Floorplanning Using a Tree Representation

Pei-Ning Guo, Toshihiko Takahashi, Chung-Kuan Cheng, *Fellow, IEEE*, and Takeshi Yoshimura, *Member, IEEE*

***Abstract*—We present an ordered tree (O tree) structure to represent nonslicing floorplans. The O tree uses only $n(2 + \lceil \lg n \rceil)$ bits for a floorplan of n rectangular blocks. We define an admissible placement as a compacted placement in both x and y directions. For each admissible placement, we can find an O-tree representation. We show that the number of possible O-tree combinations is $O(n!2^{2n-2}/n^{1.5})$. This is very concise compared to a sequence pair representation that has $O((n!)^2)$ combinations. The approximate ratio of sequence pair and O-tree combinations is $O(n^2(n/4e)^n)$. The complexity of O tree is even smaller than a binary tree structure for slicing floorplan that has $O(n!2^{5n-3}/n^{1.5})$ combinations. Given an O tree, it takes only linear time to construct the placement and its constraint graph. We have developed a deterministic floorplanning algorithm utilizing the structure of O tree. Empirical results on MCNC (www.mcnc.org) benchmarks show promising performance with average 16% improvement in wire length and 1% less dead space over previous central processing unit (CPU) intensive cluster refinement method.**

***Index Terms*—Building block placement, floorplan, rooted ordered tree.**

I. Introduction

As THE circuit size gets larger, design hierarchy and IP blocks are intensively and increasingly used to reduce the design complexity. The floorplan or building block placement is becoming critical to the performance of hierarchical design process.

One of the key factors to most floorplanners is the representation of geometric relationship. The structure that represents the geometric relation for a floorplan will affect the basic operations to the structure and determine the inherent complexity to the approaches using it.

A. Previous Works

For a floorplan with slicing structure [3], we can use a binary tree representation. The leaves of the binary tree correspond to the blocks and each internal node defines a vertical or horizontal merge operation of its two descendents. The number of possible configurations for the tree is $O(n!2^{5n-3}/n^{1.5})$. Note that this complexity is an upper bound. There are efforts to identify the redundancy [5]. Other efforts have been published to extend the binary tree to the representation of nonslicing structure [9], [10].

For nonslicing floorplan, Onodera *et al.* [14] classify the topological relationship between two blocks into four classes and use the branch-and-bound method to solve the problem. The solution space for this approach is $O(2^{n(n+2)})$, which makes the problem too large to handle at a time.

In [6] and [7], sequence pair and bounded slicing grid approaches were presented to handle nonslicing floorplan with smaller solution space. These two approaches are different representations, but they are basically based on a constraint graph to manipulate the transformation between the representation and their placement, which makes it complicated.

In [6], Murata *et al.* propose a sequence pair representation. They use two sets of permutations to present the geometric relation of blocks. Thus, the combination of the sequence pair is $O((n!)^2)$. From sequence pair to its placement, the transformation operation takes $O(n \lg n)$ time [11]. In [7], Nakatake *et al.* devised a bounded slicing grid approach. An n by n grid plane is used for the placement of n blocks. The representation itself has much redundancy; one placement may have several choices of representations.

Xu *et al.* [8] propose an iterative approach to optimize area and interconnection by cluster refinement. For a small k such as the cluster size, the runtime complexity for each iteration is $O(n^{2+k/2})$. This approach is central processing unit (CPU) intensive and difficult to handle if we choose a larger cluster size.

B. Contributions

The results of previous research show that the complexity of the problem increases a lot from slicing floorplan to nonslicing floorplan. It is challenging to find a comparable or even better representation for nonslicing floorplan.

Our thought is encouraged by the observation that any floorplan is bound on a two-dimensional (2-D) plane and could be represented by a planar graph. There might be some means to reduce the redundancy of the floorplan representation.

We first focus on a class of placement defined as admissible. Given a placement, we can derive an admissible placement by compacting the blocks to the left and to the bottom edges. An admissible placement is a compacted placement in which all blocks can neither move down nor move left. A rooted O tree is devised to represent the admissible placement. In the following, we describe the advantages of O tree.

- O tree takes only $n(2 + \lceil \lg n \rceil)$ bits to describe, where n is the number of blocks. Note that a sequence pair takes $2n\lceil \lg n \rceil$ bits . Even a binary tree for slicing structure takes $n(6 + \lceil \lg n \rceil)$ bits.

Manuscript received September 6, 1999. This work was supported in part by grants from the National Science Foundation (NSF) under Project MIP-9987678 and the California MICRO Program. This paper was recommended by Associate Editor M. Sarrafzadeh.

P.-N. Guo is with the Mentor Graphics Corporation, San Jose, CA 95131 USA (e-mail: pn_guo@mentor.com).

T. Takahashi is with Niigata University, Niigata, Japan (e-mail: takahasi@ie.niigata-u.ac.jp).

C.-K. Cheng is with University of California at San Diego, La Jolla, CA 92093-0114 USA (e-mail: kuan@cs.ucsd.edu).

T. Yoshimura is with the NEC Corporation, Kawasaki, Japan (e-mail: yoshi@ccm.cl.nec.co.jp).

Publisher Item Identifier S 0278-0070(01)00939-3.

0278–0070/01$10.00 © 2001 IEEE

- The runtime for transforming an O tree to its representing placement is linear to the number of blocks, i.e., $O(n)$. For a sequence pair, its takes $O(n \lg n)$ operations to construct the placement. Note that it also takes $O(n)$ operations to construct a slicing structure from a binary tree.
- Given an admissible placement, there is an O-tree representation. The combination of O tree is $O(n!2^{2n-2}/n^{1.5})$. This is very concise compared with the sequence pair. The combination of a sequence pair is $O((n!)^2)$. The ratio of the complexity between a sequence pair and O tree is approximately $O(n^2(n/4e)^n)$. The complexity of O tree is even lower than a binary tree structure for a slicing floorplan that has $O(n!2^{5n-3}/n^{1.5})$ combinations.
- We prove that the transformation between O tree and constraint graph can be done in linear time. This shows that the O tree is equivalent to constraint graph for admissible placement.
- Another benefit of using O tree is that the compaction operation is already included in the structure. One instance of O tree will map into exactly one placement. The transformation and its compaction are done at the same time. O tree needs no extra effort for the computation of compact operations.
- Because of the simplicity of O tree, we can easily contemplate interconnect cost or other considerations as well as area cost. The optimized chip size and wire plan can improve the quality and performance of physical layout.
- We use a deterministic algorithm to demonstrate the first approach using the O-tree structure. The algorithm is very straightforward and very fast. Within a couple of tens of seconds, we can obtain competitive and even much better solutions to other CPU-intensive approaches in MCNC benchmark cases.

The paper is organized as the following. Section II states the floorplanning problem. Section III gives the descriptions of admissible placement and constraint graph. Section IV defines the properties and operations for O tree. Section V presents a deterministic algorithm based on O tree. Section VI shows our experimental results for MCNC benchmarks. Further, potential applications and heuristics based on the O-tree structure that could improve our basic deterministic version of the approach conclude this paper in the last section.

II. Problem Statement

A set $\mathbf{B} = \{B_1, B_2, \ldots, B_n\}$ of rectangular blocks lie parallel to the coordinate axes. Each rectangular block B_i is defined by a tuple (h_i, w_i), where h_i and w_i are the height and the width of block B_i, respectively.

A placement $P = \{(x_i, y_i)\colon 1 \leq i \leq n\}$ is an assignments of coordinates to the lower left corners of the rectangular blocks such that there is no two rectangular blocks overlapping. A representation of a placement is a set of structures and operations that realizes P.

The cost function we use for a placement consists of two parts: one is the area of the smallest rectangle that encloses the placement and the other is the interconnection cost between rectangular blocks.

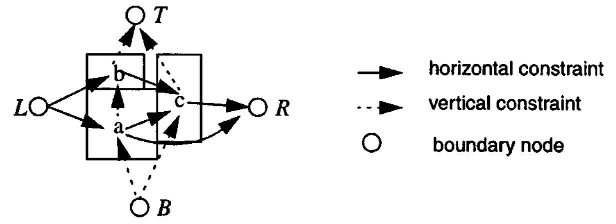

Fig. 1. Constraint graph.

III. Admissible Placement and Constraint Graph

A. Constraint Graph

A constraint graph for a placement is a graph $G = (V, E)$, where the nodes in V are placement blocks with additional four nodes used for the boundaries of the placement and the edges in E are the geometric constraints between two blocks. A geometric constraint exists when we can draw a horizontal or vertical line between two blocks without passing through other blocks.

The edges in E are directed. There are two kinds edges: one with the direction from a left node to a right node and another with the direction from a bottom node to a top node. The weight $d(e)$ for an edge $e = (B_i, B_j)$ is the separation distance between two nodes: $d(e)$ is equal to $x_j - x_i - w_i$ for horizontal edge and $y_j - y_i - h_i$ for vertical edge. The weight $d(e)$ is equal to zero when two nodes B_i and B_j are adjacent to each other; otherwise, it is positive.

In Fig. 1, a constraint graph is shown for the placement of three blocks a, b, and c. In addition to the three blocks represented by three nodes, four nodes L, R, T, and B are added to represent the four boundaries of the placement. The weights of edges (b, c), (a, R), and (B, c) are positive, and the others are zero.

Since we consider geometric constraints in two dimensions, the edges in constraint graph can be divided into two sets: E_h for horizontal constraints and E_v for vertical constraints. Then, we have the horizontal constraint graph $G_h = (V, E_h)$ and the vertical constraint graph $G_v = (V, E_v)$. Both G_h and G_v are s-t planar directed acyclic graphs (s-t PDAG). In Fig. 1, horizontal constraints are drawn in solid lines and vertical constraints in dotted lines.

Both constraint graphs G_h and G_v are planar because the placement is planar and there is no edge crossing other edge. According to graph theory [15], the number of edges in a planar graph is less than or equal to three times of the number of nodes minus six. Therefore, we have the following lemma.

Lemma 1: G_h and G_v are planar graphs and both sizes $|E_h|$ and $|E_v|$ are less than $3|V| - 6$ for $|V| > 3$.

B. L-Compact and B-Compact Placement

A placement is L compact if and only if there is no block that can shift left from its original position with other components fixed. In other words, a placement is L compact when it is x direction compacted to the left edge. The definition of B-compact placement is similar, substituting "left" with "bottom" and "x direction" with "y direction."

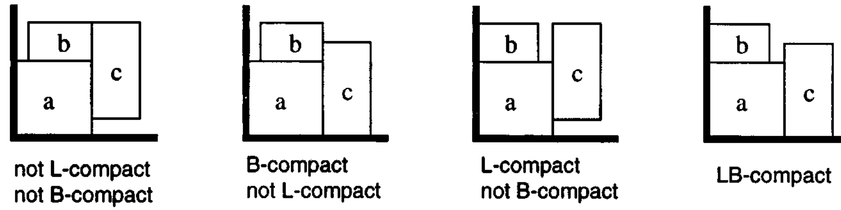

Fig. 2. L-compact, B-compact, and L-B compact placements.

C. LB-Compact Placement

A placement is LB compact if and only if it is both L compact and B compact. Examples of placement with L-compact, B-compact, and LB-compact properties are given in Fig. 2.

For any placement, we can fix the bottom and left edges and perform x direction and y direction compaction iteratively. The final placement after all compact operations converge is an LB-compact placement in respect to the original placement. Thus, we have the following lemma.

Lemma 2: Given any placement P_1, we can find a corresponding LB-compact placement P_2 by a sequence of x direction and y direction compactions. The overall area of P_2 is equal to or less than the overall area of P_1.

Proof: Neither x direction compaction nor y direction compaction increases the overall area of placement. Thus, the overall area of P_2 is equal to or less than the overall area of P_1.

D. Admissible Placement

A placement is admissible if it is a LB-compact placement.

IV. O Tree and Placement

A tree contains a finite set T of one or more nodes. There is one specially designated node called the root of the tree. The root has zero or more branches and the branches are directed edges pointed from the root to its children. Let $m \geq 0$, $T_1, \ldots, T_m$ be a set of trees. We call $T_1, \ldots, T_m$ the subtrees of the root.

An O tree is a rooted directed tree in which the order of the subtrees $T_1, \ldots, T_m$ is important. When we visit the tree using depth-first search (DFS), the order of the subtrees $T_1, \ldots, T_m$ determines the DFS order when we traverse the tree.

A. Tree Encoding with $(\boldsymbol{T}, \pi)$

To encode a rooted O tree with n nodes [12], we need a $2(n-1)$ bit string $\boldsymbol{T}$ to identify the branching structure of a tree and a permutation π as the labels of n nodes. The bit string $\boldsymbol{T}$ is a realization of the tree structure. We write a "**0**" for a traversal which descends an edge and a "**1**" when it subsequently ascends that edge in tree. The permutation π is the label sequence when we traverse the tree in DFS order. The first element in permutation π is the root of tree. The following example demonstrates the encoding of an eight-node rooted O tree.

Given an eight-node tree as shown in Fig. 3, its root node has three subtrees rooted at a, b, and c. We can represent it by (00110100011011, $adbcegf$). Starting from the root, we visit node a first and record a bit "**0**" to $\boldsymbol{T}$ and a label "a" to π. Then, we visit node d and record a bit "**0**" to $\boldsymbol{T}$ and a label "d" to π. On the way back to the root from nodes d and a, we record two bits "**11**" to $\boldsymbol{T}$. Then, we visit subtrees b and c in sequence and record the remaining of $\boldsymbol{T}$ and π, respectively. The length of the bit string $\boldsymbol{T}$ is 14.

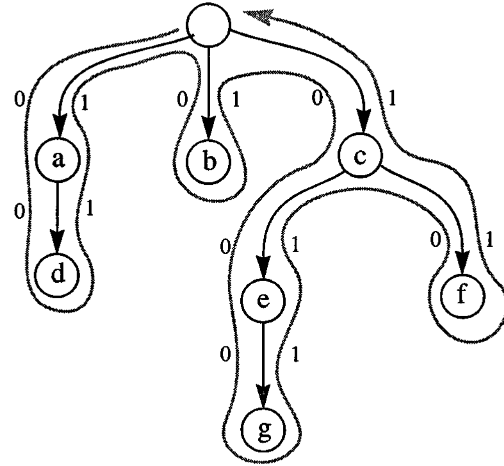

Fig. 3. Encoding of an eight-node tree.

B. Space Needed to Store $(\boldsymbol{T}, \pi)$

Given a tree with n nodes in addition to its root, each label of node can be encoded into a $\lceil \lg n \rceil$ bit string. Therefore, we need $n(2 + \lceil \lg n \rceil)$ b to store $(\boldsymbol{T}, \pi)$, where $2n$ bits for $\boldsymbol{T}$ and $n\lceil \lg n \rceil$ bits for π.

C. Count of Possible $(\boldsymbol{T}, \pi)$ Configurations

The total number of possible $(\boldsymbol{T}, \pi)$'s for a n-node tree is the product of possible configurations of bit string $\boldsymbol{T}$ and permutation π. For permutation π, the number of possible configurations is equal to $n!$. For bit string $\boldsymbol{T}$, it is the bit string with n **0**s and n **1**'s that presents an unlabeled n-node O tree. In [13], the possible configurations for an unlabeled n-node O trees is

$$\frac{1}{n}\binom{2n-2}{n-1} \tag{1}$$

by Stirling's approximation

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n. \tag{2}$$

Applying (2), we can derive (1) to its asymptotic form

$$\frac{4^{n-1}}{n\sqrt{\pi n}} + o\left(\frac{4^{n-1}}{n^{2.5}}\right) \approx \mathrm{O}\left(\frac{2^{2n-2}}{n^{1.5}}\right). \tag{3}$$

Thus, the total number of possible $(\boldsymbol{T}, \pi)$'s for n-node tree is $\mathrm{O}(n!2^{2n-2}/n^{1.5})$.

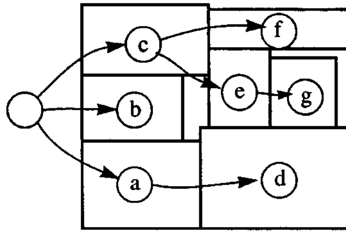

Fig. 4. O tree and its corresponding placement.

D. Horizontal O Tree

A horizontal O tree $(\boldsymbol{T}, \pi)$ represents a placement by the following ways. The nodes in $(\boldsymbol{T}, \pi)$ is the set of placement blocks B and an additional left boundary node as the root. The edges in $(\boldsymbol{T}, \pi)$ determine the horizontal related positions between blocks and the permutation π determines their vertical relationship. The definition is as follows.

The root of the O tree represents the left boundary of the chip. Thus, we set its x coordinate $x_{\text{root}} = 0$ and its width $w_{\text{root}} = 0$. The children are on the right side of their parent with zero separation distance in x coordinate. Letting B_i be the parent of B_j, we have

$$x_j = x_i + w_i. \tag{4}$$

The permutation π determines the vertical position of the component when two blocks have proper overlap in their x-coordinate projections. For each block B_i, let $\psi(i)$ be the set of block B_k with its order lower than B_i in permutation π and interval (x_k, x_k+w_k) overlaps interval (x_i, x_i+w_i) by a nonzero length. If $\psi(i)$ is nonempty, we have

$$y_i = \max_{k\in\psi(i)} y_k + h_k \tag{5}$$

otherwise

$$y_i = 0. \tag{6}$$

From an horizontal O tree, we can find a placement by visiting the tree in DFS order. The placement is always B compact by its definition, but not necessarily L compact. Fig. 4 shows a placement that is represented by the horizontal O tree in Fig. 3.

E. Vertical O Tree

A vertical O tree is similar to the horizontal O tree described above and uses a bottom edge as the root of tree.

F. Admissible O Tree

An O tree is admissible if its corresponding placement is admissible. Fig. 5 and shows two O trees, where (a) is admissible and (b) is not.

Lemma 3: Given an admissible O tree (AOT), it is equal to the shortest path spanning tree (SPST) embedded in the constraint graph of its corresponding placement.

Proof: Without loss of generality, we can assume that a horizontal O tree OT_h is admissible and has an admissible placement by definition. Supposing OT_h is not an SPST of the horizontal constraint graph G_h of its corresponding placement, we can find an edge $e = (B_i, B_j)$ in OT_h that is not in the SPST of G_h. Since edge e is not shortest, we can compact the placement between block B_i and block B_j, which contradicts the assumption. Thus, OT_h is a SPST of G_h.

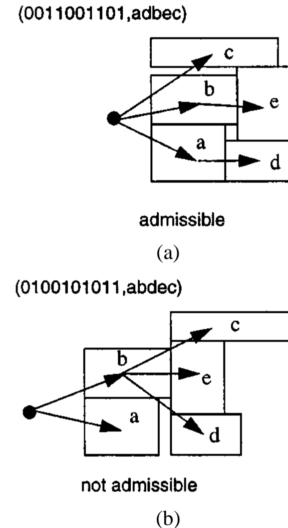

Fig. 5. Illustration of (a) admissible and (b) not admissable O tree.

Corollary: Given an admissible placement, we can construct a horizontal constraint graph. The SPST of the graph is the horizontal O tree of the placement. The same result applies to vertical O tree, too.

G. O Tree to Its Orthogonal Constraint Graph

Given an O tree, we can build up its orthogonal constraint graph by using DFS and maintaining a contour structure. Based on the definition of O tree, we develop algorithm O tree to its orthogonal constraint graph (OT2OCG), which first finds the corresponding placement of the O tree by solving (4)–(6) and then builds up its orthogonal constraint graph.

```
Algorithm OT2OCG
  Input: O-tree (T[0:2n-1], Π[0:n])
  Output: orthogonal constraint graph
  G = (V, E) and placement x[1:n], y[1:n]
                    V = Π + {V_s, V_t};
  perm = 1;
  contour = NULL;
  current_contour = 0;
  for code = 0 to 2n-1
    if T[code] = 0
      current_block = Π[perm];
      if current_contour = 0
        then x[current_block] = x
  [current_contour] +w [current_contour];
        else x [current_block] = 0;
      end if
      y [current_block] =
  find_max_y(contour, current_block)
      update_constraint_graph(G, contour,
  current_block)
      update_contour(contour,
  current_block)
```

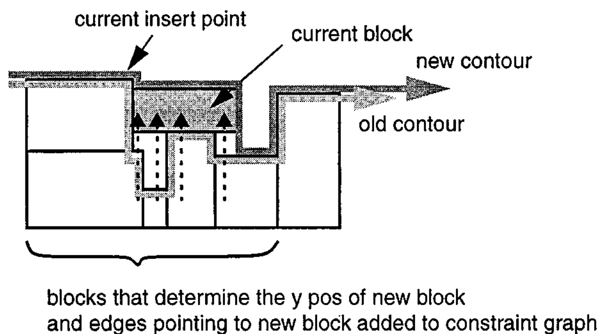

Fig. 6. Constraint graph and contour for OT2OCG.

```
    current_contour = current_block;
  else
    current_contour = prev[current_con-
tour];
  end if
end for
```

A contour structure is used in OT2OCG algorithm to reduce the runtime for finding the y coordinate of a block while solving (5) and (6). The runtime is linear to the number of blocks without the contour structure. By maintaining a contour structure, the amortized cost of finding any y coordinate becomes a constant time.

The contour structure is a double-linked list of blocks that describes the contour line in current compact direction. We use a variable *current_contour* to record the block where we want to insert next block to in the contour. Fig. 6 shows how *find_max_y* determines the y coordinate of current block, how *update_constaint_graph* adds edges in the constraint graph, and how *update_contour* updates the contour structure when we add a new block to the placement.

Lemma 4: The runtime for algorithm OT2OCG is linear to the number of blocks.

Proof: Without loss of generality, assume we can construct a vertical constraint graph from a horizontal O tree by OT2OCG. Suppose we have n blocks.

1) For the loop in algorithm OT2OCG, we visit each node exactly twice: one at the node's encode "**0**" and the other at encode "**1**." The loop executes exactly $2n$ times.
2) In the loop, we perform three operations: *find_max_y*, *update_contour*, and *update_constraint_graph* for each block B_i inserted.
3) With maintaining contour structure and *current_contour* pointing to the current starting point in the contour, we can keep tracing the contour until the y coordinate is $> x_i + w_i$.
4) By (3), we need to pass only a limited set of blocks to three operations in (2) instead of passing $|\psi(i)|$ number of blocks. The number of blocks accessed is equal to the number of edges inserted in vertical constraint graph.
5) The constraint graph is planar. By lemma 1, the number edges in vertical constraint graph $G_v = (V, E_v)$ is less than $3n-6$. The overall complexity for OT2OCG is linear because we add every edge in G_v exactly once.

Thus, the amortized complexity for each update operation of the constraint graph and contour structure is constant. Therefore, the algorithm OT2OCG has runtime complexity $O(|V|)$, which is linear to the number of blocks in placement. Q.E.D.

H. Constraint Graph to Its O tree

Given a constraint graph, we can build its O tree by an SPST algorithm. Because the constraint graph created by algorithm OT2OCG is either L or B compact, we can construct an O tree whose edges all have weights equal to zero. Instead of using a breadth-first search algorithm as a traditional approach of SPST, we use DFS algorithm constraint graph to O tree (CG2OT), which has the same performance and needs less explicit memory space. The runtime of this algorithm is linear to the number of blocks.

```
Algorithm CG2OT
  Input: constraint graph G = (V, E)
  Output: O-tree T[0:2n − 1], Π[o:n])
    set all mark to false
    perm = 0;
    code = 0;
    DFS traverse on the graph G
      n = current node
      p = parent[n]
      if not mark [n] and the weight
  [edge(p, n)] = 0 then
        mark [n] = true
        Π[perm++] = 0;
        T[code++] = n;
        for c in children [n]
          traverse (c);
        end for
        Π [perm++] = 1;
      end if
```

I. Admissible O Tree

Given any O tree, we can construct an AOT by invoking OT2OCG and CG2OT iteratively in sequence until convergence is achieved. Given a horizontal O tree T, we can get a vertical constraint graph G_y by OT2OCG. Because G_y is B compact, we can get a vertical O tree T_y by CG2OT. After applying the same procedures (OT2OCG and CG2OT), we can get a horizontal O tree that represents an L-compact placement.

All moves in the compaction are monotone because all blocks are either moving down or moving left. Therefore, the convergence of the above iteration is assured and we can get an AOT.

```
Algorithm AOT
  Input: O tree T
  Output: Admissible O-tree
  set changed = true
  while changed
    set changed = false
    set Gy = OT2OCG(T)
    set Ty = CG2OT(Gy)
    set Gx = OT2OCG(Ty)
```

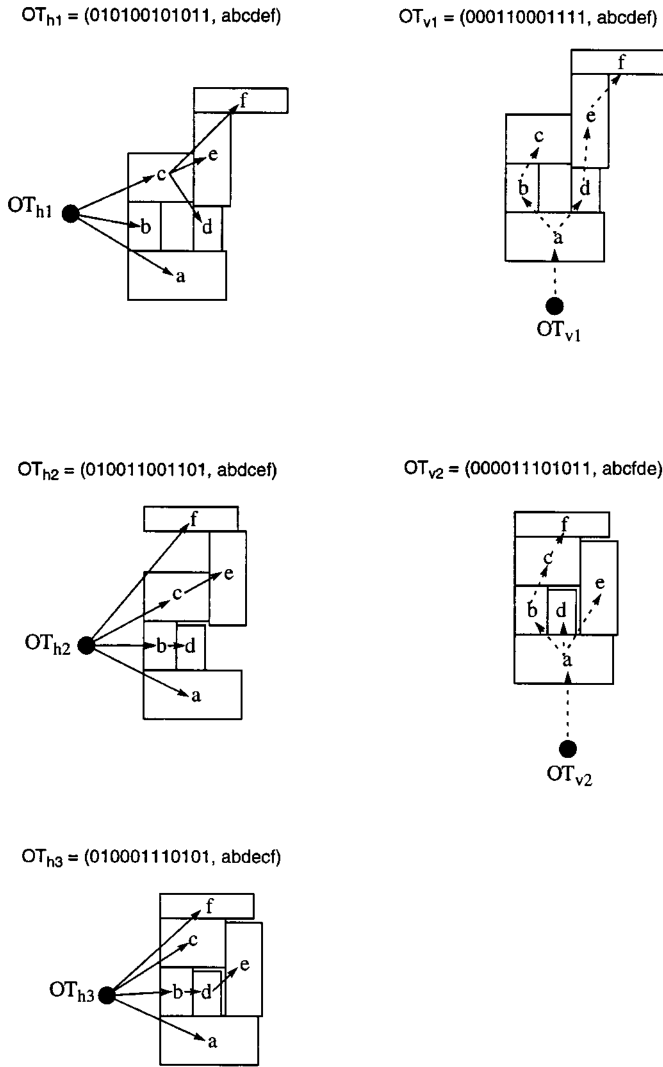

Fig. 7. Illustration of AOT process.

```
    set Tx = CG2OT(Gx)
    if (T is not equal to Tx) then
      set T = Tx
      set changed = true
    end if
  end while
  output(T)
```

Example: Assume the (w, h) tuples for blocks a, b, c, d, e, and f are (12,60), (4,6), (8,6), (3,5), (5,11), and (11,3), respectively. An O tree OT_{h1} (010100101011, $abcdef$) is not admissible. We run AOT on OT_{h1} and get a pair of O trees OT_{v1} and OT_{h2} in the first iteration. In the second iteration, we have OT_{v2} and OT_{h3}. Finally, in Fig. 7, the algorithm converges and finds an AOT OT_{h3} (010001110101, $abdecf$).

Lemma 5: All operations OT2OCG and CG2OT in the main loop are linear. The runtime complexity for each iteration of the main loop in AOT is linear to the number of blocks n.

V. Floorplan Algorithms Using O Tree

We develop a deterministic floorplan algorithm using the O tree structure described in Section IV. The basic idea is systematically perturbing a given O tree to find an optimized solution according to a preset cost function.

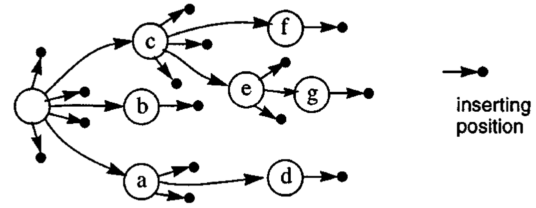

Fig. 8. Illustration of possible inserting positions.

TABLE I
Characteristics of MCNC Benchmark

circuit	#block	#net	#pin	#io
apte	9	97	214	73
xerox	10	203	696	2
hp	11	83	264	45
ami33	33	123	480	42
ami49	49	408	931	42

A. Perturbing the O Tree

We can perturb the O tree by the following steps:

1) select a block B_i in the original O tree $(\boldsymbol{T}, \pi)$;
2) delete block B_i from O tree $(\boldsymbol{T}, \pi)$;
3) insert block B_i in the position where we can get the best value of cost function among all possible inserting positions in $(\boldsymbol{T}, \pi)$ as an external node;
4) perform 1–3 on its orthogonal O tree.

The selection of inserting position from all above will create many useful configurations for the perturb operation. We may also select another method to insert a node to the tree, but it needs additional operations to split and merge its descending subtrees. It is our choice not to include them in our approach.

Given any O tree with n nodes, the number of possible inserting positions as external nodes is $2n-1$. In Fig. 8, there are 15 possible inserting positions in an eight-node tree. The operation of finding these positions on $(\boldsymbol{T}, \pi)$ is to simply add a string "01" to any position in bit string $\boldsymbol{T}$ and add the label to its related position in π.

A perturbed O tree need not be admissible. We can apply AOT to get an AOT and then evaluate it by the preset cost function to find which move is the best.

A deterministic algorithm is derived by perturbing O tree in sequence. We select nodes in sequence and find the best perturb position for each of them. Given a fixed sequence of node, we can always find a best O tree and its corresponding placement. The advantage of deterministic algorithm is that its implementation is straightforward and easy to comprehend.

```
Deterministic Algorithm
 Input:
   array of blocks with width, height,
 and pin positions
   I/O pad position and networks
 Output:
```

TABLE II
AREA, WIRE LENGTH, AND CPU COMPARISON. (CPU SECONDS ON 90-MHz SPARC 20 WORKSTATION)

circuit	cluster refinement	initial placement	deterministic algorithm
apte	48.4/321/224*	63.3/330/0.14	63.3/330/0.65
xerox	20.3/477/18.8*	25.9/506/0.44	23.8/478/0.99
hp	9.58/185/18.0*	14.3/178/0.26	9.91/167/6.32
ami33	1.21/64/603*	1.69/61.9/2.83	1.34/50.9/24.3
ami49	37.7/764/1860*	54.6/676/11.2	45.5/673/177.5

TABLE III
MINIMUM AND AVERAGE DISTRIBUTION WITH DIFFERENT WEIGHTS

circuit	w_1=0, w_2=1		w_1=w_2=0.5		w_1=1, w_2=0		improve over CR (area/wire)
	area	wire	area	wire	area	wire	
apte	48.3/56.9	317/347	47.6/53.2	317/370	47.1/50.6	343/544	3% / 1%
xerox	20.4/24.1	368/426	20.4/22.4	367/447	20.1/21.4	444/702	1% / 23%
hp	9.71/11.2	153/163	9.21/10.5	153/167	9.21/9.97	162/226	4% / 17%
ami33	1.26/1.41	51.5/57.2	1.26/1.34	51.6/59.8	1.25/1.32	61.1/87.4	-3% / 20%
ami49	41.3/49.8	636/734	39.1/42.0	671/777	37.6/39.9	819/1375	0% / 17%

```
  blocks with position and orientation
Algorithm:
  initiate O tree T and its placement
  for each block b
    set min_cost = infinite
    remove (T, b)
    for each possible position p of b in
T and T's orthogonal
      set T1 = new O tree and placement
for p
      get admissible T1 using AOT
      set c = cost (T1)
      if c < min_cost then
        set min_cost = c
        set min_T = T1
      end if
    end for
    set T = min_T
end for
Output placement for T
```

Similar to the method of the deterministic algorithm, we can get a constructive algorithm for initial placement with the following algorithm.

```
Initiate (Constructive Algorithm)
  set O-tree T = {}
  same as the main loop above, replace
  cost () by partial_cost()
  Output T
```

There are two *for* loops in the algorithm. The first loop perturbs all blocks in placement for total of n times and the second one evaluates $4n-2$ possible inserting points in the O tree. The function AOT in the inner loop contributes O(n) times to the overall procedure. Therefore, the runtime complexity for the algorithms is O(n^3).

VI. EXPERIMENTAL RESULTS

The experiments are carried out for the MCNC building block examples. There are five test cases and the number of blocks range between nine and 49. The largest case *ami49* is a circuit with 49 blocks, 42 I/O pads, 408 nets, and 931 pins. The circuit characteristics in MCNC benchmark are given in Table I.

Our program is written in C language. The core part of O-tree operation is around 1000 lines of codes, and the overall package, including an X Windows interface, is a little more than 6000 lines of source codes. The program runs on the platform of a 200-MHz Ultra-1 Sparc station with 512-MB memory.

We compare the wire length and chip area with the result of cluster refinement [8]. Table II shows the results of initial placement using the given sequence order and the results after one run of deterministic algorithm. The cost function here is solely by the wire length, which is the sum of half bounding box of all nets in the circuit. In each table entry, there are three numbers: the first number is the chip area (mm^2), the second number is the wire length (mm), and the last number is the CPU time in seconds.

In Table II, we achieved results with better wire length for the three largest cases while their chip area are comparable. Note that the CPU time is much less than the cluster refinement approach. Comparable results can be reached with only a few minutes for the largest case.

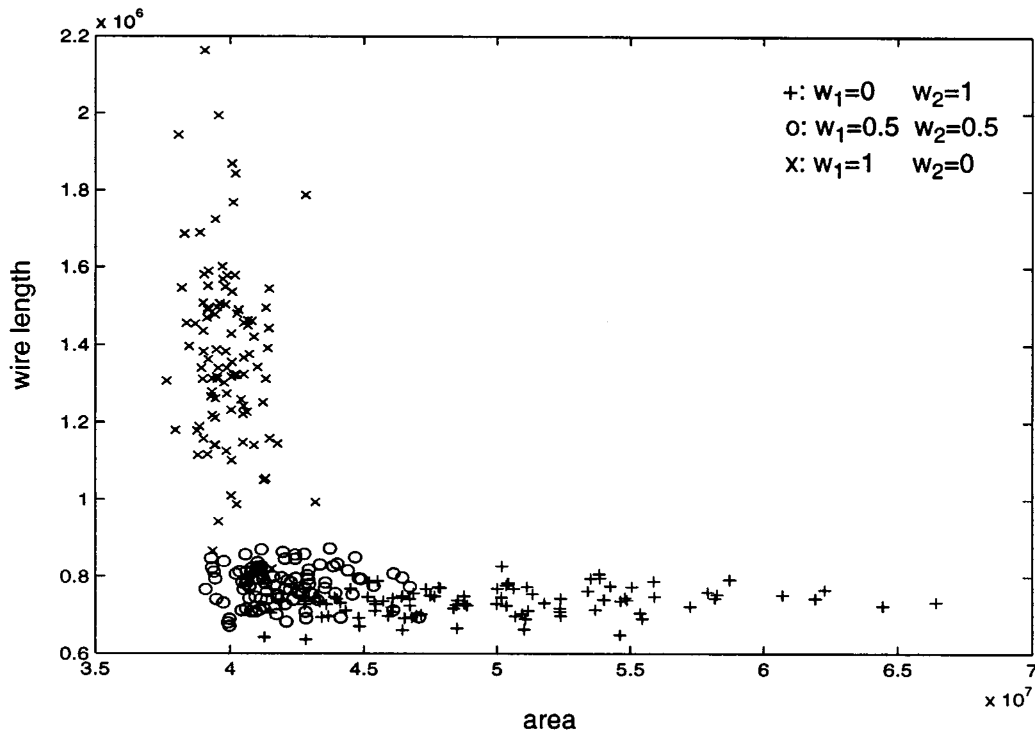

Fig. 9. Placement results of random sequence with different weights.

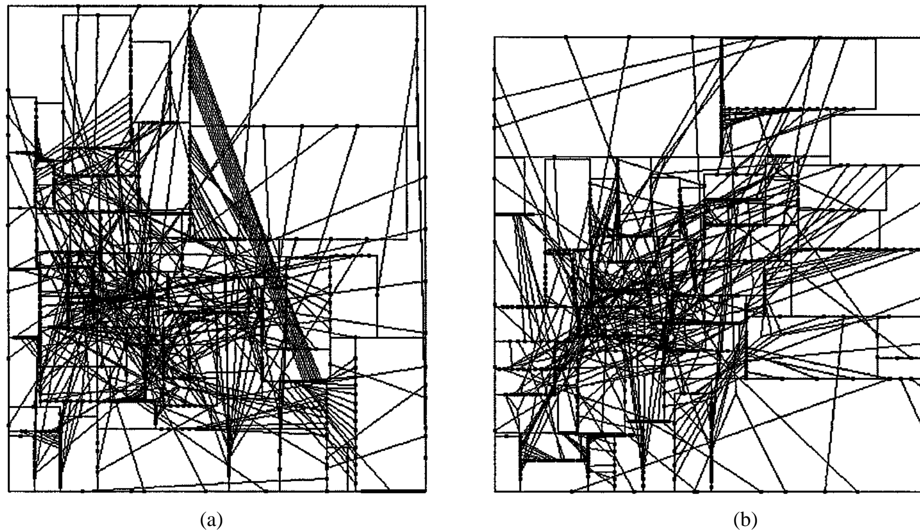

Fig. 10. (a) Placements before improvement. Area = 40.8 (5.92 x 6.89). Wire length = 810. (b) Placements after deterministic improvement of *ami49*. Area = 39.9 (6.17 x 6.47). Wire length = 680.

Based on the basic version of initial and deterministic algorithm, we can use a randomly generated sequence instead of the original sequence in MCNC benchmarks. In conjunction with different weights to area and to wire length in cost function, we have the results that an optimized solution can be found when the weights are balanced.

In Table III, the distribution for the results of MCNC testcase using 100 runs of randomized sequences is given. We use a cost function like $w_1 * area + w_2 * wirelength$, where the weights w_1 and w_2 are for area and wirelength, respectively, and the two terms area and wirelength are normalized. Table III shows three sets of $\{w_1, w_2\}$ values: $\{0, 1\}$, $\{0.5, 0.5\}$, and $\{1, 0\}$. Each table entry has two numbers: the first one is the minimum value of results among all runs, and the second one is the average of the results.

Fig. 9 shows the area/wirelength plot for the *ami49* circuit. We run 100 randomized sequences for different weights in the cost function. When the weights are 0.5 for both area and wire length, the plots are very concentrated near the area of 45° line, where chip size and wire length are almost balanced at that region.

Comparing the best results with cluster refinement, we have 1% to 23% improvement in the wire length and -3% to 4% improvement in area. For hp circuit (see Table II), we can find a better solution which has 4% improvement in area and 17% improvement in wire length. On average, we can get about 16%

improvement in wire length while the chip area is comparable. In Fig. 10, the placement after improvement shows a better interconnection than the placement before improvement.

VII. Conclusion

We displayed a tree representation of floorplanning. The method reduces the redundancy by a compaction process. The exact floorplan topology is defined according to the exact block width and height. The tree structure is well known in applied mathematics and computer science and the properties of trees are very straightforward and simple.

Our algorithm shows improvement in both chip area and wire length. The implementation of algorithm is achieved with much less CPU time. Other measures, such as timing, congestion, and routability, are now formulated to our approach.

Acknowledgment

The authors would like to thank the reviewers for their helpful comments. They also appreciate the support of A. Takahashi and Y. Kajiitani for clarifying the independence of the tree-representation invention for compact-block placement by the two groups in the author list.

References

[1] B. Preas and M. Lorenzetti, *Physical Design Automation of VLSI Systems: 4.7 Floorplanning, 6.3 Compaction*. Redwood City, CA: Benjamin Cummings, 1988, pp. 129–132, 231–266.

[2] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout: 7.2 General Cells: Floorplanning, 10. Compaction*, New York: Wiley, 1990, pp. 328–361, 579–647.

[3] B. T. Preas and W. M. VanCleemput, "Placement algorithms for arbitrarily shaped blocks," in *Proc. ACM/IEEE Design Automation Conf.*, 1979, pp. 474–480.

[4] R. H. J. M. Otten, "Automatic floorplan design," in *Proc. ACM/IEEE Design Automation Conf.*, 1982, pp. 261–267.

[5] D. F. Wong and C. L. Liu, "A new algorithm for floorplan design," in *Proc. ACM/IEEE Design Automation Conf.*, 1986, pp. 101–107.

[6] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajatani, "Rectangular-packing-based module placement," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 1995, pp. 472–479.

[7] S. Nakatake, K. Fujiyoshi, H. Murata, and Y. Kajitani, "Module placement on BSG-structure and IC layout applications," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 1996, pp. 484–491.

[8] J. Xu, P.-N. Guo, and C.-K. Cheng, "Cluster refinement for block placement," in *Proc. ACM/IEEE Design Automation Conf.*, 1997, pp. 762–765.

[9] T.-C. Wang and D. F. Wong, "An optimal algorithm for floorplan area optimization," in *Proc. ACM/IEEE Design Automation Conf.*, 1990, pp. 180–186.

[10] P. Pan and C. L. Liu, "Area minimization for floorplans," *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 123–132, Jan. 1995.

[11] T. Takahashi, "An algorithm for finding a maximum-weight decreasing sequence in a permutation, motivated by rectangle packing problem," *Tech. Rep. IEICE*, vol. VLD96, no. 201, pp. 31–35, 1996.

[12] K. Keeler and J. Westbrook, "Short encoding of planar graphs and maps," *Discrete Appl. Math.*, vol. 58, pp. 239–252, 1995.

[13] D. E. Knuth, *The Art of Computer Programming*, 2nd ed. Reading, MA: Addison-Wesley, 1973, vol. 1, pp. 385–395.

[14] H. Onodera, Y. Taniguchi, and K. Tamaru, "Branch-and-bound placement for building block layout," in *Proc. IEEE/ACM Design Automation Conf.*, 1991, pp. 433–439.

[15] R. Diestel, *Graph Theory*, New York: Springer-Verlag, 1997, pp. 81–90.

[16] MCNC Electronic and Information Technologies [Online]. Available: www.mcnc.org

Pei-Ning Guo received the B.S. degree in computer science from the National Taiwan University, Taiwan, R.O.C, in 1988, the M.S. degree in computer science from New York University, New York, in 1994, and the Ph.D. degree in computer science and engineering from the University of California, San Diego, in 1998.

He joined the Mentor Graphics Corporation, San Jose, CA, as a Senior Engineer in 1999. His research interests include floorplan and placement approaches for very large scale integration (VLSI) physical design.

Toshihiko Takahashi received the B.E., M.E., and D.E. degrees from the Tokyo Institute of Technology, Tokyo, Japan, in 1985, 1988, and 1991, respectively.

He is currently an Associate Professor at the Graduate School of Science and Technology, Niigata University, Niigata, Japan. His current research interests include combinatorial algorithms and very large scale integration (VLSI) layout design.

Chung-Kuan Cheng (S'82–M'84–SM'95–F'00) received the B.S. and M.S. degrees in electrical engineering from National Taiwan University, Taiwan, R.O.C, in 1976 and 1978, respectively, and the Ph.D. degree in electrical engineering and computer sciences from the University of California, Berkeley, in 1984.

From 1984 to 1986, he was a Senior Computer-Aided Design Engineer at Advanced Micro Devices Inc. In 1986, he was with the University of California at San Diego, La Jolla, where he is currently a Professor in the Computer Science and Engineering Department and an Adjunct Professor in the Electrical and Computer Engineering Department. He served as a Chief Scientist at Mentor Graphics in 1999. His research interests include network optimization and design automation on microelectronic circuits.

Dr. Cheng received the NCR Excellence in Teaching Award in 1991 and the IEEE Transactions on Computer-Aided Design Best Paper Award in 1997. Since 1994, he has been an Associate Editor of the IEEE Transactions on Computer-Aided Design.

Takeshi Yoshimura (M'86) received the B.E., M.E., and Dr.Eng. degrees from Osaka University, Osaka, Japan, in 1972, 1974, and 1997, respectively.

He joined the NEC Corporation, Kawasaki, Japan, in 1974, where he has been engaged in the research and development of computer application systems for communication network design, hydraulic network design, very VLSI CAD. From 1979 to 1980, he was on leave at the Electronics Research Laboratory, University of California, Berkeley, where he worked on very large scale integration computer-aided design layout.

Dr. Yoshimura is a Member of the Institute of Electronics, Information, and Communication Engineers of Japan and the Information Processing Society of Japan. He received the Best Paper Award from the Institute of Electronics, Information, and Communication Engineers of Japan (IEICE).