An Integrated Logical and Physical Design Flow for Deep Submicron Circuits

Amir H. Salek, Jinan Lou, and Massoud Pedram
Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089
{amir, jlou, massoud}@zugros.usc.edu

ABSTRACT - This paper presents a set of techniques and a new design flow to be used in the synthesis of high-performance deep-submicron logic circuits. The design flow consists of circuit partitioning into tree-like clusters, floorplanning, global routing, and timing analysis/budgeting steps, followed by simultaneous technology mapping and linear placement of each cluster. The strength of this approach lies in the dynamic programming-based algorithms used in performing simultaneous technology mapping and linear placement of the logic clusters. The two algorithms we propose, one for exact total (gate plus routing) area minimization and the other for total (gate plus routing) delay minimization, generate a set of non-inferior solutions for each cluster enabling designers to perform trade-offs between total-area and total-delay. Experimental results on a large number of MCNC benchmarks prove the effectiveness of the proposed flow.

I. INTRODUCTION

During the process of designing high performance VLSI circuits, designers often find that their implementations do not meet the timing and/or area constraints after layout. This condition is primarily caused by the weak interaction between logic synthesis and physical design tools. Logic synthesis which is capable of significantly altering the circuit timing and area as it proceeds from Boolean network optimization to technology mapping, uses relatively simple models for wires. In contrast, physical design which has accurate wire information from back-end extraction tools, does not have the ability to alter the circuit structure and node functionality and therefore cannot drastically change the timing/area profile of the circuit.

This problem is compounded as feature sizes shrink to a quarter micron and below exposing IC designers to a new era characterized by the following phenomena:

- The dominance of interconnect delay and area due to faster and smaller gates, yet increasing chip dimensions.
- Design iterations due to discrepancies in post and pre synthesis delay calculations.
- Adverse affects of complicated second-order effects such as cross-coupling and ground bounce on chip performance.
- Sharp increases in the number of nets at performance risk.
- Failure of existing delay calculators to guarantee consistent timing across the design stages.
- Sheer complexity of multi-million gate chips which overwhelms the existing design tools.

Design technologies must be improved for design flows, methodologies, tools, and standards to enable the production of chips with a 100 million or more transistors using the same number of designers in the same time it takes today for a 5 million transistor chip. Without these enhancements, semiconductor and electronics industries will die economically as they fall off the productivity curve [Bu97].

There are generally three different approaches to solve this problem: *synthesis centric*, *physical design centric*, and *unification-based*. The logic synthesis centric approach focuses on the innovation of more powerful synthesis techniques. During each step of a logic synthesis flow, local iterations of logical and physical techniques are performed with different coarseness. For example, RT-level floorplanning is used during technology independent logic synthesis to more accurately estimate the global interconnect, whereas gate placement is used during technology mapping. The physical design centric approach focuses on the improvement of physical design techniques. Logic synthesis is performed as post-layout restructuring and re-synthesis techniques to meet various timing/physical constraints. The unification-based approach which is the most powerful approach, attacks the deep submicron problem by considering several design steps simultaneously such that the optimization is performed concurrently for all these steps.

The work presented in this paper on logical-physical co-design belongs to the class of unified techniques. It offers a solution to some of the design difficulties by introducing two new algorithms, SiMPA-E and SiMPA-D, in addition to a novel design flow, FPD-SiMPA, which properly takes advantage of the capabilities of these two algorithms. SiMPA-E (Simultaneous Technology Mapping and Linear Placement Algorithm for Exact Area Minimization) is a polynomial complexity, dynamic programming based algorithm that simultaneously performs technology mapping and linear placement on tree-like circuits. This algorithm minimizes the total (gate plus routing) area by generating and propagating gate area versus cut cost trade-off curves. SiMPA-D (Simultaneous Technology Mapping and Linear Placement Algorithm for Delay Minimization), an algorithm similar to SiMPA-E, can perform optimization with respect to both total-delay and total-area by generating three dimensional gate area, cut cost, and total-delay trade-off curves. Every trade-off curve, in both SiMPA-E and SiMPA-D, contains a set of non-inferior points, each representing a distinct implementation for the input tree-like circuit. SiMPA-E guarantees the exact minimum total-area implementation to be among the non-inferior solutions of its final trade-off curve. Similarly, the best solution found by SiMPA-D with respect to area, is proved to be, at worst, a constant factor away from the optimal solution. This shows that SiMPA-D, which is capable of delay minimization, is an approximation algorithm with respect to total-area, and not simply a heuristic with unbounded error.

FPD-SiMPA (Floorplan Driven Simultaneous Technology Mapping and Linear Placement Algorithm) exploits the capabilities of SiMPA-E and SiMPA-D to perform optimization on general DAG-structured circuits. FPD-SiMPA partitions a given circuit into a set of non-overlapping trees (clusters) and then floorplans the generated clusters using their corresponding estimated area and delay. Subsequently, it performs global routing for the inter-cluster connections and calculates the delays of the global wires. The next step is computing the area and delay budgets and assigning them to the clusters. For each tree, SiMPA-E or SiMPA-D is called to generate and pick the implementation which satisfies the budgets generated in the earlier stage. Finally, FPD-SiMPA refines the floorplan solution to eliminate overlaps and generate the final chip layout. FPD-SiMPA can take either an unmapped or a mapped circuit as its input. In *standard FPD-SiMPA flow,* the input circuit is unmapped and the area/delay of the tree clusters are estimated prior to the floorplanning stage. In *remapping FPD-SiMPA flow,* the input circuit is already mapped and the area/delay values are computed using the mapped solution, and only a selective set of tree clusters will be remapped/re-placed to enhance the performance of the circuit. In this paper due to space limitations only the standard FPD-SiMPA flow is introduced and discussed.

This paper is organized as follows: In section II, background and prior work on placement, synthesis and the methodologies combining these two are introduced. In section III, SiMPA-E, SiMPA-D, and the related techniques used in this methodology are introduced. Section IV describes the proposed design flow (FPD-SiMPA) in which the combined placement and technology mapping method works for general circuits. In section V, experimental results are presented and analyzed. The concluding remarks are stated in section VI.

II. BACKGROUND AND PRIOR WORK

II.1. Technology Mapping

The problem of optimally binding a DAG-form subject graph to an arbitrary library of components is NP-hard [HS96]. In 1987, Keutzer [Ke87] pointed out the similarity between the library binding problem and optimal code generation in a compiler. In his algorithm, the circuit is partitioned into tree sub-graphs and then each sub-graph is mapped using a dynamic programming algorithm that finds the minimum gate area mapping of the tree in polynomial time. This work was later extended by Rudell [Ru89] to minimum delay technology mapping and by Touati et al. [TMBW90] to minimum area mapping under delay constraints. In that approach, a range of required times at each node is computed and divided into equal intervals. Then, during a traversal from primary inputs to primary outputs, the best mapping solution for each of the required times is generated and stored at the node. Finally, the best solution is generated during a traversal from primary outputs to primary inputs. In [CP92], Chaudhary and Pedram presented a dynamic programming algorithm to construct the set of all possible mappings of a tree with different area-delay trade-offs. A general disadvantage of all the above techniques is that they all assume the dominance of area and delay of gates over interconnects which leads to a considerable amount of design errors in the current technologies.

For future reference, the pseudo code of Keutzer's algorithm, which is referred to as *KA* in this paper, is included below. The inputs to this algorithm are a tree network, *N*, and a technology library, *L*.

```
KA(N, L)
INPUT: A tree-network, N, and a technology library, L
OUTPUT: A mapped network
1. Decompose N using a technology decomposition algorithm
2. Perform a reverse depth-first search (DFS) from the primary inputs to the primary outputs
3. Foreach node, n, in the reverse DFS order
4.     Foreach match, $\mu$, of n
5.         gateArea = accumulated sum of gateArea over all the inputs of $\mu$ + gateArea of $\mu$
6.     Store the best gateArea, and its corresponding match and inputs at n
7. Select the best area solution at the primary output, and recursively build the mapping
   solution for the whole circuit
```

Figure 1, by a simple example, demonstrates how KA works on a tree. In this example, a match, μ, is being considered for node *n*. The gate area up to this point is calculated by summing the gate areas of its three inputs ($input_0$, $input_1$, $input_2$) and the gate area of μ itself.

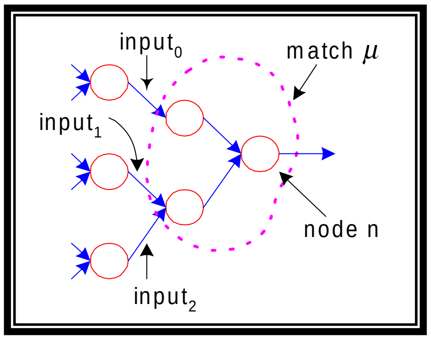


Figure 1 : Illustration of Keutzer's Algorithm

For completeness, the following lemma about KA is presented.

Lemma 1 Keutzer's algorithm gives the minimum gate-area mapping of a tree [Ke87].

II.2. Linear Placement

The linear placement problem of a graph has been studied extensively by many mathematicians. MINSUM is a variation of that problem in which the objective is to find a placement with the minimum total interconnecting wire length. Likewise, the goal of the MINCUT linear placement problem is to find a placement with the smallest maximum cut-width. Both MINSUM and MINCUT problems are NP-complete for the general graphs [GJ79]. For MINSUM, Shiloach [Sh79] introduced an algorithm with $O(n^{2.2})$ complexity which solves that problem for (rooted) trees. The complexity of that technique was later reduced to $O(n^{1.58})$ by Chung [Ch84]. In solving the MINCUT problem, Lengauer [Le82] invented a polynomial time algorithm for (rooted) trees whose worst case cut-width is within a factor of two of the optimal solution. More recently, Yannakakis [Ya85] introduced an $O(n \times logn)$ dynamic programming algorithm which optimally solves the MINCUT problem for trees. This paper focuses on the MINCUT problem and relies on Lengauer's and Yannakakis' algorithms.

Subsequent to introducing the following two definitions, the Lengauer's and Yannakakis' solutions to the MINCUT are briefly introduced and discussed.

Definition 1 For a linearly-placed tree, the *heavy side* is defined as the side (with respect to the placed root) which includes the maximum cut-width, and the other side is called the *light side*.

Definition 2 A linear placement of a tree with equal maximum cut-on each side of the placed root is called a *balanced placement*.

II.2.1. Lengauer's Algorithm

Lengauer's algorithm (*LA*) gives an approximate solution to the MINCUT placement problem of a tree, where at worst, the cut-width of its output placement is within a factor of two away from the minimum cut-width. LA is a dynamic programming based algorithm which solves the problem for sub-trees and then merges these solutions to construct the complete solution. At each step of dynamic programming, LA preserves the already constructed placement solution for each sub-tree, sorts these solutions in descending order with respect to their cut-widths (ties are broken by giving priority to balanced placements), and places them in that order and in an alternating manner around the root node (see Figure 2 for details.) Therefore, the farthest sub-trees (on the right and left) are those with the highest cut-width. Furthermore, LA places the sub-solutions such that their heavy side faces away from the root [Le82]. Figure 2 shows the heavier sides of sub-trees with thicker lines. For one level of dynamic programming, the pseudo-code of LA is presented below. The inputs to the algorithm are the current root, and the list of its immediate sub-trees, which are assumed to have already been placed.

The following statements formally define *Cut-width and Balance-bit Cost Function (CBCF)* which is the cost function used by LA.

Definition 3 For a linear placement, T, $CBCF(T)$ is defined as a two-tuple of $(cutWidth(T), balanceBit(T))$, where $cutWidth(T)$ is the cut-width of T and $balanceBit(T)=1$ when T is a balanced placement (it is 0 otherwise).

Definition 4 $\alpha=CBCF(T)$ is defined to be smaller than $\beta=CBCF(T')$ (denoted as $\alpha<\beta$) if and only if α is lexicographically smaller than β. In other words, $cutWidth(T)<cutWidth(T')$ or when $CBCF(T)=CBCF(T')$ then $balanceBit(T)< balanceBit(T')$.

The following lemma states that Definition 4 imposes an order on CBCF's. Therefore, a set of CBCF's can be sorted and represented on an axis.

Lemma 2 The '<' relation on CBCF's (cf. Definition 4) is a transitive relation.

Proof: Suppose $\alpha<\beta$ and $\beta<\gamma$, where $\gamma=CBCF(T')$. Following Definition 4, it is shown that $cutWidth(T)<cutWidth(T')$ or when $CBCF(T)=CBCF(T')$ then $balanceBit(T)< balanceBit(T')$. Therefore, it is proved that $\alpha<\gamma$. ■

```
LA( r, p_1 , p_1 , … , p_k)
INPUT: A root node, r, and the list of its already placed immediate sub-trees
OUTPUT: The placement of the tree rooted by r
1.  Sort the placed sub-trees in non-increasing order w.r.t. their CBCF's;
    rename them as P_0 , P_1 , P_2 , …, P_k
2.  Generate and return a placement, P = P_0 P_2 … r … P_3 P_1 , such that for
    each P_i, the heavy side is facing away from the root
```

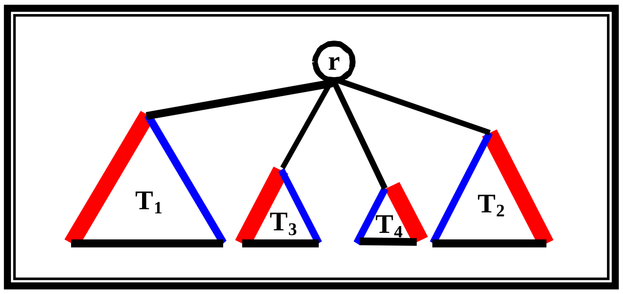


Figure 2 : Illustration of Lengauer's Algorithm

Lemma 3 At worst, the maximum cut-width of a placement generated by LA is twice the cut-width of its corresponding optimal linear placement [Le82].

Lemma 4 $LA(r , p_1 , p_2 , \dots , p_k)$, which is a call to one level dynamic programming of YA, has $O(k\times log(k))$ worst-case runtime complexity, where k is the number of input sub-placements.

Proof: Sorting the input sub-placements with $O(k\times log(k))$ worst-case runtime complexity dominates the other operations. ■

II.2.2. Yannakakis' Algorithm

Yannakakis' algorithm (*YA*) is an exact solution for the tree MINCUT problem. It is a dynamic programming-based, bottom-up algorithm that extends LA to achieve the optimal solution. At each level of dynamic programming, YA does not preserve the already constructed placement solution for the sub-trees; instead it merges them together in a complicated manner in order to achieve the optimal solution.

II.2.2.1. Cut Cost Function

LA uses CBCF of a placement as the cost function during its operation whereas YA uses a generalized form of that cost function, called Cut Cost Function (*CCF*), which is briefly defined and discussed below.

Definition 5 The cut cost function of a linearly placed tree, *P*, is a *k*-tuple of integers that captures the maximum and minimum cut-width locations in the placement.

Suppose that tree T with root r corresponds to the linear placement P. The CCF is calculated as a (finite) sequence $CCF(P)=< g_1 , h_1 , g_2 , h_2 , \ldots >$ as follows:

1. If the maximum cut, g_1, occurs on only one side of P, then **return** $CCF(P)=< g_1 >$
2. Else, let p_1 and p_2 be the two points closest to r on each side where g_1 occurs
3. If p_1 and p_2 are right next to r, then **return** $CCF(P)=< g_1 , g_1 >$
4. Else, let h_1 be the minimum cut between p_1 and p_2
5. If h_1 occurs only on one side in the interval of $[p_1 , p_2]$, then **return** $CCF(P)=< g_1 , h_1 >$
6. Else, let q_1 and q_2 be the two points closest to r in the interval of $[p_1 , p_2]$ where h_1 occurs, and let g_2 be the maximum cut between q_1 and q_2
7. If g_2 occurs only on one side in the interval of $[q_1 , q_2]$, or $g_2 = h_1$, then **return** $CCF(P)=< g_1 , h_1 , g_2 >$
8. Else, let P' be the restriction of P to the interval of $[q_1 , q_2]$ and **return** $CCF(P) = < g_1 , h_1 , CCF(P') >$

The example in Figure 3 shows how a CCF is calculated for a placement, P. In $CCF(P)$, 4 comes first because the maximum cut-width of P is 4. Since max-cut 4 occurs on both sides of P and they are not located adjacent to the root r, we continue to find the minimum cut-width between these two maximum cut-width locations, which is 1. That minimum cut-width again occurs on the both sides of root r. We therefore call CCF recursively to find the CCF of P', where P' is the restriction of P between where the two 1's occurred. Again, we first have to find the maximum cut-width of P', which turns out to be 3. There are 3's on both sides of r which are not right next to it, so we find the minimum cut between them, which is 2. Again, 2 occurs on both sides of r. Between the two 2's, the maximum cut is now the same as the minimum cut 2. We stop here (as in step 6 of CCF calculation method given above), and return the cut cost function CCF as $CCF(T)$=<4,1,3,2,2> .

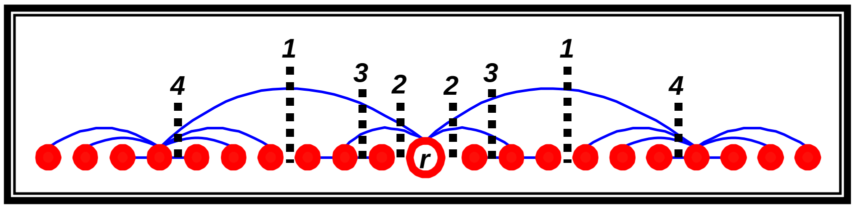


Figure 3 : An Example for Cut Cost Function

Definition 6 $\alpha=CCF(P_1)$ is defined to be smaller than $\beta=CCF(P_2)$ (denoted as $\alpha<\beta$) if and only if α is a prefix of β or α is lexicographically smaller than β.

The following lemma states that Definition 6 imposes an order on CCF's. Therefore, a set of CCF's can be sorted and represented on an axis.

Lemma 5 The '<' relation on CCF's (cf. Definition 6) is a transitive relation.

Proof: It is proved by a simple manipulation following Definition 6. ■

Figure 4 shows three linear placements P_1 , P_2 , and P_3 (roots are shown by larger circles). For these placements, we have: $CCF(P_1)=<2>$, $CCF(P_2)=<2,1>$ and $CCF(P_3)=<2,2>$ and according to the definition $CCF(T_1)<CCF(T_2)<CCF(T_3)$.

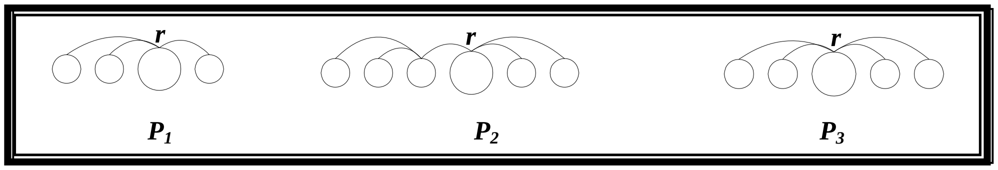


Figure 4 : Ordering on CCF's

Intuitively, CCF's reflect the degree of difficulty involved in performing the MINCUT optimization on a placement. Suppose that the linear placements shown in Figure 4 are to be merged with another placement, say P . Using P_1, the maximum cut of the combined placement remains unchanged if P is placed on the right side of P_1, however, for P_2 regardless of the relative position of P, the maximum cut-width of the combined placement is increased by one. Both P_2 and P_3 have the maximum cut-width of two on both sides, however, P_2 has a minimum cut of one on its left side which can be used for performing further mincut optimizations by breaking P_2 at that location and inserting P between the two parts. YA actually relies on the optimization techniques similar to one mentioned for P_2 in order to construct the optimal solution. More details will be given in section II.2.2.2.

Lemma 6 Assume *CCF(P)* is the cut cost function of a cut-width optimal linear placement *P*. The following two formulas give bounds on the cardinality of *CCF(P)*:

$$|CCF(P)| \leq num(P) + 1$$

$$|CCF(P)| \leq cutWidth(P) + 1$$

where *num(P)* and *cutWidth(P)* denote the number of nodes and the maximum cut-width in the placement, *P*, respectively [Ya85].

Lemma 7 Assume *P* is a cut-width optimal linear placement of a fixed degree tree *T*. Then, $cutWidth(P) \leq log(|P|)$, where $|P|$ denotes the number of nodes in *P* [Ya85].

II.2.2.2. Exact MINCUT Linear Placement Algorithm

Yannakakis' algorithm is a relatively complicated algorithm and in this section only the skeleton of the algorithm will be presented. Interested readers are referred to [Ya85] for more details. The following pseudo-code briefly describes the main steps of YA in one level of its dynamic programming.

```
YA( r , $p_1$ , $p_2$ , … , $p_k$ )
INPUT: A root node, r, and the list of its already placed immediate sub-trees
OUTPUT: The placement of the tree rooted by r
  1. Sort the placed sub-trees in non-increasing order w.r.t. their CCF's; rename
     them to $P_0$ , $P_1$ , $P_2$ , …, $P_k$ ; form a placement, P, similar to Lengauer's
     Algorithm
  2. Return P unless one of the following cases occurs:
     Case A: The maximum cut-width occurs only once in P (on the left side of r)
             inside the placement $P_{2t}$ . Return the refined placement after
             applying function OP1 as shown in Figure 5
     Case B: The maximum cut-width occurs on both sides of r . Among the
             placements on the right side of r in which the maximum cut-width
             occurs, choose the one with the largest index, say $T_{2t+1}$ . Return the
             refined placement after applying function AN1 as shown in Figure 5
```

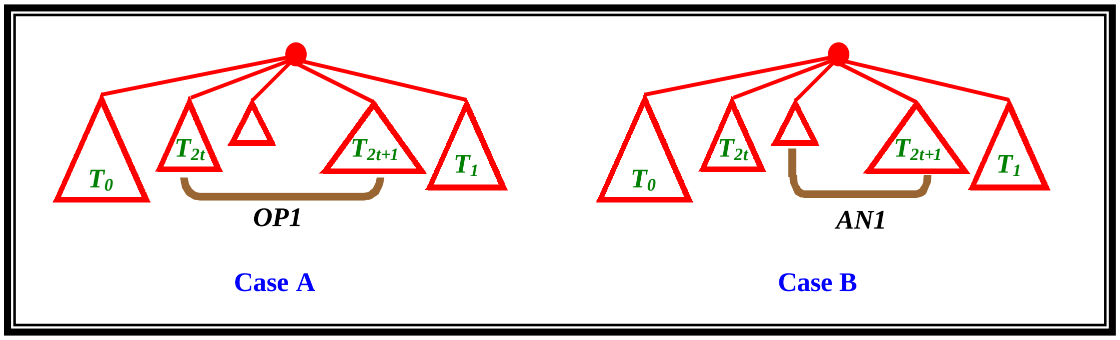


Figure 5 : Illustration of Yannakakis' Algorithm

Initially, YA places the sub-trees in a manner similar to LA except that it sorts them with respect to their CCF's instead of their cut-width and balance bits (CBCF's). Then, it performs some refinement operations on the resulting solution so that the optimal placement is achieved. During a refinement operation (such as *OP1* or *AN1*), YA may break a placement along one or more of its mincut solutions (to prevent avoidable increases in the cut-width), and inserts the rest of the placement in the generated gap. Figure 6 demonstrates two of these break up schemes. In that figure, the drawing on the left shows that a placement is broken into two parts and subsequently the rest of the placement, *P′*, is inserted in between them to form the final placement. The drawing on the right demonstrates a case where *P* is broken into three parts (left, middle, and right) and *P′* is divided into two parts (right and left). The left section of *P′* is inserted between the left and middle parts of *P*, and similarly the right part of *P′* is inserted between the middle and right parts of *P* . As mentioned earlier, the placements are ordered based on their CCF's because the mincut information reflected in the CCF's is essential to the aforementioned refinement operations. Note that there are many other breaking and inserting schemes in the YA which can be found in [Ya85].

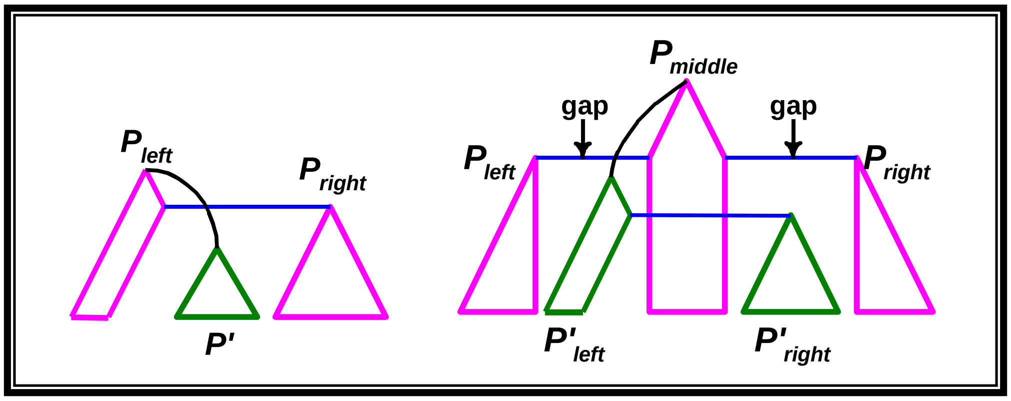


Figure 6 : Refinement Operations in Yannakakis' Algorithm

Lemma 8 YA finds the minimum cut-width placement of a tree [Ya85].

Lemma 9 YA is monotone with respect to CCF's, that is, while combining the input placements at one level of dynamic programming, the CCF of the generated placement becomes larger if other placements with larger CCF's replace one or a subset of the original input placements. [Ya85]

Lemma 10 $YA(r , p_1 , p_2 , \dots , p_k)$, which is a call to one level dynamic programming of YA, has $O(k \times l)$ worst-case runtime complexity, where k is the number of input sub-placements and l is the summation of $|CCF(p_i)|$ for $1 \leq i \leq k$. [Ya85]

II.2.3. An Example of Linear Placement

Figure 7 shows a 15-node tree that has been linearly placed using three different algorithms: linear projection algorithm, LA, and YA. The linear projection method preserves the order on the projection of nodes of the input tree to the x-axis and performs no optimization. In this example, that approach has resulted in a placement with the maximum cut-width of 4 whereas Lengauer's algorithm via some optimization techniques, has been able to reduce it to 3. For instance, LA has flipped over the sub-tree rooted by node 4 such that its heavy side is facing away from the root. Yannakakis' algorithm further reduces the maximum cut-width to 2 by splitting sub-tree placements. For instance, the sub-tree rooted by node 2 is broken into two parts, and the rest of the placements (node 1 and 3) are inserted in the generated gap.

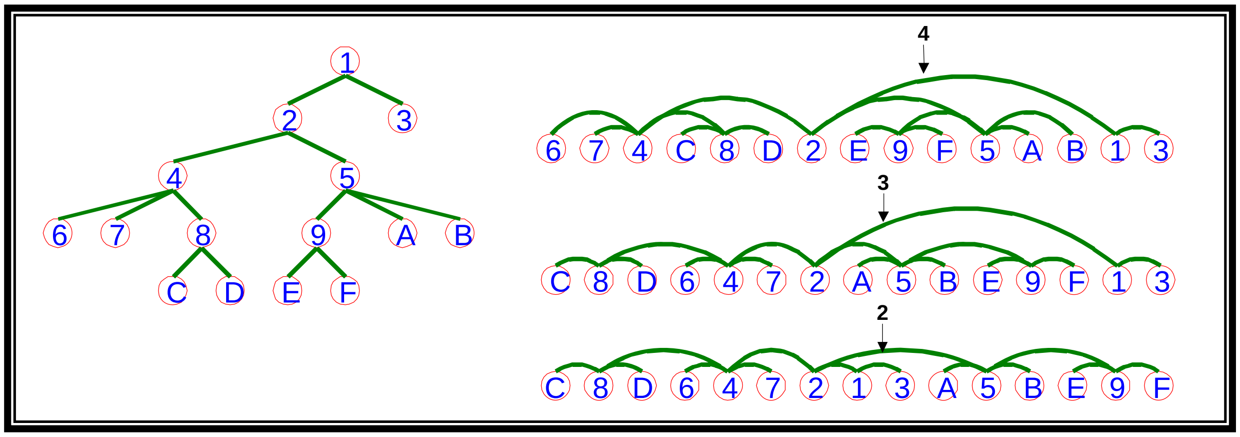

1 2 3 4 5 6 7 8 9 A B C D E F
4
6 7 4 C 8 D 2 E 9 F 5 A B 1 3
3
C 8 D 6 4 7 2 A 5 B E 9 F 1 3
2
C 8 D 6 4 7 2 1 3 A 5 B E 9 F

Figure 7 : An Example of Linear Placement

II.3. Combining Synthesis and Physical Design

Bridging the gap between the synthesis and physical design phases has been accomplished by three approaches. The first approach starts with synthesis techniques and extends downward to the world of physical design. By contrast, the second approach begins with physical design techniques and extends upward to the synthesis domain. The last approach is based on a hybrid strategy and eliminates the gap by unifying the design steps.

II.3.1. From Synthesis to Physical Design

Pedram and Bhat's work in [PB91a] was the first attempt in combining physical design with logic synthesis. They introduced the notion of coupled mapping and placement in order to consider the effect of wires during mapping. The key idea was to generate a "companion" placement during the mapping phase. The placement information is used to evaluate the cost of a gate match during the mapping process. The placement is dynamically updated in order to maintain the correspondence between the logic and layout representations. At the conclusion, a mapped network along with a placement solution is generated. The placement solution is then globally relaxed in order to produce a feasible placement according to the target layout style. This algorithm assumes that during the bottom-up process of concurrent mapping and placement, the dynamic programming principle holds, which is only an approximation. The same authors extended this idea to logic restructuring and technology decomposition in [PB91b].

II.3.2. From Physical Design to Synthesis

Physical design consists of a myriad of techniques designed specifically for the variety of problems which arise in this field, ranging from physical partitioning and floorplanning to placement and routing. Most of the connections already created between this field and synthesis are established through placement. Placement algorithms that apply local netlist transformations, after an initial placement, have the advantage of using accurate information about delay and area for performing a good synthesis. The algorithm proposed in [KSF94] starts from an initial placement followed by timing optimization using fanout buffering and gate resizing transformations. Estimations of the net delays based on the initial placement are used for selecting the most useful transformations. In Chuang et al. they applied a linear programming approach for resizing and relocating of critical gates [CH94]. In [LPPD93], the authors proposed re-synthesizing the logic in the most congested regions of the chip so as to reduce the routing area. Stenz et al. in [SRRJ97] proposed a technique that performs iterative timing driven netlist transformations on a companion placement. All of these techniques perform re-synthesis on an already mapped and placed circuit while using different types of re-synthesis techniques. None of these techniques integrate placement and synthesis.

II.3.3. Unified Approach

Simultaneously performing logic synthesis and physical design eliminates the problem of weak interaction between the two steps. Technology mapping is the last transformation during logic synthesis in which a considerable amount of freedom in changing the circuit netlist exists. In addition, placement is among the operations performed very early during physical design that can largely alter the physical implementation of the circuit. The algorithms provided in [LSP97] and [LSP98] (which are the earlier presentations of the algorithms proposed in this paper) for the first time merge the aforementioned design steps and provide a unified design step. The details of this approach and the proposed algorithms are given throughout this paper.

III. SIMULTANEOUS TECHNOLOGY MAPPING AND LINEAR PLACEMENT

During the process of VLSI circuit design, the size of the solution space becomes so large that in many cases achieving the optimum implementation is impractical, or nearly impossible. In certain situations however, there are special algorithms capable of handling large solution spaces by locating the optimum solution with relatively little effort. Otherwise, designers have to employ one or a combination of the following two strategies to solve their design problems: adopting heuristic algorithms, or reducing the complexity by dividing design tasks into several sub-tasks. The first strategy sacrifices quality in order to improve the algorithm efficiency. In the latter case, even if the optimal solution for each sub-task is found, overall optimality is not guaranteed. Thus, it sacrifices optimality for reduced complexity as well.

The general forms of technology mapping and linear placement problems are NP-hard. As described earlier, YA and KA optimally solve these problems for tree-structured circuits. The sequential application of YA and KA does not however produce a global optimal mapping and placement solution even for this restricted class of circuit structures. This paper addresses this shortcoming by proposing SiMPA (Simultaneous Technology Mapping and Linear Placement) which constructs a concurrent mapping and placement solution by searching in the combined solution spaces of these design tasks. In particular, two major variations of SiMPA are proposed here: SiMPA-E and SiMPA-D. SiMPA-E (E stands for Exact Minimum Total-area) optimally finds the implementation with the least total (gate+wiring) area. SiMPA-D (D stands for Total-delay Minimization) is an approximation algorithm which finds a set of non-inferior solutions with respect to both area and delay. These algorithms are introduced and discussed in detail in this section. Note that throughout this paper, the input library of gates is assumed to be finite.

III.1. SiMPA-E

***Problem formulation*:** *Given a library, L, and a decomposed tree-structure circuit, N, develop an algorithm for performing technology mapping and linear placement for N which results in minimum total (gate+routing) area.*

Before describing the details of our proposed algorithm, the model used for calculating the total-area as well as other issues fundamental to SiMPA-E, are presented and discussed below.

III.1.1. Total-area Calculation

SiMPA targets the standard-cell design style and consequently the area model given here calculates the total-area of such designs. This model captures the two main geometric characteristics of the standard-cell design: uniform gate height and proportionality of the routing area with the maximum cut-width of the linear placement. Figure 8 illustrates how these characteristics are used in total-area modeling and calculation.

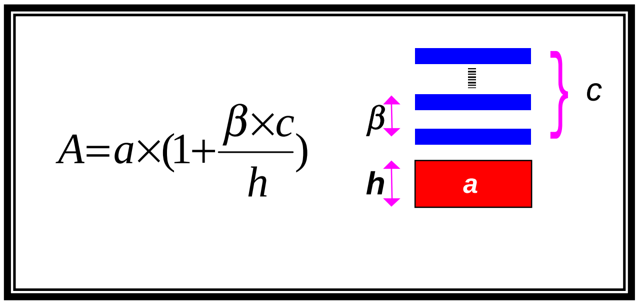


Figure 8 : Total-area Calculation

In Figure 8, the lower box shows the space taken by gates (denoted by *a*), all of which have the same height, *h*. The lines above it demonstrate the interconnections. Also, the space between every two neighboring wires is a constant β, and the maximum cut-width of placement is denoted by *c*. This equation is exact for two-layer routing technologies. For α-layer routing, with α' horizontal routing layers, one must simply use $\lceil c/\alpha' \rceil$ instead of *c*.

The above equation used as the cost function of SiMPA-E, shows that the total-area of a row in a standard-cell layout is proportional to the product of the gate-area (*a*) and the cut-width (*c*). In this algorithm, gate area is optimized by KA and cut-width is optimized by YA.

Observation 1 The gate area of any linear placement for a mapped circuit is the summation of the areas of its gates.

III.1.2. Gate Area versus CCF Curves

At every stage of a bottom-up decision making scheme, solutions are built on top of the existing sub-solutions made available by previous calls to the same procedure. However, if the cost function is uni-variable at each level, all the generated solutions except the one minimizing the cost function are discarded. For the case of KA, the cost function is the total gate area (*a*) which is a uni-variable function. Similarly, YA employs a uni-variable cost function called CCF and always carries the solution with the lowest CCF (cf. Definition 6). This pruning process does not have an adverse effect on the optimality of these algorithms, but results in their polynomial complexity.

SiMPA-E's objective is to minimize the total-area of the final implementation. However, total-area cannot be used as the cost function during SiMPA-E's dynamic programming approach. The counter example given in Figure 9 demonstrates a case in which the optimality of the algorithm is compromised if total-area is used as the cost function.

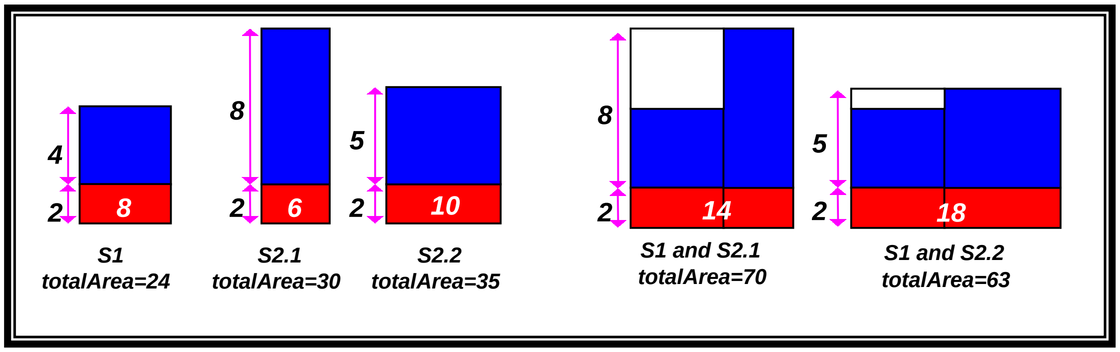


Figure 9 : A Counter-Example

Figure 9 demonstrates a dynamic programming step in which the solutions of two sub-problems are to be combined. Let's assume that for the first sub-problem one solution, S_1, is available and for the other, two solutions, $S_{2,1}$ and $S_{2,2}$, have already been generated. As shown above, the total-area of S_1 , $S_{2,1}$, and $S_{2,1}$ are 24, 30 and 35 units, respectively. If total-area was to be used as the cost function, $S_{2,2}$ would have been deemed inferior with respect to $S_{2,1}$ and thus dropped. However, the above figure shows that the combination of S_1 and $S_{2,2}$ yields an implementation with lower total-area (63 units) compared to the combination of S_1 and $S_{2,1}$ (70 units). Therefore, total-area, if used as the cost function of a bottom-up algorithm, may eliminate solutions which can potentially lead to better solutions in subsequent steps. This discussion shows that total-area is not a proper cost function and may result in the loss of optimality. However, by explicitly storing the two variables of which total-area is a function (i.e. CCF and gate-area), it becomes possible to guarantee optimality.

Fortunately, the dynamic programming nature of KA and YA facilitates their coupling in SiMPA-E. Through the bottom-up solution generation, KA and YA explicitly require the gate-area and CCF information associated with each solution in order to generate the optimal solutions. Therefore in SiMPA-E two-dimensional solution curves (CCF versus gate-area) are generated and stored at each step of dynamic programming. Later in this section, it is proved that this scheme optimally solves the problem of minimum total-area simultaneous technology mapping and linear placement. Figure 10 illustrates a simple two-dimensional CCF versus gate-area solution curve. Note that although CCF is a k-tuple, the ordering relation defined in Definition 6 enables us to one-dimensionally sort CCF's and have curves like the one below with one axis for CCF.

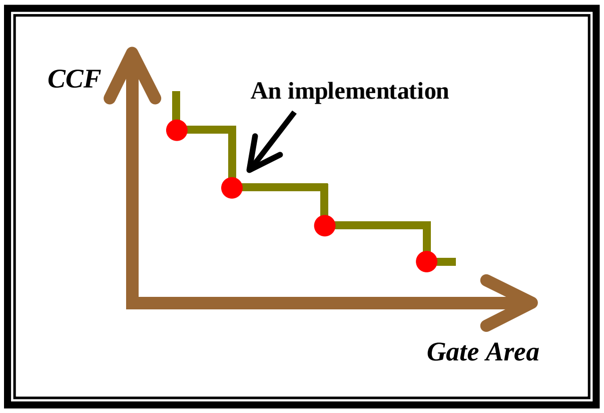


Figure 10 : A CCF versus Gate Area Curve

Definition 7 Assume Ω is a CCF versus gate-area solution curve. Solution $\sigma \in \Omega$ is defined to be *inferior (dominated)*, if there exists a $\gamma \in \Omega$ where $CCF(\gamma) \leq CCF(\sigma)$ and $gateArea(\gamma) \leq gateArea(\sigma)$.

Back to the example of Figure 9, according to Definition 7, $S_{2,2}$ is not dominated by $S_{2,1}$. Therefore, both solutions are preserved for the next level of dynamic programming. This resolves the problem described previously and enables SiMPA-E to find the implementation with the minimum total-area.

III.1.3. SiMPA-E in Detail

SiMPA-E, using a bottom-up strategy, generates and propagates CCF versus gate-area solution curves in order to find the minimum total-area implementation of a circuit. The algorithm consists of two major steps, the *merge* and *prune* operations, which are introduced below while describing SiMPA-E's pseudo-code.

```
SiMPA-E( N , L )
INPUT: A tree network N and a library L
OUTPUT: A mapped and linearly placed network
1.  Decompose N
2.  Foreach node, n, in reverse depth-first order from the primary inputs to the primary
    output of N
3.      Foreach match $\mu$ of node n
4.          Let $IN_1 , \dots , IN_{\alpha(\mu)}$ be pointing to the roots of the sub-trees connected to the
            to inputs of $\mu$
5.          Until all combinations of $(s_1 , \dots , s_{\alpha(\mu)})$ where $s_i \in IN_i$ are enumerated do
6.              $p = YA( \mu , s_1 , \dots , s_{\alpha(\mu)})$
7.              $g = \sum_{\forall i} gateArea\,(s_i)$
8.              Add <g,CCF(p)> along with $( \mu , s_1 , \dots , s_{\alpha(\mu)})$ to the solution curve of n
9.      Prune the inferior points in the 2-D solution curve of n
10. Find the best solution in the primary output's 2-D curve and recursively trace back its
    constituent sub-solutions
```

SiMPA-E takes a tree-like circuit and a library of gates as its inputs. In line 1 of the above pseudo-code, SiMPA-E decomposes the input circuit, as a required transformation prior to performing technology mapping. Then it starts visiting and processing all the nodes of the decomposed tree network in reversed depth-first order (line 2) and executes the following operations on each node. For each visited node, SiMPA-E tries all the possible matches in the library and identifies the sub-trees connected to the inputs of each match (line 3 and 4). Note that the bottom-up dynamic of SiMPA-E guarantees the correct processing order for the sub-trees by SiMPA-E and the availability of their corresponding two-dimensional curves.

The solution curves of the sub-trees are combined by the *merge* operation (lines 5 through 8). In line 5, all combinations of non-dominated solutions (implementations) for each input sub-tree are enumerated. Next, for every combination along with the current match YA is called to generate the optimum placement (line 6.) The generated implementation corresponding to the current combination has a gate-area that is simply the summation of the gate-areas of the sub-solutions (line 7.) In line 8, the two cost function values for the newly generated solution (gate-area and CCF), the pointers to the corresponding sub-solutions, and the match are inserted into the solution curve of the current node.

Having called the merge operation for all of the input sub-solution combinations, SiMPA-E investigates the generated solution set (curve) and deletes the dominated ones (line 9.) This is referred to as the *prune* operation whose execution is defined by Definition 7. The prune operation eliminates the inferior solutions since, according to the following theorems, they cannot be used in any optimal solution. Therefore, this operation, which maintains the polynomial complexity of SiMPA-E, is safe as far as the optimality of SiMPA-E is concerned.

In line 10, SiMPA-E has visited all the circuit nodes from the primary inputs through the primary output. The solution set associated with the primary output, *final curve*, contains all the non-inferior implementations of the circuit (including the minimum total-area implementation) and the designer has the option to choose the one with desired aspect ratio. Once a solution in the final curve is picked, the pointers to its constituent sub-solutions (cf. line 8) are traced back and the details of the implementation are retrieved.

Generating, propagating, and maintaining solution curves are the main tasks performed by SiMPA-E. Before stating the properties of SiMPA-E, the following statements describe the characteristics of those operations and their related issues.

Lemma 11 The number of non-inferior points on any gate-area versus CCF curve is $O(n)$, where n is the number of nodes in the technology decomposed tree circuit.

Proof: Suppose A is the largest area of a gate in the input library. A (loose) upper bound on the maximum gate-area of sub-solutions is nA. By normalizing that value with respect to the smallest difference between two gate areas δ, an upper bound on the number of possible distinct gate-area values, nA/δ, is obtained. The prune operation keeps at most one solution per each distinct gate-area value in a curve, therefore the maximum number of points in a gate-area versus CCF curve is $O(n)$. ■

Theorem 1 The worst-case runtime of the merge operation in SiMPA-E, excluding the time for linear placement generation, is $O(n^k)$, where k is the maximum number of inputs to any gate in the library.

Proof: The merge operation combines at most k curves at a time. Lemma 11 states that the number of solutions in each curve is bounded by $O(n)$. Also, the merge operation examines all combinations of the input sub-solutions in line 5 of the pseudo-code. Therefore, the worst-case runtime of the merge operation is $O(n^k)$. ■

The following theorem justifies the claim that the prune operation is safe with respect to the overall optimality of SiMPA-E.

Theorem 2 At each step of the dynamic programming where SiMPA-E is generating the solution curve of node α, adding a point S_2 (which is assumed to be inferior to S_1) to an input curve cannot add a non-inferior solution to the curve of node α.

Proof: This is a proof by contradiction. Assume S_2 along with an arbitrary combination of sub-solutions for the other inputs of α generates a non-inferior solution S'_2, i.e. S'_2 is made of $\{\sigma_1,\ldots,\sigma_{b-1},S_2,\sigma_{b+1},\ldots,\sigma_k\}$. Also, consider another possible implementation S'_1 which is made of $\{\sigma_1,\ldots,\sigma_{b-1},S_1,\sigma_{b+1},\ldots,\sigma_k\}$.

1. The only different constituent sub-solutions of S'_2 and S'_1 are S_2 and S_1; Furthermore, it is assumed that $gateArea(S_2)\geq gateArea(S_1)$. Since the gate-area of a solution is the summation of the gate-areas of its constituent sub-solutions (cf. Observation 1), $gateArea(S'_2)\geq gateArea(S'_1)$.

2. The only differing constituent sub-solutions of S'_2 and S'_1 are S_2 and S_1; Furthermore, it is assumed that $CCF(S_2)\geq CCF(S_1)$. The CCF-monotone property of YA (cf. Lemma 9) implies that $CCF(S'_2)\geq CCF(S'_1)$.

The above inequalities prove that S'_2 must be inferior with respect to S'_1, which is a contradiction. ■

In the rest of this section, the term '*solution space*' refers to a set consisting of all mappings of the decomposed circuit, each optimally placed with respect to the total-area.

Theorem 3 The final solution curve, generated by SiMPA-E at the root of tree T, includes all the non-inferior gate-area versus CCF solutions for the simultaneous technology mapping and linear placement problem of T.

Proof: Without the use of the prune operation, SiMPA-E would visit all the points of the solution space including every non-inferior solution. That is because, at each level, SiMPA-E exhaustively examines all the combinations of input sub-solutions and calls YA to generate the minimum total-area implementation for each. Note that for a mapped circuit, YA generates the minimum total-area placement since h, β, and W (cf. Observation 1) are constants. Despite the use of the prune operation, SiMPA-E is still guaranteed to visit all the non-inferior solutions, since according to Theorem 2 the dropped sub-solutions can never be used in any non-inferior implementation. ■

The following two corollaries are useful for post-layout logic re-synthesis purposes. Using the gate-area versus CCF final curve, SiMPA-E is capable of performing constrained re-synthesis, i.e. it guarantees to find the best solution to meet a user-specified constraint.

Corollary 1 Given an upper bound on the number of tracks (channel height), SiMPA-E finds the minimum gate-area (total cell width) implementation satisfying this constraint.

Corollary 2 Given an upper bound on the gate-area (total cell width), SiMPA-E finds the minimum cut-width (channel height) implementation satisfying this constraint.

Lemma 12 For a decomposed circuit as input, suppose that M_{KA} is the mapping generated by KA. The gate-area of the left-most point in the final curve generated by SiMPA-E is equal to the gate-area of M_{KA}.

Proof: The optimality of KA with respect to gate-area guarantees that the gate-area of the M_{KA} is the lowest among all the mappings. Therefore, the solution with that gate-area and the lowest CCF, S, is a non-inferior solution. Theorem 3 proved that the final curve includes all the non-inferior solutions including S, and since there is no solution with a lower gate-area, S always stands on the left-most point of the final curve.

Lemma 13 Assume that C_{YA} is the minimum cut-width of the implementations generated by calling YA on all possible mappings of the input circuit. In the final curve, the cut-width of the right-most point is exactly C_{YA}.

Proof: The proof is similar to the one given for Lemma 12.

Theorem 4 The final curve includes the minimum total-area implementation of the input circuit.

Proof: From Theorem 3, the final curve generated by SiMPA-E includes all the non-inferior solutions. So, it is sufficient to prove that the minimum total-area is a non-inferior solution. Call the minimum total-area implementation of the input tree-like circuit S_{min} . This is a proof by contradiction, so assume that S_{min} is an inferior solution. Therefore, there must be a solution, say S_d, with $gateArea(S_{min}) \geq gateArea(S_d)$ and $CCF(S_{min}) \geq CCF(S_d)$. By referring to the equality defining total-area, it is clear that the above inequalities result in $totalArea(S_{min}) \geq totalArea(S_d)$ which contradicts the initial assumption about S_{min}. ■

Theorem 5 The memory usage of SiMPA-E is $O(n^2)$.

Proof: According to Lemma 11, the number of points in each solution curve is $O(n)$. There are at most n solution curves, one for each node in the decomposed circuit. Therefore, the memory requirement is bounded by $O(n^2)$. ■

Lemma 14 The linear placement step in SiMPA-E (cf. Line 6 in the pseudo-code) has $O(log(n))$ worst-case runtime complexity where n is the total number of nodes in the decomposed circuits.

Proof: A call to YA has $O(k \times l)$ worst-case runtime complexity (cf. Lemma 10). In SiMPA-E, k (the number of fanins) is bounded in the input library. Also, $l = \Sigma\ |CCF(s_i)| \leq \Sigma\ log(|s_i|)$, where $|s_i|$ denotes the number of nodes in s_i (cf. Lemma 7), and Σ in this proof means the summation of the argument over $1 \leq i \leq k$. Using algebraic manipulations,

$$O(k\times l)= O(\Sigma \log(|s_i|))=O(\log((\Sigma|s_i|/k)^k))=O(\log((n/k)^k))=O(k\times \log(n)-k\times \log(k))=O(\log(n)) \quad ■$$

Theorem 6 The worst-case runtime complexity of SiMPA-E is $O(n^{k+1}\times \log(n))$, where k is the maximum number of inputs to a gate in the library.

Proof: In SiMPA-E, YA has $O(\log(n))$ worst-case runtime complexity for each call to its dynamic programming based procedure (cf. Lemma 14). During the inner loop of SiMPA-E which is repeated n^k times (cf. Theorem 1) YA is called each time. Also, that loop is repeated for every node in the decomposed circuit and therefore the worst-case runtime complexity of SiMPA-E is $O(n^{k+1}\times \log(n))$. The prune operation is basically a sorting operation, so its worst-case runtime is $O(n\times \log(n))$. It therefore has no effect on the overall worst-case runtime complexity. ■

III.2. SiMPA-D

As discussed earlier, SiMPA-E optimally generates the implementation with the minimum total-area. Although in practice SiMPA-E achieves a moderate post-layout delay improvement, it does not perform explicit delay optimization. The employment of YA causes the invalidation of the already calculated delay values, mainly because it breaks the linear placement of the sub-solutions prior to combining them. By using an approximate linear placement algorithm (LA), SiMPA-D is able to explicitly perform area-delay trade-off in three dimensional solution curves by maintaining and employing the available delay information. SiMPA-D can work with many simple and/or detailed delay models; sub-section III.2.1 introduces a suitable delay model which has been used in this work. Following that, sub-section III.2.2 gives a detail discussion of SiMPA-D.

III.2.1. Total-delay Calculation

The arrival time of a signal at the output of a gate, $OutTime(G)$, is calculated based on the following equations:

$$OutTime(G)=MAX_i\{OutTime(G_i)+WireDelay(G_i, Pin_i)+GateDelay(i, load)\}$$

where:

$InTime(Pin_i)$ is the signal arrival time at the ith input pin of G.

$WireDelay(G_i, Pin_i)$ is the delay through the wire connecting the output of ith fanin gate to Pin_i of G.

$GateDelay(i, load)$ is the delay through G from Pin_i to the output of G.

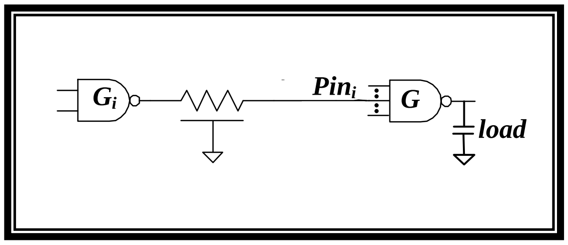


Figure 11 : Delay Calculation

There are many different models for gate delay calculation. In SIS [SSLM92], both rise and fall gate delays are calculated using this equation:

$$GateDelay(i,load)=K_{1,i}+K_{2,i}\times load$$

where $K_{1,i}$ and $K_{2,i}$ are the intrinsic gate delay and the load dependent fanout delay, respectively. This delay equation, which uses a linear model, is pin-dependent and assumes step input signals. This simple delay equation however turns out to be inaccurate, especially in deep sub-micron process technologies.

In this work, a four-parameter delay model which captures the effect of the input slew-rate will be used:

$$GateDelay(i,load)=K_{1,i} + K_{2,i}\times load + TransitionTime_i\times (K_{3,i}+K_{4,i}\times load)$$

where $K_{1,i}$ to $K_{4,i}$ are constants and $TransitionTime_i$ denotes the transition time of the signal at input pin i. A similar equation is used for calculating the transition times for the gate outputs (obviously with a different set of coefficients). The constants of this pin-dependent delay model are obtained by curve fitting for each library cell using the results of extensive circuit-level HSpice simulations.

For wire delay calculations, the Elmore delay model [El48] is used as shown in the following figure:

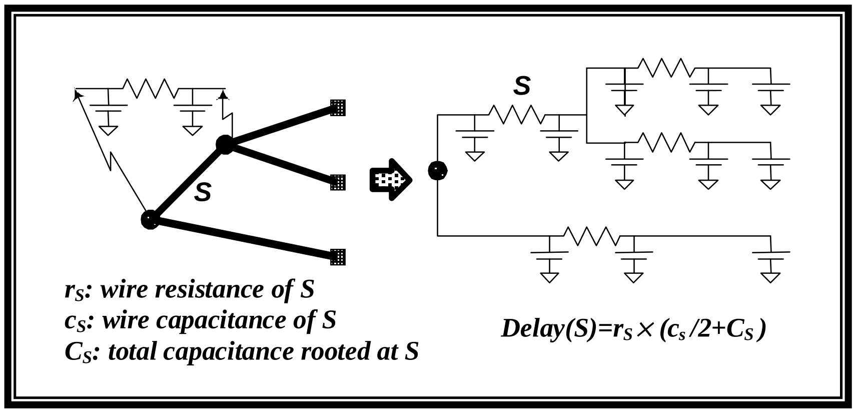

S
S
r_S: wire resistance of S
c_S: wire capacitance of S
C_S: total capacitance rooted at S
$Delay(S)=r_S\times(c_s/2+C_S)$

Figure 12 : Elmore Delay Model

In SiMPA-D, both physical and logical information are available, therefore the total (gate+wire) delay is accurately calculated. The example in Figure 13 demonstrates a gate *g* with three inputs where the sub-trees connected to the inputs of *g* are already mapped and placed. Consequently, the signal arrival time at the output of *g* can accurately be calculated. Note that the accuracy is somewhat reduced due to the so-called unknown load problem. This problem refers to the uncertainty about the mapping and routing in front of the output of *g*.

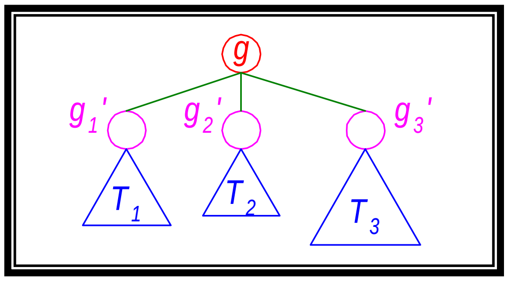


Figure 13 : Total-delay Calculation

In SiMPA-D, the unknown load is assumed to be a constant value equal to the input capacitance of a minimum-size two-input NAND gate plus a statistically estimated wire load (cf. [VP95].) In subsequent steps when the mapping and placement of the fanout become available, the original estimated delay is corrected according to the accurate information about the load. This technique prevents the propagation and accumulation of the error encountered by assuming a constant load ahead.

III.2.2. SiMPA-D in Detail

SiMPA-D targets the problem of simultaneous technology mapping and linear placement for total area-delay optimization. It generates and propagates three-dimensional (cut-width and gate-area versus total-delay) solution curves throughout its bottom-up algorithm and allows the designer to choose the implementation which satisfies the timing and area constraints. The dynamic of this algorithm is similar to that of SiMPA-E, with some important differences as described below.

Figure 14 demonstrates a simple example where YA breaks the placement of a sub-solution T into two parts, T_{left} and T_{right}, and inserts P' in the gap. The length (load) of the wire connecting the two broken parts, *wire*$_1$, is significantly altered by this operation invalidating the prior delay calculations for *T*. In contrast, LA preserves the placement of the sub-solutions and consequently their calculated delays while combining them. This opportunity allows SiMPA-D to use a three-parameter cost function (cut-width, gate-area, and total-delay) and perform optimization with respect to both geometry and timing at the same time. Note that in the cost function and the solution curves, CCF is replaced with cut-width because LA uses cut-width and a balance-bit (CBCF) as its cost function.

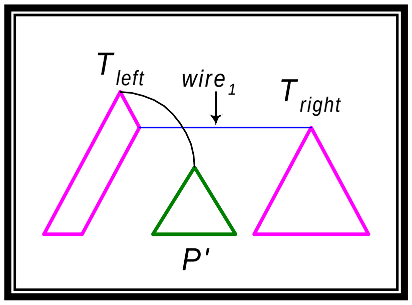


Figure 14 : Violation of the Dynamic Programming Principle

At every step of dynamic programming, three-dimensional solution curves are generated (merge operation) and the inferior points are dropped out (prune operation) in order to maintain the polynomial complexity of the algorithm.

Definition 8 Assume Ω is a cut-width and gate-area versus total-delay solution curve. Solution $\sigma \in \Omega$ is defined to be *inferior* (dominated), exactly when there exists a $\gamma \in \Omega$ and the following conditions hold: $CBCF(\gamma) \leq CBCF(\sigma)$, $gateArea(\gamma) \leq gateArea(\sigma)$, and $totalDelay(\gamma) \leq totalDelay(\sigma)$.

As previously stated, LA produces a placement solution whose cut-width can be at most 2X away from the optimum value. Therefore, SiMPA-D, unlike SiMPA-E, does not guarantee finding the minimum total-area implementation. Inherently, SiMPA-D is an approximation algorithm with respect to total-area with 2X upper bound. However in practice, that upper bound is barely hit and the increase is usually within the range of only a few tracks.

The pseudo-code for SiMPA-D is identical to the one given for SiMPA-E except that in line 6 of the pseudo-code (cf. III.1.3), SiMPA-D calls LA instead of YA.

The following statements describe the characteristics of SiMPA-D.

Lemma 15 The number of non-inferior points on any three-dimensional gate-area, cut-width, versus total-delay curve, generated by SiMPA-D, is bounded by $O(n \times log(n))$, where *n* is the number of nodes in the technology decomposed form of the tree circuit.

Proof: The proof of Lemma 11 shows that the maximum number of distinct gate-area values is bounded by nA/δ. Lemma 7 states that the cut-width of a cut-width optimal linear placement is bounded by $log(n)$. The cut-width of any placement generated by LA is at worst 2X larger than the optimal value. Therefore, in SiMPA-D the cut-width of the sub-solutions is bounded by $2log(n)$. Subsequently, there are $2log(n) \times nA/\delta$ distinct values for (gate-area, cut-width) tuples. In a solution curve, for every possible combination of gate-area and cut-width values the implementation with the least total-delay is stored and therefore the maximum number of non-inferior solutions in such curves is bounded by $O(n \times log(n))$. ■

Theorem 7 The worst-case runtime of the merge operation in SiMPA-D is $O(n^k \times log^k(n))$, where *k* is the maximum number of inputs to any gate in the library.

Proof: The merge operation combines at most *k* curves at a time. It calls LA for all the combination of the input solutions and calculates the gate-area of the resulting placement. Considering that the maximum number of fanins is bounded by a constant determined by the input library, a call to LA has

$O(1)$ worst-case runtime complexity (cf. Lemma 4). Lemma 15 states that each curve contains at most $O(n\times log(n))$ solutions and therefore the worst-case runtime of the merge operation is $O(n^k\times log^k(n))$. ■

LA is monotonic with respect to the cut-width as stated by the following statements.

Lemma 16 Suppose that the root of tree, T, has a set of a immediate sub-placements, $T=\{s_1, s_2, \ldots, s_a\}$. T' is similar to T except in one sub-placement, say s_α, which is replaced with s_α'. If $CBCF(s_\alpha') \le CBCF(s_\alpha)$ then $CBCF(T') \le CBCF(T)$ after calling LA for T and T'.

Proof: Without any loss of generality, suppose that $s_1, s_2, \ldots, s_\alpha, \ldots, s_\beta, s_{\beta+1}, \ldots, s_a$ are sorted decreasingly with respect to their CBCF's, and also assume that $CBCF(s_{\beta+1}) \le CBCF(s_\alpha') \le CBCF(s_\beta)$. Therefore, the resulting sorted list would be $s_1, s_2, \ldots, s_{\alpha-1}, s_{\alpha+1}, \ldots, s_\beta, s_\alpha', s_{\beta+1}, \ldots, s_a$. By comparing T' with T, it is clear that every sub-placement in T is replaced with another with lower or equal CBCF. Also, it must be noted that in a placement the number of wires going above a sub-placement is a function of the position of that sub-placement in the sorted list and not a function of its CBCF. Therefore, that replacement would result in the decrease of the cut-width of T' compared to T. ■

Theorem 8 Two trees T and T' are similar except in certain sub-trees, $s_1, s_2, \ldots, s_\delta$ in T and $s_1', s_2', \ldots, s_\delta'$ in T'. If $0<i<\delta+1$, $CBCF(s_i') \le CBCF(s_i)$ then $CBCF(T') \le CBCF(T)$.

Proof: By replacing one sub-placement at a time and applying Lemma 16 to the resulting placement, the above claim is proved. ■

Lemma 17 For a mapped tree circuit, the placement given by LA is at most 2X away from the corresponding minimum total-area implementation.

Proof: Regardless of the linear placement order, the total width W is constant, (cf. Observation 1.) For a given library, h and β are constants too. Call the minimum possible cut-width of a tree c_{min}. From Figure 8 and Observation 1, the total-area of the minimum area implementation of the circuit is $A_{min}=W\times(h+\beta\times c_{min})$. The cut-width of the placement built by LA (c_{LA}) is at most 2X larger than c_{min} (Lemma 3.) Thus, $A_{LA}=W\times(h+\beta\times c_{LA})$, is at most 2X away from the optimum value (A_{min}). This upper bound is reached where $h=0$ and T is a balanced tree [Le82]. ■

Lemma 18 The total-area of every implementation in the three-dimensional solution curve is at most 2X larger than the total-area of its corresponding optimal placement.

Proof: The proof follows from Lemma 17 and the method by which SiMPA-D calls LA and prunes the inferior points. ■

Theorem 9 For any total-area optimal solution S (the solution found by SiMPA-E), there exists a solution S' on SiMPA-D's final curve with a less than or equal gate-area and a total-area that is at most 2X larger than that of S.

Proof: In SiMPA-D's final curve, if there exists a solution with the same mapping of S, the proof directly follows from Lemma 17. Otherwise, assume S'' to be a solution built by running LA on the mapping of S. The total-area of S'' is at most 2X larger than S according to Lemma 17. S'' must be inferior with respect to a solution in the final curve S'. The mapping of S'' differs from the mapping of S' in certain minimal sub-trees. These mappings and their corresponding LA placements have been dropped during SiMPA-D by their corresponding solutions in S' which posses lower gate-area and cut-width. Therefore considering Theorem 8, the total-area of S' is smaller than or equal to the total-area of S'', and hence within 2X of S. ■

Theorem 10 The memory usage of SiMPA-D is $O(n^2\times log(n))$.

Proof: According to Lemma 15, the number of points in each solution curve is $O(n \times log(n))$. There are at most n solution curves, one for each node in the decomposed circuit. That proves the above claim. ■

Theorem 11 The worst-case runtime complexity of SiMPA-D is $O(n^{k+1} \times log^k(n))$, where k is the maximum number of inputs to a gate in the library.

Proof: In SiMPA-D, the merge and the prune operations are executed n times, once for every node of the decomposed circuit. Theorem 7 states that each call to the merge operation costs $O(n^k \times log^k(n))$. In addition, the prune operation works by sorting the solutions, so it takes $O(n \times log(n))$. Obviously, for any reasonable library $k \geq 1$, and therefore the merge operation dominate the prune operation in terms of worst-case runtime. Consequently, the overall worst-case runtime of SiMPA-D is, as claimed, $O(n^{k+1} \times log^k(n))$. ■

III.3. Improving Performance

In the previous sub-sections, it was proved that SiMPA-E and -D both have the worst-case polynomial runtime and memory complexities, with high-order polynomials. By using the bucketing technique, those worst-case complexities can be reduced considerably with a small penalty in terms of the quality of the results. Lemma 11 states that the number of distinct gate-area values is bounded by $O(n)$. However, the quality of the results is not compromised by much if a constraint on the resolution for distinguishing the gate-area values is imposed.

Suppose that in every solution curve, the gate-area axis is divided into a maximum number of m buckets (intervals). Consequently, the solutions with gate-areas within the same bucket are considered to have the same gate-area. This technique bounds the number of distinct gate-areas by $O(m)$. Therefore, the worst-case runtime and memory complexities of SiMPA-E and –D are considerably reduced, as shown below. Note that m is not a function of the problem size, hence $O(m^k) = O(1)$.

	Upper-bound on the number of non-inferior points in every solution curve	**Worst-case Runtime Complexity**	**Worst-case Memory Complexity**
SiMPA-E	$O(m)$	$O(m^k \times nlog(n))$	$O(m \times n)$
SiMPA-D	$O(m \times log(n))$	$O(m^k \times n\ log^k(n))$	$O(m \times n\ log(n))$

IV. FLOORPLAN-DRIVEN SiMPA

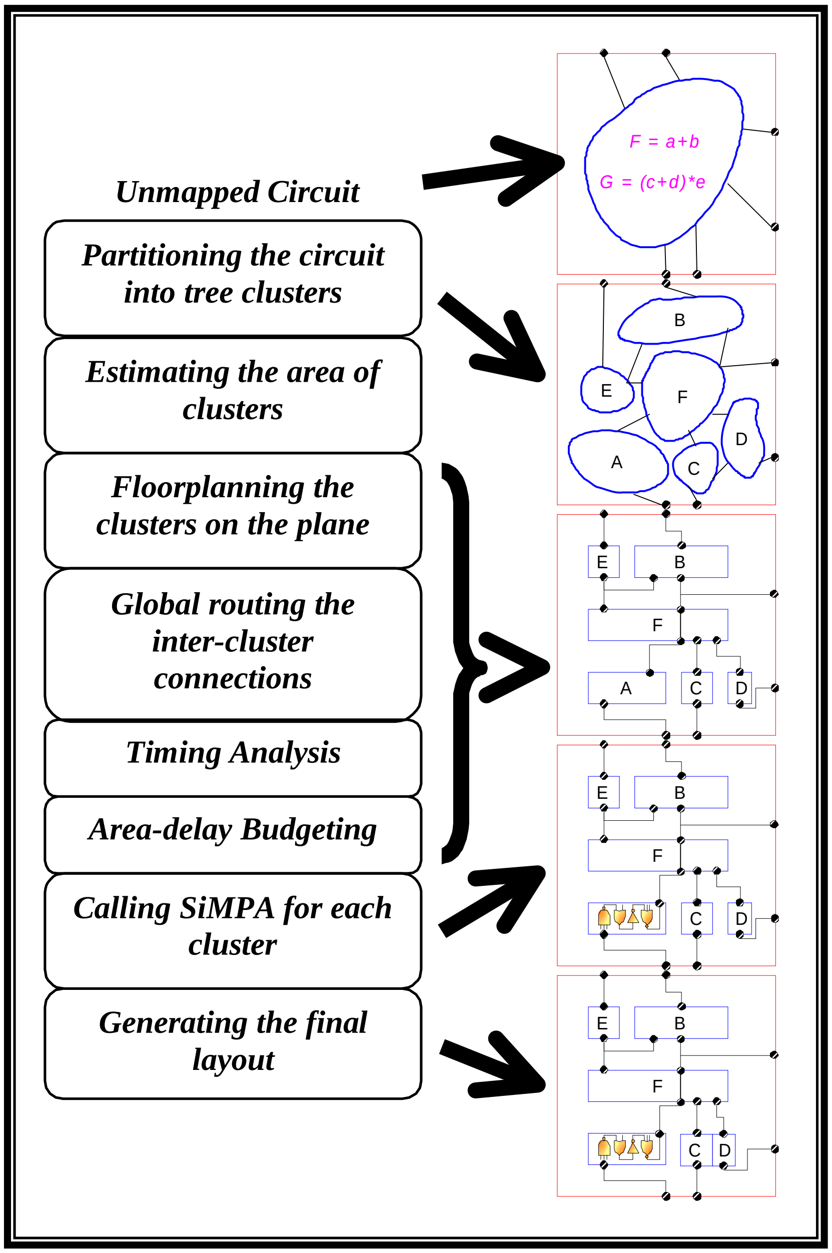
FPD-SiMPA (Floorplan-Driven Simultaneous Technology Mapping and Linear Placement Algorithm) extends SiMPA-E and SiMPA-D and effectively employs their capabilities to achieve high quality results for any direct acyclic circuit. FPD-SiMPA consists of tree partitioning, floorplanning, global routing, slack assignment, and the SiMPA step. It can also be changed to a re-synthesis flow by applying a set of simple modifications.

Definition 9 The *Primary graph* is defined to be the DAG corresponding to the input circuit.

Definition 10 Considering that the clustering step partitions the primary graph into a set of maximal tree clusters, the *secondary graph* is defined to be the reduction of the primary graph in which every cluster is replaced with a node. All edges that entirely lie inside the cluster disappear from the secondary graph.

IV.1. FPD-SiMPA in Detail

FPD-SiMPA consists of a set of design steps shown in Figure 15. Below, each step is graphically depicted in the right side column by showing the effect of the corresponding transformation on the circuit. The detailed desription of each step is given later in this section.



Unmapped Circuit
Partitioning the circuit into tree clusters
Estimating the area of clusters
Floorplanning the clusters on the plane
Global routing the inter-cluster connections
Timing Analysis
Area-delay Budgeting
Calling SiMPA for each cluster
Generating the final layout
F = a+b
G = (c+d)*e
A
B
C
D
E
F

Figure 15 FPD-SiMPA Flow

Tree partitioning: As the first step, FPD-SiMPA clusters the primary graph into locally maximal trees (clusters). Subsequently, a secondary graph is generated out of the primary graph by replacing each cluster with a node. Each cluster is later implemented separately.

Area estimation: In this step, all the clusters are unmapped and their areas are estimated using area predictors. One such predictor uses the number of literals in the expression describing the functionality of the cluster. Another more accurate, although more time-consuming, predictor uses the area of the minimum area technology mapping of the cluster. The minimum area mapper runs quite fast for trees and it is indeed a practical approach for predicting the area. In fact, the area predictor used at this stage is not required to be highly accurate. The estimated values are used by the floorplanner to place the secondary graph on the plane. Consequently, small errors in the area estimates of individual nodes do not have a big impact on the final layout.

Earlier, it was mentioned that the tree clusters must be maximal in the sense that they cannot grow in the primary graph while maintaining their tree structure. In some cases, however, a cluster may become too large which is not desirable. This is mainly because clusters are placed linearly and therefore the large ones may become very long and narrow compared to the others. This scenario may generate large dead spaces and subsequently a considerable amount of area and delay penalty in the final layout. Figure 16 demonstrates this situation and depicts a simple solution used in this work for tackling this problem. Having estimated the area of each cluster, FPD-SiMPA detects the relatively large clusters and simply breaks them into smaller trees such that the variations in their areas are reduced.

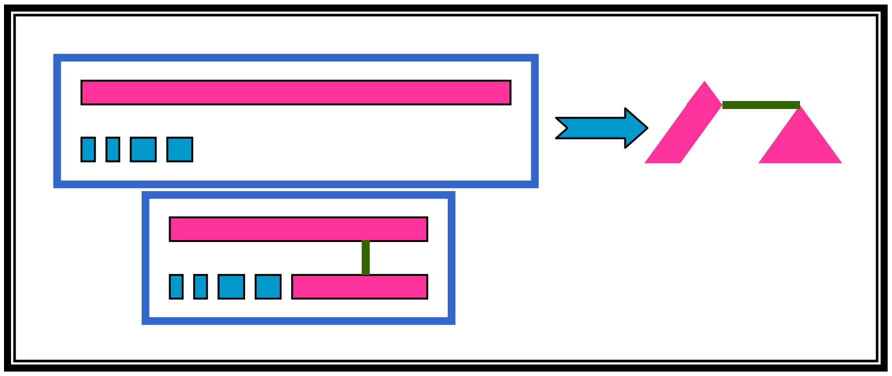


Figure 16 Breaking a Large Tree

Floorplanning: Using the estimated areas for nodes of the secondary graph, the clusters are properly placed on a two-dimensional plane. A range of height and width is assigned to each cluster based on the corresponding estimated area. Due to the lack of exact implementation information at this stage, the assigned height and width ranges are considered as soft constraints, i.e. the clusters are treated as flexible shape blocks.

Global routing: Prior to timing analysis, it is essential to have the global routing information available. This step produces the global connection topology and routes for multi-pin nets on the floorplan solution, and hence provides accurate estimates of the routing capacitances.

Timing analysis: The timing information is required by the delay budgeting step later during the FPD-SiMPA flow. Using a delay predictor, the timing analysis begins with estimating the delay of the nodes in the secondary graph. This prediction can be performed using the logic-level description and the loading information. The load ahead of a node consists of two parts, the input capacitance of the fanout node and the load associated with the wires connecting the node to its fanouts. Every node's input capacitance is approximated with the input capacitance of a moderate size two-input NAND gate of the library. Since the floorplan and the global routing information are known in our design flow, the wire loads can be calculated accurately. As expected in deep sub-micron technologies, global wires constitute a major part of the overall interconnections. Therefore, during the timing analysis the error due to employing delay predictors and applying approximations is not generally significant because in FPD-SiMPA the global wires are planned earlier in the design flow.

Delay budgeting: Using the results of the timing analysis step, FPD-SiMPA measures the slack of every path in the secondary graph. For a path, p, the slack, S_p, is defined as the difference between the required time and the arrival time at the output of p. Then, the S_p values are used to assign a slack value to each node of the secondary graph. The slack assigned to a node represents the degree of freedom (in terms of delay) which is allowed during the implementation of that node.

Observation 2 In the secondary graph, for an arbitrary path, say *p*, the sum of the assigned node slacks is not to exceed the path slack. In other words:

$$\sum_{i=1}^{k} S_i \leq S_p$$

where k is the number of nodes located on p, and S_i is the slack value assigned to the *i*'th node.

The path slacks can be distributed in many ways among the nodes on the path. No matter what method is used the above equation should remian valid for all the paths in the secondary graph.

Area budgeting: In the standard-cell design style, all the gates are aligned on rows and the distance between every two neighboring rows is determined by the routing congestion in that area. After the area estimation and floorplanning steps, FPD-SiMPA identifies the tallest macro-cell in each row.

Definition 11 The difference between the height of a cell (cluster) and the height of the tallest cell in the same row is called the *height slack*.

The height slack of a cluster is used to determine how much space is available on top of it. That space is traded for smaller width and/or higher performance with no adverse effect on the height of the design.

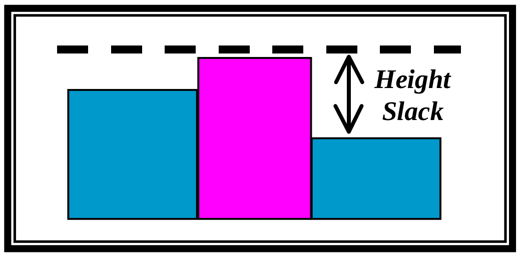


Figure 17 Height Budget Assignment

Definition 12 The difference between the width of a row and the longest row in the layout is called the *width slack* of that row.

The width slack of a row is distributed among the clusters of that row. Each cluster uses its width slack to determine how much space is available on its sides for trading for lower height and/or higher performance with no adverse effect on the width of the whole design.

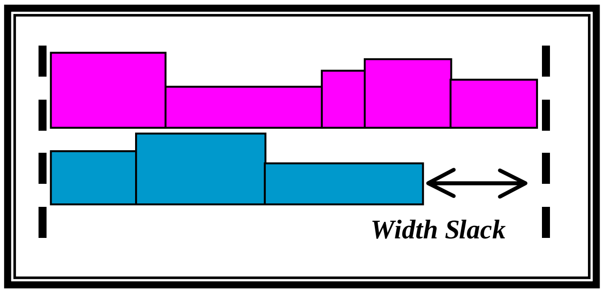


Figure 18 Width Budget Assignment

SiMPA: Depending on the design objective and constraints, SiMPA-E and/or SiMPA-D are called for every node of the secondary graph. SiMPA (either –E or –D) returns a set of implementations for every tree cluster, each with different specifications. Based on the estimated geometrical and timing values and slacks for every cluster a proper implementation is chosen. It is clear that SiMPA-E solely focuses on the geometrical aspects of the design and returns a set of implementations including the minimum total-area implementation. So, it may not be appropriate to employ SiMPA-E in places where trade-off between area and delay is required. However, it is a good choice for those clusters which are not delay critical and their minimum implementations are required. SiMPA-D, on the other hand, provides the opportunities of trading-off delay for area with a bounded sacrifice on the minimum total-area implementation. So, the application determines which SiMPA algorithm is more appropriate to be used.

Final layout generation: After implementing each cluster, it is possible to see overlaps between the clusters mainly because of the mismatches between the predicted and the actual area values for the clusters. Therefore, a DOMINO-type local optimization algorithm is called to remove cell overlaps and delete empty spaces between cells in the same row. Note that these two conditions could arise since the area estimates used during floorplanning are not exact. After all, detailed routing is performed on the layout.

IV.2. An Example of FPD-SiMPA

In this sub-section, a set of screen shots demonstrate the operations FPD-SiMPA performs on a benchmark (*pcle*). Figure 19, Figure 20, and Figure 21 show the netlist, floorplan, and final layout of *pcle* generated by FDP-SiMPA.

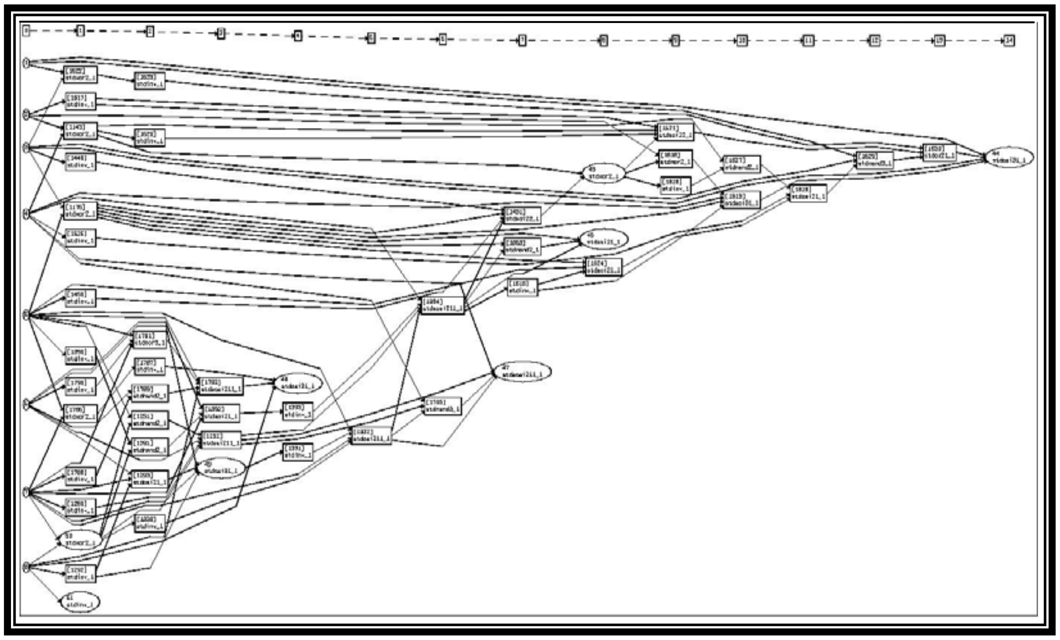


Figure 19 : Netlist of *pcle*

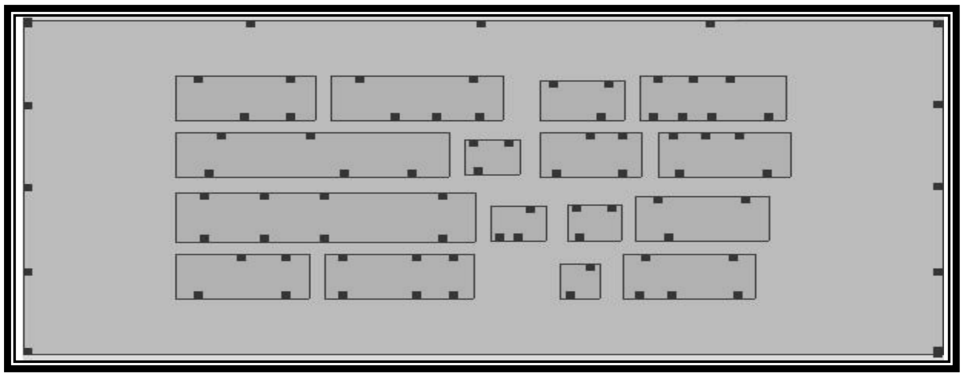


Figure 20 : Floorplanned *pcle*

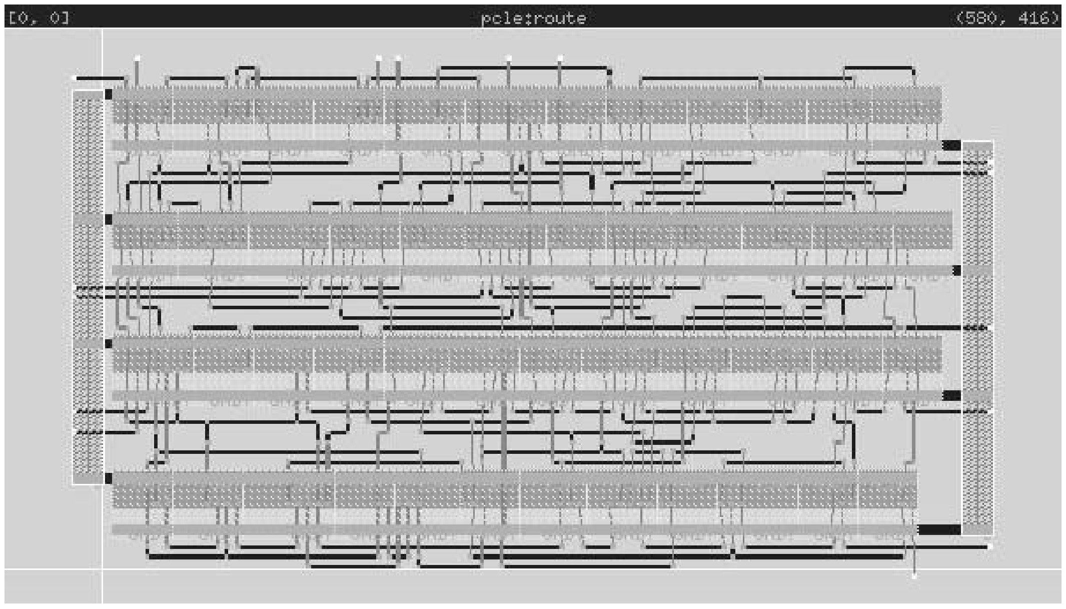


Figure 21 : Final Layout of *pcle*

IV.3. A few Comments about FPD-SiMPA

1. As previously mentioned, FPD-SiMPA decomposes DAG-like circuits into tree clusters, a requirement imposed by the use of SiMPA-E and -D. However, a general floorplan-driven approach can partition the input decomposed circuit into sub-DAG clusters and heuristically implement each separately.
2. Cluster implementations generated by SiMPA-E and -D consist of one-dimensional placements. This restriction is not in contrast with the general two-dimensional IC design style, since the current circuits are decomposed into a large number of clusters and the clusters are placed on a two-dimensional plane prior calling SiMPA-E or -D.
3. In our implementation of FPD-SiMPA, we employed Bear-FP in the floorplanning step. However, FPD-SiMPA may use any floorplanning or two-dimensional placement tool in its floorplanning step. The current circuits are generally so large that they are decomposed into many tree clusters and the

size of each cluster is small compared to the overall circuit. Consequently, a timing-driven global point placement (such as SPEED [RE95]) can be used instead of floorplanning.

V. EXPERIMENTAL RESULTS

FPD-SiMPA has been implemented in the SIS environment [SSLM92]. In this implementation, the 4-parameter delay equation and the Elmore delay model (as presented in Section III.2.1) have been employed to calculate circuit delays. The library used in these experiments is a CASCADE-generated standard cell library in a 0.5u HP CMOS process.

Before presenting the experimental results of FPD-SiMPA, Table 1 and Table 2 show the results of SiMPA-E and SiMPA-D for tree-like circuits. In the left most columns of these tables, the suffix after the name of the tree denotes its corresponding number of inputs. The experimental results show that on average SiMPA-E is able to reduce the total (gate plus wire) area by 14% without any significant impact on their corresponding delays. Table 2 reports that on average SiMPA-D is able to simultaneously reduce the total-delay and the total-area by 25% and 7%, respectively.

Circuit	Conventional				SiMPA-E					
	Gate Area	Cut	Delay	Total Area	Gate Area	Cut	Delay	Total Area	Area Ratio	Delay Ratio
tree6	7260	3	1.71	10428	7260	2	1.82	9372	89.87%	106.43%
tree8	10054	3	0.89	14441	10347	2	0.81	13357	92.49%	91.01%
tree16	17732	4	1.08	28049	18002	2	1.23	23239	82.85%	113.89%
tree20	30536	5	1.55	52744	31224	3	1.62	44849	85.03%	104.52%
tree32	38665	6	1.81	72409	39149	4	1.88	61927	85.52%	103.87%
tree48	66396	7	2.58	133999	68432	4	2.38	108247	80.78%	92.25%
									86.09%	101.99%

Table 1: Experimental Results of SiMPA-E

Circuit	Conventional				SiMPA-D				Ratio	
	Gate Area	Cut	Delay	Total Area	Gate Area	Cut	Delay	Total Area	Area Ratio	Delay Ratio
tree6	9482	3	1.55	13620	10103	2	1.22	13042	95.76%	78.71%
tree8	13444	4	0.75	21266	15154	3	0.61	21767	102.35%	81.33%
tree16	22304	5	0.98	38525	19098	4	0.65	30210	78.42%	66.33%
tree20	51030	6	1.03	95565	54342	5	0.78	93863	98.22%	75.73%
tree32	48468	6	1.27	90767	50015	5	0.89	86390	95.18%	70.08%
tree48	90342	7	1.92	182327	85796	6	1.47	160673	88.12%	76.56%
								Average:	93.01%	74.79%

Table 2: Experimental Results of SiMPA-D

To assess the effectiveness of the simultaneous technology mapping and placement approach on general DAG-structured circuits, the following experiments have been set up and the results are presented in Table 3. The three design flows reported in Table 3 are described below:

- The first flow, introduced as the conventional flow, uses SIS to perform technology mapping, GORDIAN-L [KSJA91] for placement, DOMINO [DJS91] for updating the placement, TimberWolf [SS86] for global routing, and finally YACR [RSS85] for detail routing.
- The second flow, named as the MINCUT flow, employs SIS's technology mapping, followed by clustering the mapped network into maximal tree clusters. Those clusters are floorplanned using

Bear-FP [PK92] and then each is linearly placed by the Yannakakis' algorithm. Afterwards, DOMINO, TimberWolf, and YACR are called for updating the placement, global routing, and detailed routing. In contrast to the previous flow, this flow partitions the circuit into a set of tree clusters and linearly places each. However, it still differs from the FPD-SiMPA flow since the mapping and the placement steps are not performed simultaneously.

- The third flow, introduced as the MINSUM flow, is similar to the second flow except in the linear placement step where a minsum-based linear placement algorithm [Sh79] instead of Yannakakis' MINCUT algorithm is used.

Table 3 demonstrates that the differences between these three runs is rather small. This proves that FPD-SiMPA's effectiveness is due to its properly integrated mapping and placement technique.

Circuits	I. Conventional		II. MINCUT		III. MINSUM		II over I		III over I	
	Area	Delay	Area	Delay	Area	Delay	Area	Delay	Area	Delay
alu2	1550995	18.00	1462677	18.01	1645847	18.50	0.94	1.00	1.06	1.03
apex6	3764845	12.66	3856989	12.83	3843396	11.84	1.02	1.01	1.02	0.94
b9	550940	4.36	527307	4.25	496015	4.17	0.96	0.97	0.90	0.96
cordic	208936	2.65	204672	2.58	213200	2.58	0.98	0.97	1.02	0.97
frg1	528515	6.20	556895	6.24	537911	6.16	1.05	1.01	1.02	0.99
pcle	260000	4.29	255000	4.29	255000	3.99	0.98	1.00	0.98	0.93
z4ml	176085	3.88	173349	3.75	164045	3.75	0.98	0.97	0.93	0.97
C1908	2169180	16.98	2217684	16.80	2065490	17.28	1.02	0.99	0.95	1.02
C880	1589525	13.48	1578927	13.23	1574758	13.74	0.99	0.98	0.99	1.02
C3540	10101300	27.63	9673587	26.09	9749415	26.98	0.96	1.31	0.97	1.34
						Average:	0.99	1.02	0.98	1.02

Table 3: Comparison of Conventional Flows

In Table 4, FPD-SiMPA is quantitatively compared against the conventional flow. All the input circuits are unmapped and therefore the standard FPD-SiMPA flow has been utilized to generate the experimental results. The area and delay reported here are the total chip area and the total chip delay after detailed routing. In this table, the conventional flow is the same as the one introduced in Table 3. In the FPD-SiMPA flow, Bear-FP is used to floorplan the tree clusters using their estimated areas. Having finished the global routing and budgeting steps, SiMPA is called to synthesize the clusters. DOMINO and TimberWolf are then executed to update the chip layout, and finally YACR performs the detailed routing.

In Table 5, the runtime of FPD-SiMPA flow is compared against the runtime of the conventional flow. Two separate comparisons are provided in that table. The first set of runtimes compares the complete FPD-SiMPA and the complete conventional flows against each other. In this case, TimberWolf is the most time consuming step. In order to evaluate the real runtime of SiMPA, the second set of experiments reports the same experiments this time excluding the time taken by TimberWolf. As it can be seen, SiMPA runs faster for some circuits because technology mapping step in the conventional flow for non-tree circuits is much slower than the technology mapping performed by SiMPA for its tree-like clusters. For relatively small size circuits, the overhead of SiMPA is completely absorbed by this factor.

Circuit	conventional		FPD-SiMPA		Ratio	
	area	delay	area	delay		
alu2	2458623	13.13	2016738	8.60	0.82	0.65
alu4	4258415	22.89	3725073	11.89	0.87	0.52
apex6	4554276	12.01	5106345	6.82	1.12	0.57
apex7	1253996	6.45	1316772	4.31	1.05	0.67
b9	619830	3.62	597806	2.57	0.96	0.71
cm150a	225624	3.22	215424	1.42	0.95	0.44
cm151a	167657	2.97	94221	1.46	0.56	0.49
cm162a	229200	2.50	234024	1.82	1.02	0.73
comp	640710	2.84	543582	2.22	0.85	0.78
con1	82530	1.55	92805	0.76	1.12	0.49
cordic	375683	2.74	266000	1.90	0.71	0.69
dalu	6355808	19.45	5958750	14.75	0.94	0.76
duke2	3282904	10.16	2707709	8.64	0.82	0.85
e64	1601895	8.29	1647555	9.13	1.03	1.10
f51m	299712	5.76	369891	4.42	1.23	0.77
frg1	645414	3.34	687990	2.47	1.07	0.74
k2a	7555270	15.96	9421164	12.37	1.25	0.78
lal	560190	4.09	536934	2.60	0.96	0.64
misex3	3374076	12.89	3558875	10.73	1.05	0.83
mux	221200	3.03	233784	1.46	1.06	0.48
pcle	471075	3.66	299936	2.41	0.64	0.66
pcler8	601614	3.90	569734	3.00	0.95	0.77
ritex	337344	2.16	298920	1.84	0.89	0.85
rot	4747557	8.91	4864220	8.20	1.02	0.92
term1	710710	3.92	777777	2.72	1.09	0.69
z4ml	266696	3.45	182360	1.66	0.68	0.48
5xp1	641190	7.66	545846	4.23	0.85	0.55
b12	393419	3.81	355239	1.97	0.90	0.52
bw	1017282	9.33	845445	5.77	0.83	0.62
clip	688974	7.85	767637	3.61	1.11	0.46
misex2	517047	4.03	513918	2.48	0.99	0.62
rd53	225624	3.16	154997	2.00	0.69	0.63
rd73	335243	3.78	345703	2.73	1.03	0.72
rd84	857033	7.01	766945	4.61	0.89	0.66
sao2	815022	6.20	777849	3.49	0.95	0.56
table3	5014467	55.17	5355504	53.51	1.07	0.97
vg2	543235	3.66	462223	2.67	0.85	0.73
C17	40050	0.78	40762	0.37	1.02	0.47
C432	2809631	11.91	1328756	12.28	0.47	1.03
C1908	4287597	17.46	2538349	14.17	0.59	0.81
C880	2672775	9.53	2376701	10.74	0.89	1.13
C1355	2452165	8.84	2683512	7.84	1.09	0.89
C499	2304419	8.64	2702085	6.55	1.17	0.76
C2670	4678425	8.40	4596994	7.17	0.98	0.85
C3540	10101300	27.63	6984801	22.20	0.69	0.80
C5315	11231622	16.03	10201209	16.22	0.91	1.01
C7552	12738138	24.53	14157864	11.77	1.11	0.48
			Average:		0.93	0.71

Table 4: Experimental Results on FPD-SiMPA

	Including TimberWolf			Excluding TimberWolf		
	Conventiona	FPD-SiMPA	Diff	Conventional	FPD-SiMPA	Diff
alu2	146	178	21.9%	48	42	-12.5%
apex7	63	67	6.3%	10	24	140.0%
cm150a	15	22	46.7%	4	4	0.0%
cm151a	11	15	36.4%	2	3	50.0%
cm162a	16	17	6.3%	3	3	0.0%
duke2	146	152	4.1%	27	56	107.4%
k2a	519	753	45.1%	159	435	173.6%
rot	215	312	45.1%	36	121	236.1%
table3	243	265	9.1%	144	96	-33.3%
C1908	255	291	14.1%	109	73	-33.0%
C880	155	199	28.4%	37	32	-13.5%
C1355	191	243	27.2%	74	73	-1.4%
C3540	663	711	7.2%	255	262	2.7%
		Average:	22.9%		Average:	47.4%

Table 5: Runtime Comparison

The above experimental results show that FPD-SiMPA is able to improve the delay by as much as 53%, and reduce the area by more than 50%. On average, the performance improvement is 29% while the total-area is reduced by 7%. For some circuits, such as *C7552*, FPD-SiMPA has been able to reduce the delay by more than 50% while keeping the area in an acceptable range (10% area increase). For many other circuits, it has reduced both the delay and the area. However, in a rare case like *e64*, FPD-SiMPA has been unable to improve either delay or area. An explanation for such a result is that the missed inter-cluster optimization opportunities due to the tree-clustering step are so large that floorplanning and SiMPA cannot compensate for them. A possible solution for such cases is to use a non-tree partitioning technique that enables FPD-SiMPA to perform optimization across the tree boundaries (see Section IV.3).

VI. CONCLUSION

This paper presents two novel algorithms SiMPA-E and SiMPA-D, which perform simultaneous technology mapping and linear placement for tree-structured circuits targeting minimum total chip area and/or delay. The proposed dynamic programming based algorithms generate and propagate solution curves which contain a set of non-inferior implementations and provide the designer with a variety of geometrical and timing trade-offs. This paper also describes a new methodology FPD-SiMPA, which exploits these algorithms to synthesize high-performance sub-half micron logic circuits. This methodology is capable of controlling the trade-off between area and delay, and produces circuit implementations with highly predictable performance characteristics. The experimental results have proved the effectiveness of our proposed flow and algorithms.

VII. REFERENCES

[Bu97] R. Bushroe “Panel: physical design and synthesis: merge or die," In *Proceeding 34th Design Automation Conference*, pages 238-239, June 1997.

[CP92] K. Chaudhary, and M. Pedram “A near-optimal algorithm for technology mapping minimizing area under delay constraints,” In *Proceeding 29th Design Automation Conference*, June 1992.

[CH94] W. Chuang and I. N. Hajj, “Delay and area optimization for compact placement by gate resizing and relocation,” In *IEEE/ACM International Conference on Computer-Aided Design*, pages 145-148, 1994.

[Ch84] F. R. K. Chung, “On optimal linear arrangements of trees,” In *Comput. Math. Applic. 10*, pages 43-60, 1984.

[DJS91] K. Doll, F. M. Johannes, and G. Sigl, “DOMINO: Deterministic placement improvement with hill-climbing capabilities,” in *IFIP International Conference VLSI*, 1991.

[El48] W. C. Elmore, “The transient response of damped linear network with particular regard to wideband amplifiers,” In *Journal of Applied Physics 19*, pages 55-63, 1948.

[GJ79] Michael R. Garey and David S. Johnson, “Computers and Intractability”, Bell Telephone Lab. Inc., 1979.

[HS96] G. D. Hachtel, and F. Somenzi, *Logic synthesis and verification algorithms.* Kluwer Academic Publishers, Norwell, MA, 1996.

[Ke87] K. Keutzer, “DAGON: Technology mapping and local optimization,” In *Proceedings of Design Automation Conference,* pages 341-347, June 1987.

[KSF94] L. N. Kannan, P. R. Suaris, and H. Fang, “A methodology and algorithms for post-placement delay optimization, “In *31st ACM/IEEE Design Automation Conference*, 1994.

[KSJA91] J. M. Kleinhans, G. Sigl, F. M. Johannes, and K. J. Antreich, “GORDIAN: VLSI Placement by Quadratic Programming and Slicing Optimization,” in *IEEE Transactions on Computer-Aided Design*, Vol. 10, No. 3, March 1991.

[Le82] T. Lengauer, “Upper and lower bounds on the complexity of the min-cut linear arrangement problem on trees,” In *SIAM Journal of Algorithm Disc. Meth.,* vol. 3, no. 1, pages 99-113, March 1982.

[LPPD93] S. Liu, K. Pan, M. Pedram, and A. M. Despain, “Alleviating routing congestion by combining logic resynthesis and linear placement,” In *European Conference on Design Automation,* pages 578-582, 1993.

[LSP97] J. Lou, A. H. Salek, and M. Pedram, “An exact solution to simultaneous technology mapping and linear placement problem,” In *Proceedings of the International Conference on Computer-Aided Design,* pages 671-675, November 1997.

[LSP98] J. Lou, A. H. Salek, and M. Pedram, “An integrated flow for technology remapping and placement of sub-half-micron circuits,” In *Proceedings of Asia and South Pacific Design Automation Conference,* pages 295-300, February 1998.

[PB91a] M. Pedram, and N. Bhat “Layout driven technology mapping,” In *Proceedings of the 28th Design Automation Conference,* pages 99-105, June 1991.

[PB91b] M. Pedram, and N. Bhat “Layout driven logic restructuring / decomposition,” In *Proceeding International Conference on Computer-Aided Design*, pages 134-137, 1991.

[PK92] M. Pedram, and E. Kuh, “BEAR-FP: A Robust Framework for Floorplanning,” in *International Journal of High Speed Electronics*, Vol. 3, No. 1, pages 137-170, 1992.

[RE95] B. M. Riess, and G. G. Ettelt, “Speed: Fast and efficient timing driven placement,” in *IEEE International Symposium on Circuits and Systems*. Pp. 377-380, 1995.

[RSS85] J. Reed, A. Sangiovanni-Vincentelli, and M. Santamauro, “A new symbolic channel router: YACR2,” in *IEEE Transactions on Computer Aided Design*, Vol. CAD-4, pages 208-219, March 1985.

[Ru89] R. Rudell “Logic synthesis for VLSI design,” *Memorandum UCB/ERL M89/49, Ph.D. Dissertation*, University of California at Berkeley, April 1989.

[SS86] C. Sechen, and A. Sangiovanni-Vincentelli, “TimberWolf 3.2: A new standard cell placement and global routing package,” *in Proceedings of 23rd Design Automation Conference*, pages 432-439, 1986.

[SSLM92] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vicentelli, “SIS: A System for Sequential

Circuit Synthesis," *Memorandum UCB/ERL M92/41*, Electronics Research Laboratory, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1992.

[Sh79] Y. Shiloach, "A minimum linear arrangement algorithm for undirected trees," In *SIAM Journal of Computers 8*, pages 15-32, 1979.

[SRRJ97] G. Stenz, B. M. Riess, B. Rohfleisch, and F. M. Johannes, "Timing driven placement in interaction with netlist transformations," In *International Symposium on Physical Design*, pages 36-41, April, 1997.

[TMBW90] H. J. Touati, C. W. Moon, R. K. Brayton, and A. Wang "Performance oriented technology mapping," In *Proceedings of the Sixth MIT Conference Advanced Research VLSI*, pages 79-97, April 1990.

[VP95] H. Vaishnav, and M. Pedram, "Logic extraction based on normalized netlengths," In Proceedings *of the International Conference on Computer Design*, October 1995.

[Ya85] M. Yannakakis, "A polynomial algorithm for the min-cut linear arrangement of trees," In *Journal of the Association for Computing Machinery*, vol. 32, no. 4, pages 950-988, October 1985.