g_best
        return SUCCESS
```

Figure 8: Procedure to do the gate resizing transformation

the fanout buffering transformation, and R is the gate resizing transformation. The transformations are tried in turn, and whenever one is successfully applied, the timing trace is redone and the procedure works on the new critical path. The order in which the transformations are applied does affect the quality of the result, but only slightly. The procedure terminates when no transformation can be successfully applied or when a certain number of iterations (specified by the user) are completed.

4 A Placement Based Synthesis System

The algorithms developed in this paper have been incorporated into a system for doing placement-based resynthesis. Fig. 10 shows the steps in the placement-based resynthesis process. In addition to doing delay optimization with improved routability, it also attempts to satisfy a number of other design/technology constraints.

We start by stripping off all the buffers and removing all redundant inverter cascades introduced by the initial synthesis run. This will allow us to do a placement-based buffer insertion taking the true wire parasitics into account.

The next step is to do the placement-based delay optimization using fanout buffering and gate resizing transformations to reduce the delays along the critical paths in the network.

```
DELAY-OPTIMIZE(N)
    count ← 0, success ← TRUE
    while success = TRUE and count < MAX_ITER
        success ← FALSE
        do_timing_analysis(N)
        C ← get_critical_path(N)
        choose term i(∈ C) for applying transformation
        foreach transformation in (F, R)
            if transformation successful
                success ← TRUE
                count ← count + 1
                break
```

Figure 9: Procedure to do the delay optimization

Transformations to satisfy maximum load/maximum fanout constraints are then done. Most ASIC cell libraries have a maximum load/maximum fanout constraint on the output pins of the cells that specifies the limiting load or the maximum number of fanouts that it can drive. In order to satisfy these constraints, we need to satisfy the inequality $Maxload \geq \sum C_{in} + \sum C_{out} + C_{wire}$. Here again conventional synthesis tools may not be able to accurately estimate the wiring load, C_{wire}, so we use a set of placement based transformations to meet these constraints.

Next, we correct for holdtime violations. These are violations caused when the clock arrives too late at the clock pin of a flip-flop. This might be caused by the clock line being too long when compared to the data path. Again this would be known only when we consider the actual placement and accurately estimate the wiring delays. This is corrected by adding additional delays in the data path.

Finally the output netlist is generated along with suggested placement locations for the added components.

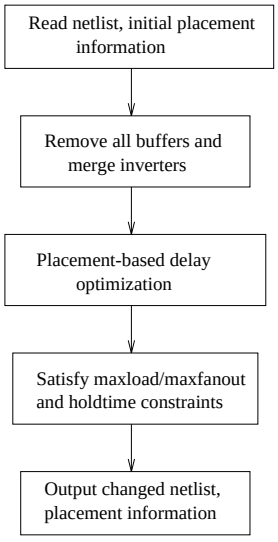


Figure 10: Steps in the placement-based resynthesis process

Figure 11: Highlighted buffered net before(top) and after(bottom) placement-based synthesis (PBS) showing improvement in routing

5 Experimental Results

The placement-based resynthesis algorithms have been tried on a number of designs obtained from industry. Both standard cell and gate array designs have been run through with libraries using a 0.8 micron process technology. Table 1 summarizes some of these results. Most of these designs started as Verilog/VHDL descriptions and were synthesized using commercial synthesis tools. They were then placed and routed using commercial tools. Some of these designs had actually gone through multiple iterations of running placement, backannotating the delays and then resynthesizing. The designs range in size from about 580 cells (1269 equivalent gates) to 5000+ cells (25K equivalent gates). We notice that the wire models sometimes tend to overestimate and sometimes underestimate the actual delays and so are not very reliable for the sub-micron devices (- indicates no wire-model was available for the library). We see an improvement in virtually all the examples with the delay reductions ranging from 13 to 22%. Another result we noticed was the improvement in the routability, as illustrated in Fig. 11. The layout on the left is the buffered net highlighted in the original design. The layout on the right is the same buffered net highlighted after it had been resynthesized using the tool. It is evident that the placement configuration of the buffers on the right results in much less congestion and therefore in a more routable design.

Table 1: Placement-based Delay Opt. Results

Circuit	Size		Delay			%
	cells	gates	w. m.	pre-	post-	red
Ckt1	585	1269	23.8	28.9	22.6	22%
Ckt2	4513	9539	81.1	71.2	58.1	18%
Ckt3	5008	25259	23.4	23.4	20.3	13%
Ckt4	2338	9352	-	4.0	3.49	13%

6 Conclusion

In this paper we have proposed a methodology and described algorithms implementing the methodology to resynthesize a design to meet the delay constraints after placement. It starts with the design after initial placement is done and makes incremental modifications to the network, taking placement into account. A final netlist and placement are then generated that meet the delay constraints after placement. We have developed a set of algorithms and implemented them as part of a system for placement-based synthesis. The results have been very encouraging. The tool has been used with a number of pre-optimized, placed designs from industry and has resulted in improvements of post-placement delays ranging from 13 to 22% with improved routability.

References

[1] H. B. Bakoglu, *Circuits, Interconnections, and Packaging for VLSI.* Addison-Wesley, 1990.

[2] P. Abouzeid, K. Sakouti, G. Saucier, and F. Poirot, "Multilevel Synthesis Minimizing the Routing Factor," *Proc. of the 27th Design Automation Conf.*, pp. 365–368, 1990.

[3] M. Pedram and N. Bhat, "Layout Driven Technology Mapping," *Proc. of the 28th Design Automation Conf.*, pp. 99–105, 1991.

[4] M. Pedram and N. Bhat, "Layout Driven Logic Restructuring/Decomposition," *IEEE Intl. Conf. on Computer-Aided Design (ICCAD-91)*, pp. 134–137, 1991.

[5] H. Vaishnav and M. Pedram, "Routability-Driven Fanout Optimization," *Proc. of the 30th Design Automation Conf.*, pp. 230–235, 1993.

[6] S. Lin, M. Marek-Sadowska, and E. S. Kuh, "Delay and Area Optimization in Standard-Cell Design," *Proc. of the 27th Design Automation Conf.*, pp. 349–352, 1990.

[7] K. J. Singh and A. Sangiovanni-Vincentelli, "A Heuristic Algorithm for the Fanout Problem," *Proc. of the 27th Design Automation Conf.*, pp. 357–360, 1990.

[8] S. Lin and M. Marek-Sadowska, "A Fast and Efficient Algorithm for Determining Fanout Trees in Large Networks," *Proc. of the European Conference on Design Automation*, pp. 539–544, 1991.

[9] T. H. Cormen, C. E. Leiserson, and R. R. Rivest, *Introduction to Algorithms.* MIT Press, 1990.