An Efficient Approach to Multilayer Layer Assignment with an Application to Via Minimization

Chin-Chih Chang and Jason (JingSheng) Cong, *Senior Member, IEEE*

***Abstract*—In this paper we present an efficient heuristic algorithm for the post-layout layer assignment and via minimization problem of multilayer gridless integrated circuit (IC), printed circuit board (PCB), and multichip module (MCM) layouts. We formulate the multilayer layer assignment problem by introducing the notion of the extended conflict-continuation (ECC) graph. When the formulated ECC graph of a layer assignment problem is a tree, we show that the problem can be solved by an algorithm which is both linear time and optimal. When the formulated ECC graph is not a tree, we present an algorithm which constructs a sequence of maximal induced subtrees from the ECC graph, then applies our linear time optimal algorithm to each of the induced subtrees to refine the layer assignment. Our experiments show that, on average, the number of vertices of an induced subtree found by our algorithm is between 12% and 34% of the total number of vertices of an ECC graph. This indicates that our algorithm is able to refine a large portion of the layout optimally on each refinement, thus, producing highly optimized layer assignment solutions. We applied this algorithm to the via minimization problem and obtained very encouraging results. We achieved 13%–15% via reduction on the routing layout generated by the V4R router [1], which is a router known to have low usage of vias. Our algorithm has been successfully applied to routing examples of over 30 000 wire segments and over 40 000 vias. Finally, we outline how our layer assignment algorithm can also be used for delay and crosstalk minimization in high-performance IC, PCB, and MCM designs.**

I. Introduction

As very large scale integration (VLSI) technology advances, interconnection and packaging technologies have become bottlenecks in system performance. For advanced integrated circuit (IC) designs, four to six routing layers are commonly used in high-performance and high-density designs. Multichip module (MCM) technology was developed to increase packing densities, eliminate the packaging level of interconnections, and provide more layers for routing. In both the multilayer IC and MCM designs, the designer or automatic layout tools may use variable widths and spacings to optimize performance. This often results in multilayer gridless layouts.

Because multilayer gridless routing is a complex three-dimensional general area routing problem, it is not easy to design a router which can simultaneously optimize many different design objectives, such as wire length, area, delay, crosstalk, number of vias, etc. Therefore, it is important to perform certain post-layout optimizations to help the router meet various design constraints and produce better routing solutions.

One of the important post-layout optimization techniques is layer assignment, in which wire segments in a routing solution are reassigned to appropriate layers to achieve certain optimization objectives. Layer assignment has become an interesting topic for the following two reasons: first, it preserves the wire lengths and topologies during optimizations; second, it provides considerable flexibility for optimizations of vias, crosstalk, and delays. In this paper, we present an efficient multilayer layer assignment algorithm for both grided and gridless layout with focus on its application to the via minimization problem.

The via minimization problem is that of minimizing the number of vias in a VLSI layout. A via is a hole filled with conductive materials to connect wire segments on different layers in a VLSI layout. Because vias often reduce the manufacturing yield, degrade the circuit performance, and increase layout area (more difficult to compact routing solutions, e.g., refer to [2]), it is desirable to minimize the number of vias without affecting routability.

The via minimization problem was first studied for two-layer VLSI layouts. There are two approaches for the two-layer via minimization problem: unconstrained via minimization (or topological via minimization) [3], [4] and constrained via minimization [5]–[13]. Topological via minimization computes both the topologies and the layer assignments of all the nets before detailed routing to minimize the overall via count. However, topological via minimization may affect routability considerably and is usually not used in practice. Moreover, the two-layer topological via minimization problem was shown to be NP-hard [4]. On the other hand, the constrained via minimization problem optimizes an *existing* routing solution by only changing the layer assignments of the wire segments. It is also referred to as the *layer assignment* problem.

For the two-layer constrained via minimization problem for Manhattan layouts with junction degrees less or equal to three, polynomial time optimal algorithms have been developed [6], [8], [9], [12]. These algorithms transform the problem into the planar maximum cut problem, which is solvable in polynomial time. However, if the layout is not Manhattan with junction degrees less or equal to three, the two-layer constrained via minimization problem was shown to be NP-hard [13]–[15].

Manuscript received February 11, 1998. This work was supported in part by DARPA/ETO under Contract DAAL01-96-K-3600 managed by the U.S. Army Research Laboratory, and a grant from Intel under the 1996–1997 California MICRO program. This paper was recommended by Associate Editor C.-K. Cheng.

C.-C. Chang is with the Computer Science Department, University of California, Los Angeles, CA 90095 USA (e-mail: cchang@cs.ucla.edu).

J. Cong is with the Computer Science Department, University of California, Los Angeles, CA 90095 USA (e-mail: cong@cs.ucla.edu).

Publisher Item Identifier S 0278-0070(99)02968-1.

There is also a considerable amount of research work done on the multilayer constrained via minimization problem. Chang and Du first proved that the three-layer constrained via minimization problem on Manhattan routing is NP-hard [16]. They also proposed a heuristic algorithm which checks vias one by one and conducts a local search up to n levels on the via-crossing graph to eliminate the via being checked (in their implementation, n was set to be two). Fang *et al*. [17] proposed a two-phase heuristic algorithm. They use heuristic ordering and backtracking to assign the layers of wire segments one by one and then do local perturbations to eliminate vias greedily. However, when one via is eliminated, other vias might be introduced. Ahn and Sahni [15] proved that the three-layer constrained via minimization problem remains to be NP-hard for Manhattan routing even if the routing is restricted to HVH channel routing. They proposed a track-by-track heuristic algorithm for layer assignment in the HVH constrained via minimization problem.

The existing methods for multilayer constrained via minimization suffer from one or more of the following problems:

- handles only a fixed number of layers;
- assumes a grid-base routing solution;
- cannot produce good solutions due to very limited range of local search;
- cannot scale to large designs efficiently.

All the experimental results on constrained via minimization reported in the literature are on small test cases with only a few hundred vias.

In this paper, we introduce the notion of the extended conflict-continuation (ECC) graph that abstracts the connectivity relations of a given layout for the layer assignment problem. The ECC graph is general enough to handle gridless layouts with any number of routing layers. When a formulated ECC graph is a tree, we show that the layer assignment problem can be solved in linear time optimally by a dynamic programming technique. For the general case of the layer assignment problem where the ECC graph is not a tree, our algorithm constructs a sequence of induced subtrees from the ECC graph and applies our linear time optimal algorithm to each induced subtree. Our experiments show that the average number of vertices of the induced subtrees constructed by our algorithm is very large, containing 12%–34% of the total number of vertices of the ECC graph. It indicates that our algorithm is able to *refine a large portion of the layout optimally each time*, which leads to highly optimized layer assignment solutions. The application of our layer assignment algorithm to the constrained via minimization problem is very successful. We have achieved 13%–15% via reduction on the routing solutions generated by the V4R router [1], which is a router known to have low usage of vias. Our algorithm scales well for handling large designs; it has been successfully applied to routing examples of over 30 000 wire segments and over 40 000 vias. An extended abstract of this work was presented at the 1997 Design Automation Conference [20].

The applications of our algorithm are not limited to the via minimization problem because our formulation is very general and can consider optimization objectives other than via minimization. At the end of this paper, we will show that our layer assignment algorithm can also be used for delay and crosstalk minimization in high-performance IC, printed circuit board (PCB) and MCM designs, when proper cost functions are used.

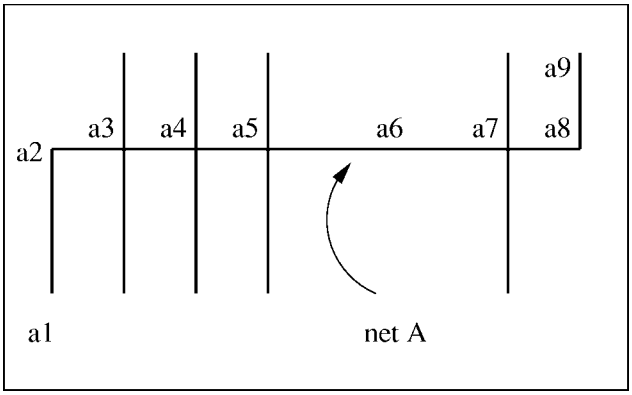

Fig. 1. Wire segmentation.

II. Problem Formulation

Given a valid K-layer layout solution, each net is divided into a set of wire segments. We assume no vias are allowed within a wire segment. Therefore, each wire segment must be assigned to a single layer, while vias can only be used to connect different wire segments of the same net. The designer may specify the segmentation of wires to impose certain layout constraints and control the tradeoff between the flexibility and complexity of the layer assignment problem. For example, Fig. 1 shows a simple layout. Assume that the wires connecting points $a1$–$a9$ are of the same net; all the other vertical lines are of different nets. A natural way of segmenting the wires is to break the nets at the points where the horizontal and vertical wires meet (cf. [6]). In this case, we will break net A into segments $(a1, a2)$, $(a2, a8)$, and $(a8, a9)$. Another possible choice is to break the nets whenever there is enough space to insert vias (cf. [9]). Suppose that there is enough space at point $a6$ to put a via between $a5$ and $a7$. In this case, we will further break $(a2, a8)$ into $(a2, a6)$ and $(a6, a8)$. In the first case, we have less flexibility to insert vias, but smaller problem size because we have fewer segments to consider. In the second case, we have less restriction on inserting vias, but more wire segments for the optimization problem. There are also other possible ways to break nets. For example, we might want to restrict the maximum length of a single wire segment. It is also possible that we want to restrict the minimum length such that we do not to break wires too often. All these choices will have impacts on the flexibility and complexity. Depending on the design, the designer may prefer one option to another based on the consideration of flexibility they want and the complexity they can handle. Our algorithm can handle any given wire segmentation without any restrictions.

After the wire segmentation is done, it is not difficult to see that there are basically two kinds of relations among wire segments in the layer assignment problem: wires that cannot be put in the same layer and wires that must be connected to

each other. The first is called the *conflict relation*; it occurs among wire segments of different nets overlapping with each other (when layers are ignored). The second is called the *continuation relation*; it occurs among wire segments with common end points of the same net.

Pinter [9] proposed the conflict-continuation graph which captures the above relationships among wires during his formulation of the two-layer via minimization problem. In his formulation, a *free run* is a maximal piece of wire that does not overlap any other wire, and can accommodate at least one via. A *"wire segment" is a piece of wire connecting two free runs*. Each vertex $v \in V$ represents wire segment v. The edge set $E = E^c \cup E^f$ represents two kinds of relationships among wire segments. The set E^c is the set of *continuation edges*. An edge $e_{u,v} \in E^c$ exists between vertices u and v if, and only if, u and v are in the same net and are connected to each other. Assigning u and v to different layers will require a via to be inserted. The set E^f is the set of *conflict edges*; an edge $e_{u,v} \in E^f$ exists between vertices u and v if, and only if, u and v are in different nets and cannot be assigned to the same layer.

The conflict-continuation graph is sufficient for the representation of the two-layer via minimization problem, but it is not suitable for multilayer representation when stacked vias are allowed.[1] For example, consider a layout which consists of three wire segments: wire segments a and b are of the same net and connected by a continuation edge; wire segment c is from a different net and overlaps with the intersection of a and b, thus, conflicting with a and b. Because stacked vias are allowed, it is legal to assign a to layer 1, and b to layer 3. In this case, if we assign c to layer 2, it can pass the conflict checking imposed by the conflict-continuation graph because it is assigned to a layer different from those of a and b. However, there is a conflict between c and the stacked via which connects a and b.

We address the above problems by extending the concept of conflict-continuation graphs so that it can handle the multilayer layer assignment problem properly. In our ECC graph, the vertex set contains *via vertices* in addition to *wire segment vertices*.[2] Each via vertex is a possible location to insert a via. It connects two or more wire segments of the same net in the given layout. It may overlap with some wires, but our formulation can represent these conflicts. A via vertex can also have a layer assignment. The layer assignment for a via is a pair of integers (l_l, l_u), $l_l \leq l_u$, indicating that a via spans from layer l_l to layer l_u. Note that we can easily represent vias connecting more than one layer (stacked vias) under this representation. If no stacked vias are allowed, we restrict $l_u \leq l_l + 1$. For the sake of simplicity in later explanation, an integer pair (l_l, l_u) is encoded into one integer. When we say that we assign the via vertex to some layer j, we mean that j will be decoded back into the integer pair.

[1]However, we can show that this graph can be used if no stacked vias are allowed, and the layout will not put two vias too close.

[2]Our definition of wire segment is somewhat different than Pinter's. In Pinter's definition, wire segments are separated by "free runs," which are also wires. A via can be inserted in a "free run," but the location is not determined. In our definition, wire segments are connected end to end, and vias can only be inserted in the end points of wire segments.

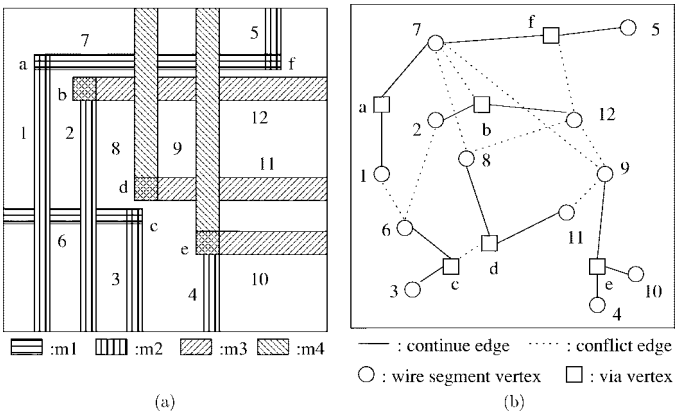

Fig. 2. (a) A given layout and (b) the corresponding ECC graph.

For a K-layer layer assignment problem, we encode (i, j) to $(2K-j+i+1)(j-i)/2+i$. It is easy to show that this encoding function forms a one-to-one mapping between integer pairs (l_l, l_u) with $1 \leq l_l \leq l_u \leq K$ to integers in $[1, K(K+1)/2]$. An simple example on the encoding for $K = 4$ is shown below

$$\begin{array}{cccc} (1,1) & (1,2) & (1,3) & (1,4) \\ & (2,2) & (2,3) & (2,4) \\ & & (3,3) & (3,4) \\ & & & (4,4) \end{array} \quad \begin{array}{c} \\ \Rightarrow \\ \\ \Leftarrow \end{array} \quad \begin{array}{cccc} 1 & 5 & 8 & 10 \\ & 2 & 6 & 9 \\ & & 3 & 7 \\ & & & 4. \end{array}$$

We use $l(v)$ to denote the number of possible assignments that v can be assigned. If v is a wire segment vertex, $l(v) = K$. If v is a via vertex that can span up to s layers, $l(v) = \sum_{0 \leq i \leq s-1}(K - i)$.

In the ECC graph formulation, the continuation edge set contains only via-to-segment continuation edges. A *continuation edge* between two vertices exists if it connects a via vertex and a wire segment vertex of the same net (and the via and wire segment are adjacent). The conflict edge set, however, contains segment-to-segment, segment-to-via, and via-to-via conflict edges. A *conflict edge* exists if the two vertices it connects cannot be assigned to the same layer. A conflict edge $e_{u,v}$ between the wire segment vertex u and another vertex v is redundant and can be removed if u connects through a continuation edge to a via vertex w, and w has a conflict edge to v. This is because u and w are connected by a continuation edge, the layers which via vertex w spans must contain the layer assigned to u. Since the conflict edge $e_{w,v}$ puts the constraint that the layer(s) which w spans will not overlap with the layer(s) occupied by v, the constraint between u and v is also implied.

To give an example for our representation, let us consider the layout shown in Fig. 2(a). Wire segments are labeled 1–12; via candidates are labeled a–f. Note that e is a stacked via connecting $m2$ to $m4$. The corresponding ECC graph is shown in Fig. 2(b). The solid lines are continuation edges and the dotted lines are conflict edges. We assume that wire segments 7 and 12 are too close to be put in the same layer. Therefore, there are conflict edges $e_{7,b}$, $e_{7,12}$, and $e_{f,12}$. Note that we did not put an edge between vertices 7 and 12, as it would be a redundant conflict edge and can be eliminated because of the existence of the continuation edge between via vertex b and

segment vertex 12. If vertex b does not have a conflict layer assignment with vertex 7, nor does vertex 12. We also assume that via vertex c and d are too close to be put in the same layer, e.g., if c connects layers $m1$ and $m2$, and d connects layers $m2$ and $m3$ at the same time, there will not be enough routing space on layer $m2$. In this situation, we need a conflict edge $e_{c,d}$. For the case that they may share some layers but not all layers, we will still introduce a conflict edge but use the cost function to distinguish the different cases (see the details of cost functions below).

In our algorithm, we use cost functions to represent the concepts of conflict and continuation. We shall first describe how to use the cost functions for the via minimization problem in this section. It is also possible to define suitable cost functions to solve the layer assignment problems for crosstalk and delay minimization, which will be explained in Section VI.

We define the cost functions for the via minimization problem as follows: For each edge $e_{u,v}$, we associate a cost matrix $M_{u,v}$; an entry $M_{u,v}[m,n]$ gives the cost for assigning u to layer m and v to layer n at the same time. The cost matrix $M_{u,v}$ will capture the penalty on the conflicting layer assignment between vertices u and v. If there is a conflict for the assignment, the entry will be ∞, otherwise, it will be zero. The conflicts may be characterized by the spacing rule violation, overlapping of wires of different nets, or disconnect of wires or vias of the same net which should be connected. This scheme is very general; for the cases when two vertices conflict only on some layers, we set the matrix entries corresponding to the conflict assignments to ∞, and the remaining entries to zero.

For each vertex v, we also associate a cost array A_v of size $l(v)$; an element $A_v[m]$ of the array specifies the costs of assigning vertex v to layer m. Clearly the cost array for a via can be used to specify whether stacked vias are allowed or not, how many layers can be stacked, etc. For example, if no stacked vias are allowed, for each via vertex v, $A_v[(l_l, l_u)] = 0$ if $l_l = l_u$, one if $l_u - l_l = 1$, or ∞ if $l_u - l_l > 1$. We can also allow stacked vias up to m layers and count each stacked via spanning s layers with cost $s-1$, by assigning cost for $A_v[(l_l, l_u)] = l_u - l_l$ if $l_u - l_l \leq m$, or ∞ if $l_u - l_l > m$.

We may also replace the cost functions with more complex ones. For example, we can put layer constraints on a certain net or part of it such that it cannot be assigned into these forbidden layers. Our representation can also handle via minimization for gridless routing where wire segments have variable widths and spacing requirements on the same or different layers. We simply model different possible conflicts on each layer using the cost matrices derived from the design rules.

Please note that the formulation of ECC graph is very general and can handle multilayer gridless layout in advanced IC and MCM designs. It accommodates different widths of wires and vias in the same layer or different layers as well as different wire-to-wire, wire-to-via and via-to-via spacing in the same layer or different layers. It also handles stacked vias when allowed by technology. Multiway splitting wires are also handled elegantly by this formulation.

We define the cost for a layer assignment σ which assigns each vertex v into layer $\sigma(v), (1 \leq \sigma(v) \leq l(v))$ to be the summation of all the edge costs and via costs, which is computed by

$$\mathit{COST}(\sigma) = \sum_{v \in V} A_v[\sigma(v)] + \sum_{e_{u,v} \in E} M_{u,v}[\sigma(u), \sigma(v)].$$

Our goal for the layer assignment problem is to find a layer assignment σ^* with the minimum cost, i.e., $\mathit{COST}(\sigma^*) \leq \mathit{COST}(\sigma)$ for any σ.

III. Optimal Algorithm for Trees

Although all the vertex and edge costs are defined for the layer assignment of adjacent vertices, our objective is to minimize the summation over all the costs of vertices and edges. It is possible that greedy local changes of layer assignments may restrict the assignments of remote vertices. A simple-minded local refinement algorithm, which does not have a global view of the problem, may produce poor results. We would like to find an efficient algorithm which can take advantage of the locality of the cost computation and also provide an efficient way to compute the impacts on the total cost when a local optimization of layer assignment is made.

Because the multilayer layer assignment is NP-hard, we do not expect to find an algorithm which can solve the general problem optimally. A special case, however, can be solved optimally: if an ECC graph is a tree, we can employee the dynamic programming paradigm to form an algorithm which optimally solves the multilayer layer assignment problem in linear time. Furthermore, we can apply this algorithm to a large portion of the entire ECC graph with a slight modification of the algorithm. This enables us to form an efficient heuristic algorithm which can refine a large portion of the layout optimally, thus, providing a more global view on the optimization problem and having a better chance to avoid bad local decisions and obtain better solutions.

Before we propose the optimal layer assignment algorithm for trees, we need to define some notations. For a vertex u in the tree, we define $\mathit{ch}(u)$ to be the set of the child vertices of u. For a layer assignment σ on a tree, we define the cost for a layer assignment σ on a subtree rooted at u as $\mathit{COST}_u(\sigma) = A_u[\sigma(u)] + \sum_{v \in ch(u)} (M_{u,v}[\sigma(u), \sigma(v)] + \mathit{COST}_v(\sigma))$. This cost contains three parts: the layer assignment cost for u, the edge costs of edges between u and its children, and the costs of subtrees rooted at the children of u. When u is the root of the tree, $\mathit{COST}_u(\sigma)$ computed by the above definition is the same as $\mathit{COST}(\sigma)$ defined in the previous section. We define $\mathit{MC}_u[i] = \min_\sigma \{\mathit{COST}_u(\sigma) \mid \sigma(u) = i\}$; i.e., it gives the minimum cost for the subtree rooted at u under the condition that u is assigned to layer i. Note that $\mathit{MC}_v[i]$ for a leaf vertex v is simply $A_v[i]$. For a nonleaf vertex u, it can be computed as follows:

$$\begin{aligned} \mathit{MC}_u[i] &= \min_\sigma \{\mathit{COST}_u(\sigma) \mid \sigma(u) = i\} \quad (1) \\ &= \min_\sigma \{A_u[i] + \sum_{v \in ch(u)} (M_{u,v}[i, \sigma(v)] \\ &\qquad + \mathit{COST}_v(\sigma)) \mid \sigma(u) = i\} \quad (2) \end{aligned}$$

ALGORITHM: K-LAT(K-layer Layer Assignment for Trees)
INPUT: An extended conflict-continuation graph G = (V, E) which is a tree. A cost matrix $M_{u,v}$ for each edge $e_{u,v}$. A cost array A_u for each vertex u.
OUTPUT: A minimum cost layer assignment.

1. For each vertex u in the postorder traverse of the tree
   For each layer $1 \le i \le l(u)$
   $MC_u[i] = A_u[i] +$
   $\sum_{v \in ch(u)} (min_{1 \le j \le l(v)} \{M_{u,v}[i,j] + MC_v[j]\})$
   For each $v \in ch(u)$,
   $m_v[i] = j^*$,
   where j^* is the layer which v gives the minimum value for $MC_u[i]$.
2. For root r of the tree, find g^* which gives the minimum cost among $MC_r[g]$'s $(1 \le g \le l(r))$, assign $\sigma(r)$ with g^*.
3. For each vertex v on the preorder traverse of the tree,
   $\sigma(v) = m_v[\sigma(u)]$,
   where u is the parent of v.
4. Return σ.

Fig. 3. K-layer layer assignment for trees (K-LAT) algorithm.

$$= A_u[i] + \sum_{v \in ch(u)} \min_{\sigma} \{(M_{u,v}[i, \sigma(v)] + COST_v(\sigma)) \mid \sigma(u) = i\} \quad (3)$$

$$= A_u[i] + \sum_{v \in ch(u)} \min_{j} \left\{ \min_{\sigma} \{(M_{u,v}[i,j] + COST_v(\sigma)) \mid \sigma(u) = i, \sigma(v) = j\} \right\} \quad (4)$$

$$= A_u[i] + \sum_{v \in ch(u)} \left(\min_{j} \{M_{u,v}[i,j] + MC_v[j]\} \right). \quad (5)$$

The deduction from (1) to (2) is based on definitions of $COST_u(\sigma)$. In (2), because the layer assignment of u is fixed, the layer assignments for its children are independent of each other, and can be optimized independently. Therefore, we can transform the minimization of the summation over several independent terms into a summation over the minimization of each term, which leads to (3). In (4), we itemize the minimization over all possible values of j. After pulling out the constant term $M_{u,v}[i,j]$ and applying the definition of $MC_v[j]$, we get (5).

The above equations show how to compute the MC_u recursively from the layer cost array A_u, the edge cost matrix $M_{u,v}$ and the MC_v of each child v of u. Clearly, to compute the $MC_u[i]$, we only need to minimize the summation of the cost of a subtree rooted at v and the edge cost between u and v for each child v of u.

Based on this formula, our algorithm consists a bottom-up cost computation and a top-down layer assignment. For the bottom-up cost computation, we need to make sure that when MC_u is being computed, for each child v of u, MC_v has been computed. To satisfy this condition, we choose to use the postorder traversal of the tree (other orders are also possible).

During the bottom-up cost computation, we also use an auxiliary array m_v on each vertex v to store the optimal layer assignments for all the possible layer assignments of the parent of v. When a minimum cost $MC_u[i]$ of vertex u is found, we set $m_v[i]$ to j^*, where j^* is the corresponding layer number of v which gives $MC_u[i]$ the minimum cost.

After the cost array MC_r is computed for the root r, we know the minimum cost for the whole tree and the best layer assignment of r by checking the minimum cost in MC_r. For a nonroot vertex v, when its parent's optimal layer assignment is known, we know to what layer vertex v should be assigned in order to give the minimum cost because we have recorded this information in the m_v array. Clearly, we can compute the optimal layer assignment of each vertex in the tree by a top-down traversal which simply assigns layers $m_v[i]$ to vertex v if the parent is known to be assigned to layer i.

We summarize our optimal layer assignment algorithm in Fig. 3.

Theorem 1: Given an ECC graph $G = (V, E)$ which is a tree, the K-LAT algorithm finds an optimal K-layer assignment in $O(|V|)$.

Proof: The optimality follows by induction. First, it is trivial to see that the subtree costs at the leaves are minimum. For the induction part, we can see that our algorithm computes MC_u by (5), which only requires to know MC_v for $v \in ch(u)$; since the cost array MC_v for each child v of u is optimally computed by the induction hypothesis, we know that MC_u is also optimally computed. Therefore, we can conclude that the MC_{root} is optimally computed by the K-LAT algorithm. Because the top-down layer assignment will realize the layer assignment for the optimal cost, it gives us an optimal layer assignment of the tree.

For the complexity of this algorithm, we first discuss the number of possible layers $l(v)$ per vertex v. For a wire segment vertex, $l(v) = K$. For a via vertex v, the number $l(v)$ depends on how many layers the via can be stacked. In the most restricted case, vias can connect only two adjacent layers, $l(v) = 2K - 1$. For the least restricted case, there can be $K(K+1)/2$ possible layer assignments for the via vertex.

For Step 1, each edge $e_{u,v}$ requires examining $l(u)l(v)$ edge costs for the computation of MC_u. Since the number of edges in the tree is $|V| - 1$, the complexity of Step 1 is $O(K^2|V|)$ for the most restricted case and $O(K^4|V|)$ for the least restricted case. Step 2 takes $O(K)$ for the most restricted case and $O(K^2)$ for the least restricted case, and Step 3 takes $O(|V|)$ time. So, this algorithm runs in $O(K^2|V|)$ for the most restricted case and $O(K^4|V|)$ for the least restricted case. Since K is an integer constant in our algorithm, we have $O(|V|)$ linear time algorithm for optimal layer assignment for trees. In practices, $K = 2$ to 5 for modern IC's, and is usually less than ten for most PCB/MCM designs. □

IV. An Efficient Heuristic for General Graphs

In general, the formulated ECC graph of a layer assignment problem may not be a tree. In this section, we shall first show that the K-LAT algorithm is also applicable to induced subtrees in a general ECC graph. Then we shall use it as the basis for our heuristic algorithm to optimize a sequence of induced subtrees in a general ECC graph. Let us first introduce the concept of induced subtrees.

Definition 1: A subgraph $G' = (V', E')$ is a *(vertex) induced subgraph* of $G = (V, E)$ if, and only if, $V' \subseteq V$ and $E' = \{e_{u,v} \mid u \in V', v \in V', \text{ and } e_{u,v} \in E\}$. When an induced subgraph of G is a tree, it is called an induced subtree.

Definition 2: Given an ECC graph G and a layer assignment σ, the layer assignment problem for an induced subgraph G' is to find a layer assignment σ'^* s.t. $\forall v \in V - V', \sigma'^*(v) = \sigma(v)$, and $COST(\sigma'^*)$ is minimum.

If we choose a subset S of the vertices in the ECC graph G and fix the layer assignments for the vertices outside of S, the original layer assignment problem is reduced to the layer assignment for S. Since we need to consider all the edges among vertices in S which are originally in G, we need to consider the subgraph induced by S. To utilize our optimal algorithm K-LAT, we are interested in an induced subgraph which is a tree.

A. Extension of the K-LAT Algorithm to Induced Subtrees

The extension of the optimal K-LAT algorithm for trees to induced subtrees is quite simple. For a vertex v in an induced subtree, we only need to increase the layer costs of v by the edge costs between v and its adjacent nontree vertices, so that the layer costs for each tree vertex will also include the costs due to its relations with its nontree neighbors. After such modification of the vertex cost arrays, we can apply the K-LAT algorithm directly. Our optimal algorithm for induced subtrees can be summarized in Fig. 4.

ALGORITHM: K-LATI(K-LAT for Induced subtrees)
INPUT: An ECC graph $G = (V, E)$, an induced subtree $T = (V', E')$, a feasible layer assignment σ, Cost matrices $M_{u,v}$ for each edge $e_{u,v}$, and cost array A_i for each vertex i.
OUTPUT: A minimum cost layer assignment for the induced subtree T.

1. For each $v \in V'$, $1 \le i \le l(v)$,
   $B_v[i] = A_v[i] + \sum_{x \in V - V', e_{v,x} \in CUT(V', V - V')} M_{v,x}[i, \sigma(x)]$.
2. Invoke K-LAT on T with the cost array A_v replaced by B_v for each vertex v.

Fig. 4. K-LATI algorithm.

Because K-LAT is optimal, K-layer layer assignment for induced trees (K-LATI) is also optimal. We have the following corollary:

Corollary 1: Given an ECC graph G, a feasible assignment σ, and an induced subtree $T = (V', E')$ of G, the K-LATI algorithm finds a minimum cost layer assignment of T in $O(|V'| + |CUT(V', V - V')|)$ time, where the $CUT(V', V - V')$ is the set of edges connecting V' and $V - V'$.

Proof: The optimality follows because we have introduced the costs of the edges to the nontree vertices in the cost array for each tree vertex, and because K-LAT is optimal. The time complexity of K-LATI is derived as follows. Step 1 takes $O(K|CUT(V', V - V')|)$ for the most restricted case on stacked vias and $O(K^2|CUT(V', V - V')|)$ for the least restricted case, as we only need to visit every edge in $CUT(V', V - V')$ once and each visit requires $l(v)$ operations. Step 2 takes $O(|V'|)$ time as shown in Theorem 1. Therefore, K-LATI runs in $O(|V'| + |CUT(V', V - V')|)$ time. □

To illustrate the K-LATI algorithm, we use the example in Fig. 5(f), which shows the ECC graph of the layout shown in Fig. 2(a). The grey nodes form an induced subtree. We assume the number of layers is four and stacked vias are allowed. Suppose vertex 1 is the root, then the set $\{2, 3, 4, 5, 8, 10, 11\}$ are the leaf vertices with no children. Let us focus on a small subtree rooted at vertex 9. It has two children, vertex 11 and vertex e. Vertex e has two children, vertex 4 and vertex 10. Vertex 9 has a nontree neighbor 12 in the graph and vertex 11 has a nontree neighbor d in the graph. So the layer cost array for vertex 9 is $B_9[] = \{0, 0, \infty, 0\}$ after we add the costs of the edges to nontree vertices, because vertex 12 is in layer 3 and conflicts with vertex 9. Similarly, the layer cost array for vertex 11 is $B_{11}[] = \{\infty, \infty, 0, 0\}$, because vertex d is in layer $(3, 4)$ and is connected to vertex 11 by a continuation edge. Now we begin the bottom-up cost computation. Since vertices 4, 10, 11 are leaf vertices, their subtree costs are equal to their layer cost arrays. So $MC_4[] = \{0, 0, 0, 0\}$, $MC_{10}[] = \{0, 0, 0, 0\}$, and $MC_{11}[] = \{\infty, \infty, 0, 0\}$. Now we compute the cost for vertex e. Assume that the cost for a stacked via is the height it spans, and recall that we encode the layer pair for a via vertex in the following sequence: $(1,1), (2,2), (3,3), (4,4), (1,2), (2,3), (3,4), (1,3), (2,4), (1,4)$. The layer cost array for via vertex e is $B_e[] = \{0, 0, 0, 0, 1, 1, 1, 2, 2, 3\}$. Because we can always assign the layers for vertices 4 and 10 to be the same as e and the costs for subtrees rooted at 4 and 10 are all zero, the $MC_e[]$ is the same as $B_e[]$, $MC_e[] = \{0, 0, 0, 0, 1, 1, 1, 2, 2, 3\}$. The corresponding assignment arrays for vertices 4 and

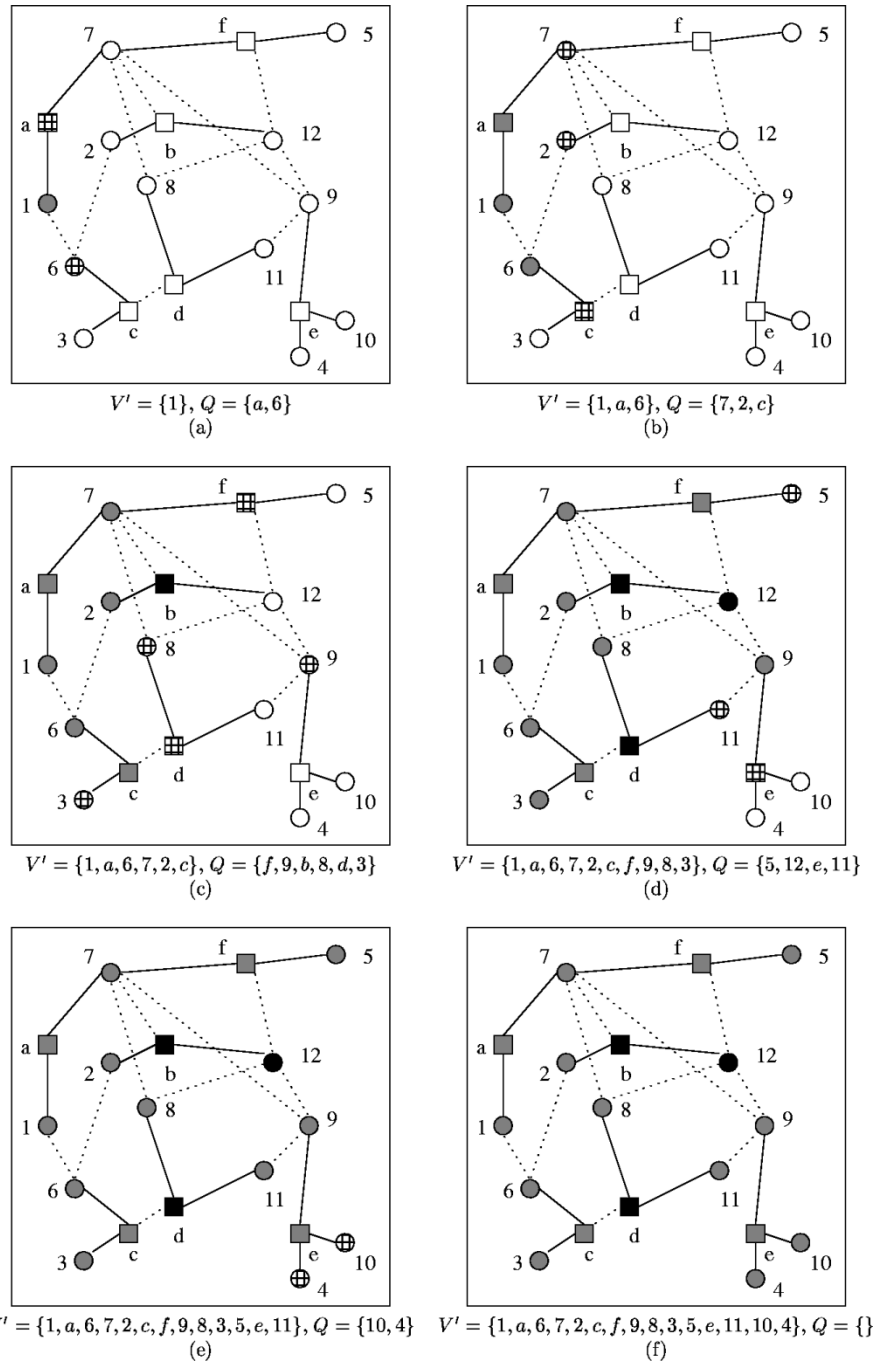

Fig. 5. Example of finding a maximal induced subtree $T = (V', E')$.

10 are: $m_4[\,] = \{1,2,3,4,1,2,3,1,2,1\}$ and $m_{10}[\,] = \{1,2,3,4,1,2,3,1,2,1\}$. Note that $m_4[5]$ can be either 1 or 2 as either layers 1 or 2 for vertex 4 gives the same cost for the subtree rooted at e. When there are several layer assignments for a vertex with the same minimum cost, we simply pick the one with a smaller layer number. This is why $m_4[5] = 1$. Now we can compute the cost of the subtree rooted at vertex 9. If we assign vertex 9 to layer 1, the minimum sum of $MC_{11}[j]$ and $M_{9,11}[1, j]$ is zero with $j = 3$ or $j = 4$. So we have $m_{11}[1] = 3$. The minimum sum of $MC_e[j]$ and $M_{9,e}[1, j]$ is zero with $m_e[1] = 1$. We add these two minimum costs to $B_9[1]$ and get $MC_9[1] = 0$. The computation of the minimum cost of assigning vertex 9 to other layers is similar. Eventually, we will get $MC_9[\,] = \{0, 0, \infty, 0\}$, $m_{11}[\,] = \{3,3,4,3\}$, and $m_e[\,] = \{1,2,3,4\}$.

Given the above optimal layer assignment algorithm K-LATI, our algorithm finds a sequence of induced subtrees and applies K-LATI algorithm to them one by one. In order to take advantage of the K-LATI algorithm, we need to find a sequence of large induced subtrees.

B. Finding Maximal Induced Subtrees

In general, we are interested in forming large induced subtrees. However, finding an induced subtree of the maximum size is NP-hard [18]. In addition, we wish to find a *set* of large

ALGORITHM: FMIST(Finding Maximal Induced SubTree)
INPUT: An ECC graph $G = (V, E)$ and a vertex r.
OUTPUT: A maximal induced subtree $T = (V', E')$ rooted at r.

1. $V' = \emptyset$. $Q = \emptyset$. For each v, $label(v) = 0$.
2. $label(r) = 1$, $parent(r) = r$, push r into queue Q.
3. While Q is not empty
   Remove a vertex v from Q.
   If $label(v) = 1$
      Add v into the induced subtree vertex set V'.
      For each u, s.t. $e_{u,v} \in E$,
         increase $label(u)$ by one.
         If $label(u) = 1$, then
            $parent(u) = v$, push u to the queue Q.

Fig. 6. FMIST algorithm.

induced subtrees to cover the graph, rather than a single large tree. For our purpose, we find that growing maximal induced subtrees (i.e., induced subtrees which are not contained in any induced subtrees) from different starting vertices is effective. On average, the maximal induced subtrees cover 12%–34% of the ECC graph.

Our induced-subtree-finding algorithm starts with an arbitrary vertex v and finds a maximal induced subtree $T = (V', E')$ rooted at v. It is basically a graph traversal algorithm with proper labeling. Each vertex v in the graph is initially labeled with $label(v) = 0$. We maintain a queue Q for the vertices which are adjacent to some vertex in the induced subtree T under construction. Initially, Q is empty. We choose a vertex r as the root, label it with one, and add it to V'. Once a vertex is added to V', we increase the label of all its neighbors by one. Once a vertex has label 1 after the label increase, we put it into Q. Then we extract the vertices from Q one by one. If the extracted vertex v has a label larger than one, we know that v is a neighbor for two or more vertices in V'. In this case, vertex v is discarded because if we add it to V', we would create cycle(s) in T. If $label(v) = 1$, we know v has exactly one connection to T. In this case, we add v to V', increase the labels of v's neighbors by one, and insert any of its neighbors with label 1 into Q. We repeat this process to extract another vertex from Q again, and stop when Q is empty.

We summarize our algorithm for finding maximal induced subtrees in Fig. 6.

Fig. 5 shows how this marking process in the finding maximal induced subtrees (FMIST) algorithm works. The grey nodes are the vertices we put in the induced subtree, the black nodes are the nontree nodes with labels larger than one, and the hash nodes are the nodes currently labeled as one. Note that in Fig. 5(c), when both the vertices 7 and 2 are added to V', the label of vertex b increases twice and becomes two, which makes it a nontree vertex. Similarly, d and 12 become nontree vertices when eight is added to V'.

Theorem 2: Given an ECC graph $G = (V, E)$, the FMIST algorithm computes a maximal induced subtree $T = (V', E')$ in $O(|V'| + |CUT(V', V - V')|)$ time.

Proof: We prove the correctness by induction. It is trivial that a single vertex in the graph G is an induced subtree. Assume $T' = (V'', E'')$ is an induced subtree. When we add a vertex v to T', the labeling process guarantees that there is exactly one vertex in T' with an edge to v. Therefore, the induced subgraph generated by $V'' \cup \{v\}$ is both connected and without cycles, thus, it is still a tree. When the algorithm stops, we cannot add in any other vertex in the graph without forming a cycle. Therefore, the subtree found is maximal. For the time complexity, we need to traverse each tree edge twice and the edge in the cut set $CUT(V', V - V')$ once. The run time is $O(|V'| + |CUT(V', V - V')|)$. □

C. An Efficient Algorithm for Layer Assignment

Now we are ready to present our heuristic algorithm for the general layer assignment problem. Given a feasible layer assignment solution σ with the ECC graph, we refine σ to minimize the layer assignment cost through several passes. In each pass, we cover G with a set of maximal induced subtrees computed by the FMIST algorithm, and apply the K-LATI algorithm to get the optimal layer assignment for these maximal induced subtrees one by one. If the cost reduction in the current pass is larger than a specified stopping threshold t, we start another pass. Otherwise, we stop the program. The algorithm is summarized in Fig. 7.

Please note that each vertex will appear in at least one induced subtree in each pass. Although we start from an unmarked vertex to grow an induced subtree at each time, the induced subtree might also contain a vertex which was part of another induced subtree. The run time of the K-LAG is $O(P \times \sum_{V' \in VP}(|V'| + |CUT(V', V - V')|))$, where P is number of passes and VP is the set of subtrees found in each pass. In practice, we observed that P is usually a small number less than ten.

In our implementation, our scheduler randomly selects an unmarked vertex as the root of an induced subtree. Other heuristics may also be used.

V. Experimental Results

The experiments are conducted on a SUN SPARC ULTRA-2(168 MHz) work station with 256 megabytes of memory. We tested our algorithms with the following test cases: *test*1, *test*2, *test*3, *mcc*1, *mcc*2, *fract*, and *struct*. The routing results of *test*1, *test*2, *test*3, *mcc*1 and *mcc*2 are generated by the V4R router [1]. Test cases *test*1, *test*2, *test*3, and *mcc*1 are routed in four layers; *mcc*2 is in six layers. Test cases *mcc*1 and *mcc*2 are industrial MCM designs provided by MCC [also available at the Collaborative Benchmarking Laboratory (CBL)]. In particular, *mcc*2 is a supercomputer with 37 IC chips. For further information on these V4R test examples, please refer to [1]. The test cases *fract* and *struct* are standard cell designs also available at CBL. The layouts of *fract* and *struct* are gridless layouts routed by an industrial router using three routing layers. They are routed with 0.6-, 0.6-, and 1.2-um minimum widths and 0.8-, 0.8-, and 1.2-um minimum spacings on METAL1, METAL2, and METAL3 respectively. The routing area for *fract* is 181.4 um × 183.3 um and the area for *struct* is 638.3 um × 599.6 um.

It is forbidden to drop vias on the terminal locations in *fract* and *struct*. Stacked vias are not allowed except for the distribution vias in the V4R test cases, in which they are used to bring the terminals to their proper routing layers.

ALGORITHM: K-LAG(K-layer Layer Assignment for ECC Graph)
INPUT: An ECC graph $G = (V, E)$, a feasible layer assignment σ, a cost matrix $M_{u,v}$ for each edge $e_{u,v}$, a cost array A_v for each vertex v, and a stopping threshold t.
OUTPUT: A refined layer assignment σ^* with minimal cost.

1. Use a scheduler to get a vertex r.
2. Run FMIST algorithm with r to get a maximal induced subtree T rooted at r and mark all vertices in T.
3. Apply the K-LATI algorithm to T.
4. If there exists unmarked vertices, goto Step 1.
5. If improvement larger than t, unmark all vertices, goto Step 1.
6. Output the refined layer assignment σ^*.

Fig. 7. K-layer layer assignment for ECC graph (K-LAG) algorithm.

TABLE I
EXPERIMENTAL RESULTS OF K-LAG ALGORITHM (WITH DEFAULT PARAMETER SETTING)

examples	# of layers	# of vias	# of vertices	# of edges	# of subtrees	avg. tree size(%)	via reduction(%)	CPU time (sec)
fract	3	428	2843	10603	17.3	975.2(34.3)	2.8(0.7)	1.1
struct	3	5229	39376	154783	136.2	5372.8(13.6)	138.7(2.7)	63.3
test1	4	1965	4328	38740	111.4	1383.7(32.0)	257.0(13.1)	14.8
test2	4	4888	8606	139047	195.2	2407.3(28.0)	694.6(14.2)	61.8
test3	4	6298	11204	217152	172.8	3146.1(28.1)	791.8(12.6)	80.2
mcc1	4	5733	11649	152104	378.5	1357.3(11.7)	827.9(14.4)	59.2
mcc2	6	40267	67555	2573594	325.8	14560.9(21.6)	6064.9(15.1)	1459.9

A. Impacts of MultiLayer Via Minimization

Table I shows the experimental results of the test cases by K-LAG algorithm. These experiments are done with the default parameter settings which will be explained later.

Each entry in the table is the average of ten experiments of the same data and parameters. The nondeterministic nature of the program comes from random selection of starting vertices for growing induced subtrees and random tie-breaking for choosing vertices to be added to the induced subtree. Although we could easily make the program deterministic, we found that randomization usually improves run time and solution quality.

Some columns in Table I may need further explanations. The third column shows the number of vias in the original layout before we apply the K-LAG algorithm. For the V4R test cases, we count all the vias on the routing layers.[3] For the test cases *fract* and *struct*, which are standard cell designs, we count all the vias of the interconnects among cells, but we do not count vias inside any cell. The fourth column shows the total number of vertices in the ECC graph for each test case. The fifth column shows the total number of edges in the ECC graph. The sixth column shows the average total number of maximal induced subtrees constructed in each run of K-LAG algorithm. The seventh column shows the average number of vertices in a maximal induced subtree; the average is taken on all the subtrees generated in all runs. In the same column, the numbers in the parentheses are the ratio of the average number of vertices in the maximal induced subtrees to the total number of vertices in the ECC graph. The eighth column shows the number of vias reduced and the ratio of via reduction to the total number of vias.

The average size of maximal induced subtrees shown in Table I is from 12% to 34% of the total number of vertices. This indicates that our algorithm can optimally refine the layer assignment of a large portion of the entire graph on each run of K-LATI algorithm, which leads to highly optimized layer assignment solutions.

The results from *fract* and *struct* are modest compared to the results from the V4R test cases. *Fract* and *struct* are routed in three layers using an industrial router, and no vias are allowed in terminal locations; this leaves little opportunity for improvement through layer assignment. This industrial router has also been integrated with some post-layout optimization procedures for via minimization. The via reduction in the V4R test cases are from 12.6% to 15.1%. This is a significant reduction since V4R is a router known to have low usage of vias. The V4R test cases are routed in four or six layers; vias are also allowed in those terminal locations. As a result, we have considerable freedom to do the via minimization in these test cases, and obtain much better via reductions.

B. Algorithm Options

Our K-LAG algorithm for multilayer gridless layouts has several options. We shall explain these options here. Their impacts to solution quality and run time will be evaluated in Sections V-C to V-E.

1) Wire Segmentation: We assume the input layouts of our program are given as collections of wire segments and

[3]The via numbers shown here are different from those reported in [1]. In V4R, the distribution vias can be stacked to bring the input–output pins on the surface to any routing layer. Each distribution via is counted as one, no matter whether it is stacked or not. Also, due to an oversight, the distribution vias in a multiterminal net are over-counted [19] (as each multiterminal net was broken as a set of independent two-terminal-nets). In our via counting routing, we count each distribution via by the number of layers it spans excluding the surface layer. Moreover, we corrected the via over-counting problem for multiterminal nets. The same via counting procedure is used to report the via numbers in all the tables.

TABLE II
EFFECTS ON WIRE SEGMENTATION

examples	original # of vias	no segmentation			medium segmentation			maximum segmentation		
		# of vertices	# of vias reduced	run time (sec)	# of vertices	# of vias reduced	run time (sec)	# of vertices	# of vias reduced	run time (sec)
test1	1965	4328	259.2	19.7	13914	263.2	86.5	52254	270.0	460.6
test2	4888	8606	695.0	94.3	29180	702.0	252.1	111470	726.6	2080.9
test3	6298	11204	793.6	175.9	35136	802.0	538.3	178718	822.2	5138.9
mcc1	5733	11649	831.8	120.7	36827	832.8	441.3	162693	852.3	2835.8
mcc2	40267	67555	6118.0	4876.2	288087	6172.6	12909.7	472395	6201.9	29143.9

vias. Our program has an option to further break each long wire segment into a set of smaller wire segments and via candidates. The breaking points are chosen from the middle points between locations where this wire segment crosses other wire segments. Furthermore, only those middle points of segments which are longer than a certain length m will be considered as breaking points. The length m is usually determined by the minimum space required by design rule for a via insertion. However, we can specify some larger number to reduce the number of breaking points.

Let us consider the wire segment $(a2, a8)$ in Fig. 1 to illustrate how to do the wire segmentation. The wire segment $(a2, a8)$ crosses other wire segments at points $a2$, $a3$, $a4$, $a5$, $a7$, and $a8$. Assume the lengths of wire segments $(a2, a3)$, $(a3, a4)$, $(a4, a5)$, and $(a7, a8)$ are all smaller than m, thus, they will not be broken. Because the length of wire segment $(a5, a7)$ is larger than m, we have a breaking point $a6$, which is the middle point of $(a5, a7)$. In this case, $(a2, a8)$ is broken to $(a2, a6)$ and $(a6, a8)$.

Our program can also restrict the number of breaking points b to be less or equal to γn by iterating through all B feasible breaking points and selecting only one out of every $B/\gamma n$ of them, where γ is an user-specified number and n is the number of vertices of the ECC graph. This gives us some controls on the degree of wire segmentation.

2) Induced Subtree Growing Strategies: During the running of FMIST algorithm to grow a maximal induced subtree, there are usually more than one vertex which can be added to the induced subtree at the same time. Choosing different vertices may end up with different maximal induced subtrees. Since different maximal induced subtrees may give different via reductions, we experimented with different strategies to grow maximal induced subtrees.

Recall the procedure FMIST, which grows a maximal induced subtree (described in Section IV-B). It uses a queue to store the vertices which are adjacent to the induced subtree. It repeatedly extracts a vertex from the queue to determine if it can be added to the induced subtree. When inserting a vertex u to the induced subtree, it increases the labels of all the neighbors of u by one. If there are some neighbors of u with label 1 after the increments, it inserts them to the queue. Because we extract and check the vertices of the queue in a first-in-first-out order, the order of insertions determines the order of vertices being added to the induced subtree. We have the following two strategies to decide the order of insertions.

Strategy A: When adding vertex u to the induced subtree, we first insert neighbor vertices which are connected to u by continuation edges, then the vertices connected by conflict edges.

Strategy B: We first separate the neighbor vertices that need to be added into the queue into two groups. The first group consists of the vertices which have not been included in any maximal induced tree in the current pass. The second group consists of the rest. The vertices in the first group will be added before the second group. We then use the ordering in Strategy A to determine the ordering in both groups.

The reason that we want to use Strategy A to grow the induced subtree is that we wish to give some preference to the continuation edges because via reduction can only happen in reassigning vertices connected by the edges. The reason why we have Strategy B is that we want to avoid growing subtrees which are similar (having a lots of vertices and edges in common) to other induced subtrees generated before.

In both strategies, there may be vertices with the same priority. Our program randomly break ties, and this explains why multiple runs of our program may give different results.

If wire segmentation is applied, we will grow the induced subtree using the original wire segments, and then break wires according to the wire segmentation.

3) Stopping Thresholds: Our program usually obtains most of the improvements from the first few maximal induced subtrees. The reduction in later passes also tends to be small. Our program has an option to stop earlier when it sees the improvement in one pass is less than an adjustable fraction of the total number of vias (the default value is set to 0.5%). We call this option "quick stopping."

C. Impacts of Wire Segmentation

Table II shows the effects on wire segmentation.[4] The data are obtained by using tree growing Strategy B and no quick stopping. All the numbers in this table are the average of ten experiments of the same setting.

We have three sets of experiments on the same data: no segmentation, medium segmentation, and maximum segmentation. In the maximum segmentation, we use all the possible breaking points defined in the subsection V-B.1 to break the wires (except for *mcc*2, which would require more than 300 megabytes of memory to do so. To limit the number of breaking points in *mcc*2, we have doubled the minimum length required to break a wire segment.) In the medium segmentation, we limit the number of breaking points to roughly the same as the original number of vertices in the ECC

[4]We do not include test cases *fract* and *struct* in Table I because there are not many long wire segments to be broken in these two test cases.

TABLE III
EFFECTS ON DIFFERENT SUBTREE GROWING STRATEGIES (WITH QUICK STOPPING)

example	strategy	# of passes	# of subtrees	# of subtrees of size $\geq 20\%$	avg. subtree size (%)	via reduction (%)	avg. # of subtrees per pass	time (sec)
fract	A	1.8	85.1	72.9	992.4(34.9)	3.6(0.8)	47.3	5.5
	B	1.4	17.3	14.7	975.2(34.3)	2.8(0.7)	12.4	1.1
struct	A	2.0	1414.8	916.7	9219.8(23.4)	152.9(2.9)	707.4	1096.6
	B	2.0	136.2	51.4	5372.8(13.6)	138.7(2.7)	68.1	63.3
test1	A	2.1	120.8	119.5	1387.6(32.1)	258.6(13.2)	57.5	16.0
	B	2.0	111.4	111.3	1383.7(32.0)	257.0(13.1)	55.7	14.8
test2	A	2.0	188.3	183.9	2407.2(28.0)	695.2(14.2)	94.2	59.5
	B	2.0	195.2	190.9	2407.3(28.0)	694.6(14.2)	97.6	61.8
test3	A	2.0	244.4	244.4	3190.0(28.5)	793.2(12.6)	122.2	115.2
	B	2.0	172.8	172.8	3146.1(28.1)	791.8(12.6)	86.4	80.2
mcc1	A	2.0	386.5	115.9	1229.8(10.6)	829.5(14.5)	193.2	54.7
	B	2.0	378.5	124.8	1357.3(11.7)	827.9(14.4)	189.2	59.2
mcc2	A	2.0	309.9	268.4	14624.9(21.6)	6085.1(15.1)	154.9	1398.6
	B	2.0	325.8	283.1	14560.9(21.6)	6064.9(15.1)	162. 9	1459.9

TABLE IV
EFFECTS ON DIFFERENT SUBTREE GROWING STRATEGIES (WITHOUT QUICK STOPPING)

example	strategy	# of passes	# of subtrees	# of subtrees of size $\geq 20\%$	avg. subtree size (%)	via reduction (%)	avg. # of subtrees per pass	time (sec)
fract	A	2.1	100.9	86.0	987.9(34.7)	4.0(0.9)	48.0	6.5
	B	2.4	29.8	25.1	964.1(33.9)	3.0(0.7)	12.4	1.9
struct	A	4.2	2969.8	1929.0	9243.1(23.5)	155.5(3.0)	707.1	2343.8
	B	6.4	440.4	167.3	5409.1(13.7)	150.0(2.9)	68.8	207.9
test1	A	2.7	136.3	134.8	1389.3(32.1)	258.2(13.1)	50.5	17.9
	B	3.1	146.7	146.6	1393.1(32.2)	259.2(13.2)	47.3	19.7
test2	A	2.9	232.2	226.1	2404.6(27.9)	695.6(14.2)	80.1	72.8
	B	3.1	295.0	287.9	2409.6(28.0)	695.0(14.2)	95.2	94.3
test3	A	2.7	299.9	299.9	3134.5(28.0)	793.4(12.6)	111.1	139.7
	B	3.8	373.8	373.8	3175.5(28.3)	793.6(12.6)	98.4	175.9
mcc1	A	4.5	855.7	248.4	1193.3(10.2)	833.6(14.5)	190.2	117.1
	B	4.6	834.6	253.7	1249.0(10.7)	831.8(14.5)	181.4	120.7
mcc2	A	6.7	1140.0	998.1	14779.8(21.9)	6115.5(15.2)	170.1	5231.6
	B	6.9	1074.2	928.8	14569.6(21.6)	6118.0(15.2)	155.7	4876.2

TABLE V
EFFECTS ON QUICK STOPPING (WITH STRATEGY B)

	with quick stopping			without quick stopping		
example	# of passes	via reduction	run time	# of passes	via reduction	run time
fract	1.4	2.8(0.7)	1.1	2.4	3.0(0.7)	1.9
struct	2.0	138.7(2.7)	63.3	6.4	150.0(2.9)	207.9
test1	2.0	257.0(13.1)	14.8	3.1	259.2(13.2)	19.7
test2	2.0	694.6(14.2)	61.8	3.1	695.0(14.2)	94.3
test3	2.0	791.8(12.6)	80.2	3.8	793.6(12.6)	175.9
mcc1	2.0	827.9(14.4)	59.2	4.6	831.8(14.5)	120.7
mcc2	2.0	6064.9(15.1)	1459.9	6.9	6118.0(15.2)	4876.2

graph, thus, limiting the increment of the number of vertices to roughly three times the original size.

This experiment shows that doing segmentation is always helpful to get better via reduction, but at the cost of generating larger ECC graphs, which means higher memory requirements and longer run times. In most of the examples, the improvement by maximum segmentation is less than 0.7% of the total vias, while the run time increases by six to 29 times. We have observed similar results by using different combinations of tree growing strategies and stopping thresholds.

D. Impacts of Different Subtree Growing Strategies

Tables III and IV show the effects on different tree growing strategies with different stopping thresholds. No wire segmentations are used to obtain these data. The column "# of subtrees of size $\geq$20%" is to show the total number of subtrees which is larger than 20% of the total number of vertices of the ECC graph. Again, all the numbers in the tables are the average over ten experiments.

Tables III and IV show that the difference on the solution quality is not significant when applying different strategies (all within 0.2% of total number of vias). The difference on run time is case by case. In most of the test cases, the differences are within 1.5 times. However, the run time may have big differences in some test cases. In test case *struct*, the run time of Strategy A is 17 times larger than the run time Strategy B. In this case, both the number of subtrees per pass and average subtree sizes are reduced by using Strategy B. Since there is still considerable amount of "large" subtrees (subtrees larger than 20% of the ECC graph) generated in this case, we do not see much impact on the solution quality.

E. Impacts of Different Stopping Thresholds

Table V shows the data on different stopping thresholds with Strategy B and no segmentation. The results on Strategy A are quite similar, so we do not show them. If quick stopping is used, the program will stop when the via reduction in one pass

is less than 0.5% of total number of vias. Our experimental data show that the program stops roughly at two passes in every test case when quick stopping is used. If quick stopping is not used, the program may require more passes to stop. For example, on average, it requires as many as 6.9 passes in *mcc2*. The sacrifice on the solution quality by using quick stopping is almost negligible (all within 0.2% compared to the total number of vias). However, the improvement on run time may up to a factor of three.

F. Recommended Parameter Settings

In conclusion, when the run time and memory usage is concerned and a slight sacrifice on solution quality is acceptable, we recommend running our program without segmentation, but using quick stopping and using Strategy B to grow the induced subtree. In fact, this is the default parameter setting of our program (The results in Table I are obtained under this parameter setting). If we wish to maximize via reduction at any cost, maximum segmentation should be used, and both Strategy A and Strategy B should be tried. The program should be run repeatedly and the best result selected.

VI. Layer Assignment for Delays and Crosstalk Minimization

As we mentioned in the previous sections, the ECC graph is a very general model of the layer assignment problem. With proper definitions of cost functions, our layer assignment algorithm can also be used to further reduce crosstalk and delay after routing.

In fact, our algorithm will work on any cost function on an ECC graph with the following properties.

1) The costs are distributed on vertices and edges, and the total cost is the summation of weighted costs over all vertices and edges.
2) The cost of a vertex is determined only by its layer assignment.
3) The cost of an edge is determined only by the layer assignment of the two vertices it connects.

For the layer assignment problem in VLSI layout, we can always represent the penalties for conflict and discontinuity by a cost matrix for each edge. We assume that we have the cost matrix $M_{u,v}[i,j]$ as defined previously to represent such penalties. All we need is to add extra cost functions to model the problem we want to solve.

For the crosstalk minimization problem, we could add in the crosstalk penalty for wire segments pairs with crosstalk concern. Assuming that for each pair of wire segments (u,v) in which we need to consider crosstalk, there is a preallocated slack $CB_{u,v}$ on the amount of crosstalk allowed for that pair of wire segments. Let us denote $CSK_{u,v}[i,j]$ as the crosstalk between u and v when they are assigned to layer i and j respectively. We can define our cost for crosstalk as

$$CM_{u,v}(i,j) = \begin{cases} \lambda_{u,v}(CSK_{u,v}[i,j] - CB_{u,v}), & \text{if } CSK_{u,v}[i,j] - CB_{u,v} \le 0 \\ MAX, & \text{otherwise} \end{cases}$$

where $\lambda_{u,v}$ is a preassigned weighting coefficient.

This function makes any assignment with crosstalk exceeding the allocated slack to have a large penalty cost. Otherwise, the cost is just the difference between the actual crosstalk and allocated slack times the corresponding coefficient constant. The weight coefficient $\lambda_{u,v}$ allows us to put priority on different pairs according to their importance.

The cost for vertices u and v to be assigned to layers i and j is $M_{u,v}[i,j] + CM_{u,v}(i,j)$. There is no vertex cost for this problem if only crosstalk minimization is considered, and the optimization objective becomes to minimize the total cost on all the edges. If we want to optimize both vias and crosstalk, these objectives may conflict each other. However, we can include each vertex cost with a suitable weight, and minimize the weighted sum of the costs on all vertices and edges to find the tradeoffs between these objectives.

We can also apply this layer assignment formulation to delay optimization. In VLSI layout, wires routed in different layers may have different widths, spacings and RC constants, so the delay of a wire may be different when it is assigned to different layers. Assume that we have a delay bound DB_u for each wire segment u and the delay for u on layer i is $d_u(i)$. We can define cost $DA_u(i)$ as the cost of each segment vertex u as follows:

$$DA_u(i) = \begin{cases} \lambda_u(d_u(i) - DB_u), & \text{if } d_u(i) - DB_u \le 0 \\ MAX, & \text{otherwise} \end{cases}$$

where λ_u is a preassigned weighting coefficient to adjust the priority of minimization delay on vertex u.

The cost function gives the maximum value if the delay exceeds the delay bound allocated, otherwise it is the difference between the delay and the delay bound times the corresponding coefficient constant. The total cost is the sum of the costs over all vertices (*DA* arrays) and edges (M matrices).

Given these definitions of cost functions, we can apply the K-LAG algorithm to solve the multilayer layer assignment problem with optimization objectives as weighted combinations of via, crosstalk, and delay minimization.

VII. Conclusion

We have introduced the notation of an ECC graph to represent the layer assignment problem in multilayer gridless layout. We showed how to use the ECC graph to represent the layer assignment problem for via minimization, crosstalk minimization, and delay minimization.

We have presented a linear time optimal layer assignment algorithm K-LAT that solves the case when an ECC graph is a tree. After a slight modification on the K-LAT algorithm, we obtained the K-LATI algorithm which optimally solves the layer assignment problem for induced subtrees in an ECC graph. Our K-LAG algorithm is an efficient heuristic algorithm for the layer assignment for a general ECC graph. The K-LAG algorithm utilizes the K-LATI algorithm as the optimization engine and can handle very large designs efficiently, with very good solutions. Our experimental results show that the K-LAG algorithm consistently finds many large induced subtrees in the ECC graph, and achieves significant via reduction compared to the results of the V4R router, which is known to have low usage of vias.

ACKNOWLEDGMENT

The authors would like to thank R. Helzerman and P. Madden for their comments. They would also like to thank K.-Y. Khoo for providing V4R routing solutions and test cases, and assistance throughout this research.

REFERENCES

[1] K. Y. Khoo and J. Cong, "An efficient multilayer MCM router based on four-via routing," *IEEE Trans. Computer-Aided Design of Integrated Circuits Syst.*, vol. 14, pp. 1277–1290, Oct. 1995.
[2] J. Cong and D. F. Wong, "Generating more compactable channel routing solutions," *Integration: The VLSI J.*, vol. 9, no. 2, pp. 199–214, Apr. 1990.
[3] C.-P. Hsu, "Minimum-via topological routing," *IEEE Trans. Computer-Aided Design of Integrated Circuits Syst.*, vol. CAD-2, pp. 235–246, Oct. 1983.
[4] M. Marek-Sadowska, "An unconstrained topological via minimization problem for two-layer routing," *IEEE Trans. Computer-Aided Design of Integrated Circuits Syst.*, vol. CAD-3, pp. 184–190, July 1984.
[5] A. Hashimoto and J. Stevens, "Wire routing by optimizing channel assignment with large apertures," in *Proc. 8th Design Automation Workshop*, June 1971, pp. 155–169.
[6] Y. Kajitani, "On via hole minimization of routing on a 2-layer board," in *Proc. IEEE Int. Conf. Circuits and Computers ICCC'80*, Oct. 1980, pp. 295–298.
[7] M. J. Ciesielski and E. Kinnen, "An optimum layer assignment for routing in IC's and PCB's," in *Proc. ACM IEEE 18th Design Automation Conf.*, July 1981, pp. 733–737.
[8] R.-W. Chen, Y. Kajitani, and S.-P. Chan, "A graph-theoretic via minimization algorithm for two-layer printed circuit boards," *IEEE Trans. Circuits Syst*, vol. CAS-30, pp. 284–299, May 1983.
[9] R. Y. Pinter, "Optimal layer assignment for interconnect," *J. VLSI, Comput. Syst.*, vol. 1, no. 2, pp. 123–137, 1984.
[10] N. J. Naclerio, S. Masuda, and K. Nakajima, "Via minimization for gridless layouts," in *Proc. 24th ACM/IEEE Design Automation Conf.*, July 1987, pp. 159–165.
[11] K. C. Chang and D. H. Du, "Efficient algorithms for layer assignment problem," *IEEE Trans. Computer-Aided Design of Integrated Circuits Syst.*, vol. CAD-6, pp. 67–78, Jan. 1987.
[12] Y. S. Kuo, T. C. Chern, and W.-K. Shih, "Fast algorithm for optimal layer assignment," in *Proc. 25th ACM/IEEE Design Automation Conf.*, 1988, pp. 554–549.
[13] N. J. Naclerio, S. Masuda, and K. Nakajima, "The via minimization problem is NP-complete," *IEEE Trans. Comput.*, vol. 38, pp. 1604–1608, Nov. 1989.
[14] H.-A. Choi, K. Nakajima, and C. S. Rim, "Graph bipartization and via minimization," *SIAM J. Appl. Math.*, vol. 2, no. 1, pp. 38–47, Feb. 1989.
[15] K. Ahn and S. Sahni, "Constrained via minimization," *IEEE Trans. Computer-Aided Design of Integrated Circuits Syst.*, vol. 12, pp. 273–282, Feb. 1993.
[16] K. C. Chang and H. C. Du, "Layer assignment problem for three-layer routing," *IEEE Trans. Comput.*, vol. 37, pp. 625–632, May 1988.
[17] S.-C. Fang, K.-E. Chang, W.-S. Feng, and S.-J. Chen, "Constrained via minimization with practical considerations for multilayer VLSI/PCB routing problems," in *Proc. 28th ACM/IEEE Design Automation Conf.*, 1991, pp. 60–65.
[18] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: Freeman, 1979.
[19] K. Y. Khoo, private communications, 1996.
[20] C.-C. Chang and J. Cong, "An efficient approach to multi-layer layer assignment with an application to via minimization," in *Proc. 34th ACM/IEEE Design Automation Conf.*, 1997, pp. 600–603.

Chin-Chih Chang received the B.S. degree in computer science from National Taiwan University, Taiwan, in 1989 and the M.S. degree in computer science from the State University of New York at Stony Brook in 1993. Currently, he is a Ph.D. degree student in the Computer Science Department of University of California, Los Angeles.

His research interests include VLSI CAD algorithms on performance-driven layout synthesis.

Jason Cong (S'88–M'90–SM'96) for a photograph and biography, see p. 420 of the April 1999 issue of this TRANSACTIONS.