An O-Tree Representation of Non-Slicing Floorplan and Its Applications

Pei-Ning Guo
Mentor Graphics Corp.
1001 Ridder Park Drive
San Jose, CA 95131, U.S.A.
pn_guo@mentor.com

Chung-Kuan Cheng*
Mentor Graphics Corp.
1001 Ridder Park Drive
San Jose, CA 95131, U.S.A.
ck_cheng@mentor.com

Takeshi Yoshimura
NEC Corp.
4-1-1 Miyazaki, Miyanae-Ku
Kawasaki 216, Japan
yoshi@swl.cl.nec.co.jp

ABSTRACT

We present an ordered tree, O-tree, structure to represent non-slicing 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 direction. 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 which 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 which 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 benchmarks show promising performance with average 16% improvement in wire length, and 1% less in dead space over previous CPU-intensive cluster refinement method.

1. 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.

1.1. Previous Works

For a floorplan with slicing structure[6][8], 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[11]. Other efforts have been published to extend the binary tree to the representation of non-slicing structure.[7][10]

For non-slicing floorplan, Onodera et al.[5] classify topological relationship between two blocks into four classes, and use branch-and-bound method to solve the problem. The solution space for this approach is $O(2^{n(n+2)})$, that makes the problem too complicate to handle at a time.

In [3][4], sequence pair and bounded slicing grid approaches were presented to handle non-slicing floorplan with smaller solution space. These two approaches are different representations but they are all basically based on constraint graph to manipulate the transformation between the representation and their placement, which makes it complicated.

In [3], 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[9]. In [4], 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 floorplan could have several choices of representations.

J. Xu et al.[12] propose an iterative approach to optimize area and interconnection by cluster refinement. For a small k as the cluster size, the run time complexity for each iteration is $O(n^{2+k/2})$. This approach is CPU-intensive and difficult to handle if we choose a larger cluster size.

1.2. Contributions

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

Our thought is encouraged by the observation that any floorplan is bound on a 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 where all blocks can neither move down nor move left. A rooted ordered tree, 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

This work is supported in part by grants from the NSF Project MIP-9529077 and the California MICRO program.
*Chung-Kuan Cheng is on leave from CSE Dept., UCSD.

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.

- The run-time 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 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 slicing floorplan which has $O(n!\; 2^{5n-3} / n^{1.5})$ combinations.
- We show 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 straight-forward and very fast. Within a couple tens of seconds, we can obtain competitive and even much better solutions to other CPU-intensive approaches in MCNC benchmark cases.

This paper is organized as follows: Section 2 states the floorplanning problem. Section 3 gives the descriptions of admissible placement and constraint graph. Section 4 defines the properties and operations for O-tree. Section 5 presents a deterministic algorithm based on O-tree. Section 6 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 are given to conclude this paper in the last section.

2. PROBLEM STATEMENT

A set $B = \{B_1, B_2, \ldots, B_n\}$ 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 $P = \{(x_i, y_i){:}\ 1 \le i \le n\}$ 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; the other is the interconnection cost between rectangular blocks.

3. ADMISSIBLE PLACEMENT AND CONSTRAINT GRAPH

A constraint graph for a placement are 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, 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.

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).

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, 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

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

Definition[L-compact and B-compact] 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 for B-compact placement is similar, substituting 'left' by 'bottom', and 'x-direction' by 'y-direction'.

Definition [LB-compact placement] A placement is LB-compact if and only if it is both L-compact and B-compact.

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 respect to the original placement. We have

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 or less than the overall area of P_1.

Definition [Admissible placement] A placement is admissible if it is a LB-compact placement.

4. 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, the branches are directed edges pointed from the root to its children. Let $m \ge 0$, $T_1, \ldots, T_m$ 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 $T_1,..., T_m$ is important. When we visit the tree using depth-first-search (DFS), the order of the subtrees $T_1,..., T_m$ determines the DFS order when we traverse the tree.

4.1. Tree Encoding with (T, π)

To encode a rooted ordered tree with n nodes[1], we need a $2(n-1)$-bit string T to identify the branching structure of tree, and a permutation π as the labels of n nodes. The bit string 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 depth-first search order. The first element in permutation π is the root of tree. The following example demonstrates the encoding of an 8-node rooted ordered tree:

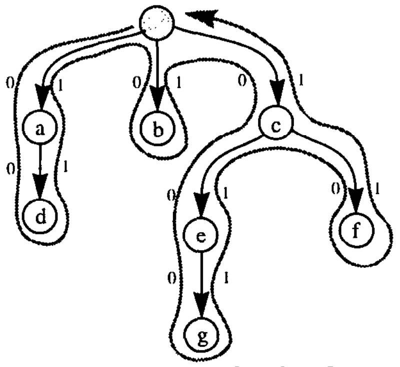


Figure 1: Encoding of an 8-node tree

Given a 8-node tree shown in Fig. 1, 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 T and a label 'a' to π. Then we visit node d and record a bit '**0**' to 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 T. Then we visit subtrees b and c in sequence, and record the remaining of T and π respectively. The length of the bit string T is 16.

Space needed to store (T,π) 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 $n(2 + \lceil \lg n \rceil)$ bits to store (T,π) where $2n$ bits for T, and $n\lceil \lg n \rceil$ bits for π.

Count of possible (T,π) configurations The total number of possible (T,π)'s for a n-node tree is the product of possible configurations of bit string T and permutation π. We can derive the asymptotic form[2] of the number of configurations as $O(n!2^{2n-2}/n^{1.5})$.

4.2. Horizontal O-Tree

A horizontal O-tree (T,π) represents a placement by the following ways: the nodes in (T,π) is the set of placement blocks B and an additional left boundary node as the root. The edges in (T,π) determines 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 $x_{root} = 0$ and its width $w_{root} = 0$. The children are on the right side of their parent with zero separation distance in x-coordinate. Let B_i be the parent of B_j, we have

$$x_j = x_i + w_i \qquad (1)$$

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 non-zero length. If $\psi(i)$ is non-empty, we have

$$y_i = max_{k \in \psi(i)}\, y_k + h_k \qquad (2)$$

otherwise $\quad y_i = 0 \qquad (3)$

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. 2 shows a placement which is represented by the horizontal O-tree in Fig. 1.

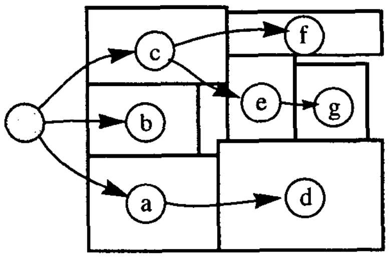


Figure 2: O-tree and placement

Similar to horizontal O-tree described above, and use a bottom edge as the root of tree.

Definition [Admissible O-Tree] An O-tree is admissible if its corresponding placement is admissible. Fig. 3 shows two O-tree where the one at the left is admissible, and the other is not.

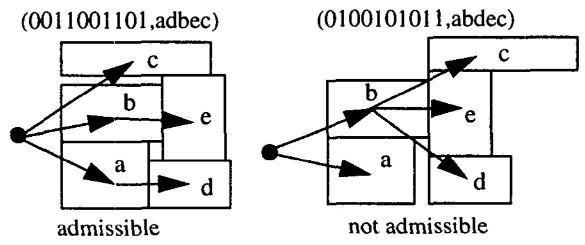


Figure 3: Admissible o-tree

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

Corollary Given an admissible placement, we can construct a horizontal constraint graph. The shortest path spanning tree of the graph is the horizontal O-tree of the placement. The same result applies to vertical O-tree as well.

4.3. O-Tree to Its Orthogonal Constraint Graph

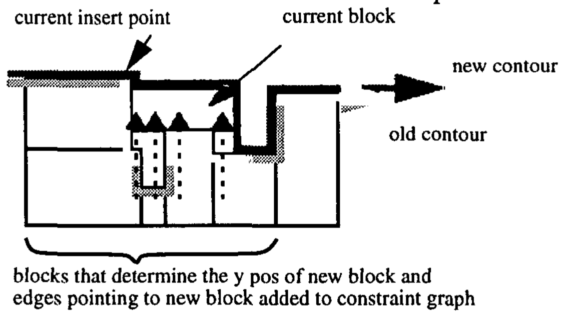
Given an O-tree, we can build up its orthogonal constraint graph by using DFS and maintaining a contour structure. Based on the definition of O-tree, we develop an algorithm (O-Tree To its Orthogonal Constraint Graph, **OT2OCG**) which first finds the corresponding placement of the O-tree by solving equations (1)-(3), 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), x[1:n], and y[1:n]
  set V = Π + {V_s,V_t};
  set perm = 1;
  set contour = NULL;
  set current_contour = 0;
  for code = 0 to 2n - 1
    if T[code] = 0 then
      set current_block = Π[perm];
      if current_contour = 0 then
        set x[current_block] = x[current_contour] + w[current_contour];
        else set x[curent_block] = 0;
      end if
      set y[current_block] = find_max_y(contour, current_block)
      update_constraint_graph(G, contour, current_block)
      update_contour(contour, current_block)
      set current_contour = current_block ;
      set perm = perm + 1
    else set current_contour = prev[current_contour];
    end if
  end for
```

A contour structure is used in **OT2OCG** algorithm to reduce the run time for finding the y-coordinate of a block while solving the equations (2) and (3). The run time 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, which 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. 4 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.



current insert point
current block
new contour
old contour
blocks that determine the y pos of new block and
edges pointing to new block added to constraint graph

Figure 4: updating constraint graph and contour

The algorithm visits each node twice, one at the node's encode '**0**' and the other at encode '**1**'. The time for updating constraint graph and contour structure is amortized constant time, because the total number of edge added for the constraint graph is $|E|$. Therefore, the run time is $O(|V| + |E|)$. By Lemma 2, we have

Lemma 4 The run time for algorithm OT2OCG is linear.

4.4. Constraint Graph to Its O-Tree

Given a constraint graph, we can build its O-tree by SPST algorithm. because the constraint graph created by algorithm **OT2OCG** is either L-compact 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 a depth-first-search algorithm (Constraint Graph To O-Tree, **CG2OT**) which has the same performance and needs less explicit memory space. The run time 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], P[0:n])
  set all mark to false
  set perm = 0;
  set code = 0;
  DFS traverse on the graph G
    set n = current node
    set p = parent[n]
    if not mark[n] and weight[edge(p,n)] = 0 then
      set mark[n] = true
      set P[perm++] = 0;
      set T[code++] = n;
      for c in children[n]
        traverse(c);
      end for
      set P[perm++] = 1;
    end if
```

4.5. Admissible O-Tree Transformation (AOT)

Given any O-tree, we can construct an admissible O-tree 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 which 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 above iteration is assured and we can get an admissible O-tree.

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

5. Floorplan Algorithms Using O-Tree

We develop a deterministic floorplan algorithm using the O-tree structure described in section 4. Systematically perturbing a given O-tree, we can find an optimized solution according to a preset cost function.

Perturbing the O-Tree

We perturb the O-tree by the following steps: (a) select a block B_i in the original O-tree (T,π), (b) delete block B_i from O-tree (T,π), (c) insert block B_i in the position with the best value of cost function among all possible inserting positions in (T,π) as an external node, and (d) perform (a)-(c) 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 other method to insert a node to the tree, but it needs more additional operation to split and merge its descending subtrees. It is one of our choices to not 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. 5, there are 15 possible inserting positions in a 8-node tree. The operation of finding these positions on (T,π) is simply adding a string '01' to any position in bit string T and adding the label to its related position in π.

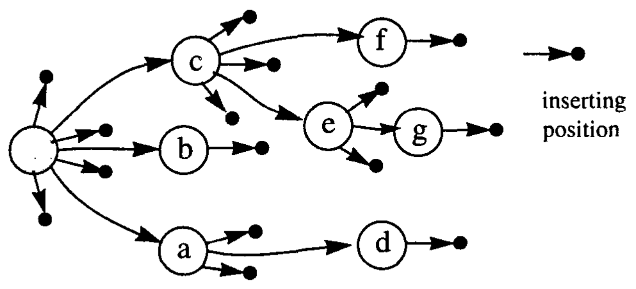


Figure 5: possible inserting positions as an external node

A perturbed O-tree need not be admissible. We can apply **AOT** to get an admissible O-tree 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 and I/O pad
position and networks
Output: blocks with position and orientation
  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 $T_1$ = new O-tree and placement for p
      get admissible $T_1$ using AOT
      set c = cost ($T_1$)
      if c < min_cost then
        set min_cost = c
        set min_T = $T_1$
      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 as the follow:

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 $O(n)$ time to the overall procedure. The run time complexity for the algorithms is $O(n^3)$.

6. Experimental Results

The experiments are carried out for the MCNC building block examples. There are five test cases and the number of blocks ranges between 9 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 can be found in [12].

Our program is written in C language. The core part of O-tree operation is around 1,000 lines of codes, and the overall package including an X Windows interface is a little more than 6,000 lines of source codes. The program is running on the platform of a 200MHz Ultra-1 Sparc station with 512MB memory.

We compare the wire length and chip area with the result of cluster refinement[12]. Table 1 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 one is the CPU time in seconds.

Table 1: Area / wirelength / CPU comparisons

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

In Table 1, 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, the comparable results can be reached with only a few minutes for the largest case.

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. Joint 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 2, the distribution for the results of MCNC testcase using 100 runs of randomized sequences is given. We use a cost function like $w_1 \times area + w_2 \times wirelength$, where the weights w_1 and w_2 are for area and wire length respectively and two terms *area* and *wirelength* are normalized. The table 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. 6 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* case, we can find a better solution which has 4% improvement in area and 17% improvement in wire length. In average, we can get about 16% improvement in wire length while the chip area is comparable. In Fig. 7, the placement after improvement shows a better interconnection than the placement before improvement.

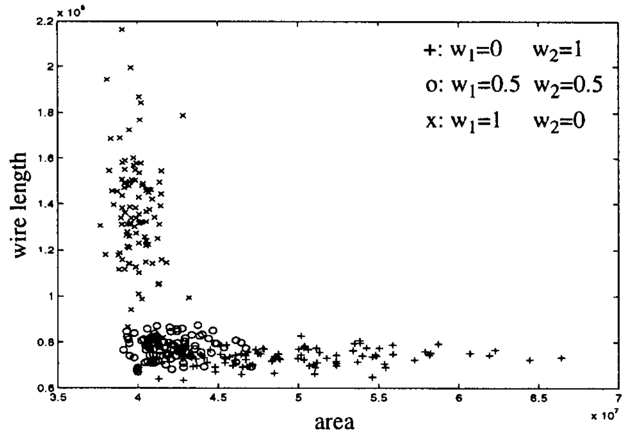


Figure 6: randomized sequence with different weights

7. Conclusion

We have successfully found a simpler layout representation and incorporated the capability for geometric compaction into the structure itself. O-tree, a simplified structure to present the geometric relation, is developed and proposed to replace the commonly used constraint graph which we find is complicated and has expensive operations. 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. Further studies on the properties of O-tree are undergoing. A varieties of approaches based on O-tree are under development, and there are some promising results already.

8. References

[1] K. Keeler and J. Westbrook, *Short Encoding of Planar Graphs and Maps*, Discrete Applied Mathematics, vol. 58, pp. 239-252, 1995

[2] D.E. Knuth, *The Art of Computer Programming*, 2nd Ed., Vol. 1, Addison-Wesley Publishing Co., pp. 385-395, 1973

[3] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajatani, *Rectangular-Packing-Based Module Placement*, ICCAD, pp. 472-479, 1995

[4] S. Nakatake, K. Fujiyoshi, H. Murata, and Y. Kajitani, *Module Placement on BSG-Structure and IC Layout Applications*, ICCAD, pp. 484-491, 1996

[5] H. Onodera, Y. Taniguchi, K. Tamaru, *Branch-and-Bound Placement for Building Block Layout*, DAC, pp. 433-439, 1991

[6] R. H. J. M. Otten, *Automatic Floorplan Design*, Proc. ACM/IEEE Design Automation Conf., pp. 261-267, 1982

[7] P. Pan and C.L. Liu, *Area Minimization for Floorplans*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and System, pp. 123-132, January 1995

[8] B. T. Preas and W. M. VanCleemput, *Placement Algorithms for Arbitrarily Shaped Blocks*, DAC, pp. 474-480, 1979

[9] T. Takahashi, *An Algorithm for Finding a Maximum-Weight Decreasing Sequence in a Permutation, Motivated by Rectangle Packing Problem*, IEICE, vol. VLD96, pp. 31-35, 1996

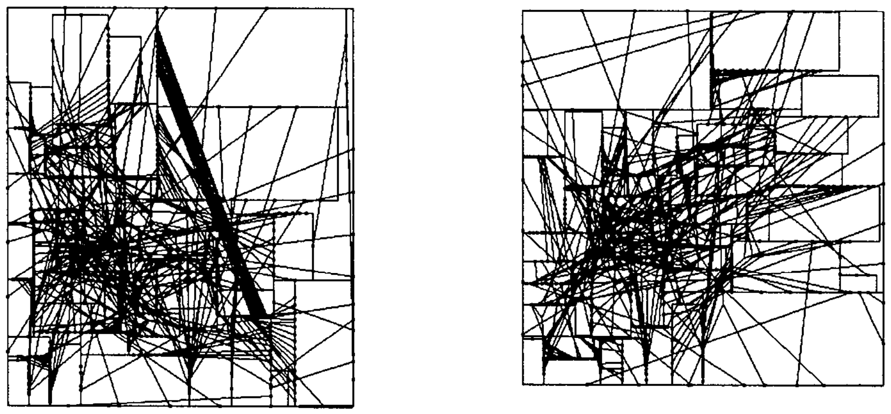
[10] T.-C. Wang, and D. F. Wong, *An Optimal Algorithm for Floorplan Area Optimization*, DAC, pp. 180-186, 1990

[11] D. F. Wong, and C. L. Liu, *A New Algorithm for Floorplan Design*, DAC, pp. 101-107, 1986

[12] J. Xu, P.-N. Guo, and C.-K. Cheng, *Cluster Refinement for Block Placement*, DAC, pp. 762-765, 1997

Table 2: Minimum / 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%



area = 40.8 (5.92 x 6.89) wire length = 810
(a) before improvement

area = 39.9 (6.17 x 6.47) wire length = 680
(b) after improvement

Figure 7: placements before and after deterministic improvement for *ami49*