Cluster Refinement for Block Placement

Jin Xu, Pei-Ning Guo, and Chung-Kuan Cheng

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093

Abstract

We propose an iterative optimization approach for mixed macro-cell and standard-cell placement, which minimizes the chip size and interconnection wire length at the same time. We present a branch-and-bound algorithm which efficiently searches for the optimal solution by evaluating all of the possible configurations on the selected cluster to minimize the gap distance between the ceiling and the floor. A virtual grid and permutation order are generated dynamically to eliminate redundant branches, which was the cause of much higher complexity in other approaches. Experimental results on the MCNC benchmark circuits show that the algorithm achieves very competitive results to manual design.

1 Introduction

Placement of blocks on a 2D surface is the first and the most critical process in VLSI layout design. The major objectives are chip area minimization and interconnection wire length minimization. Since the number of possible placements increases explosively with the number of blocks, even subsets of the problem have been shown to be NP-complete or NP-hard.

H. Murata et al.[4] introduces a P-admissible solution space of size $(n!)^2 8^n$, where n is the total number of blocks, and applies a simulated annealing method to search for a good solution. Clearly, the space is so large that there is no guarantee of finding an optimal solution in a reasonable amount of computation time.

Onodera et al.[5] presents a building block placement approach which employs a branch-and-bound strategy to search for an optimal solution within the whole solution space. However, the maximum number of blocks which can be placed in a reasonable amount of CPU time is around six, and larger problems must be decomposed, and placements are constructed hierarchically in a bottom-up manner.

Shin et al.[6] uses zone refinement technique in IC layout compaction. They first select a cluster of blocks from the ceiling, store the x- positions of the blocks as the best known solution, lower the blocks onto the floor to the leftmost positions, then find the block, one block each time, which is responsible for the minimum gap distance, search for a new candidate place for it. Lee[3] extends the zone refinement technique to arbitrarily shaped rectilinear and soft block floorplanning.

Note that the above approaches can be applied to general structures. For slicing structure, Yamanouchi et al.[8] proposes a partial clustering and module restructuring algorithm.

In this paper, we present a new iterative placement optimization approach. The major contribution of the work is a 2D branch-and-bound cluster refinement algorithm that searches for an optimal solution within the gap between the ceiling and the floor. It identifies the redundancy in the searching space trying out the same set of corners. A permutation tree is used to order the branch and bound sequence, and thus eliminate the redundancy dynamically.

To demonstrate the efficiency of the algorithm, we apply our algorithm to the MCNC benchmark circuits. Experimental results show that the algorithm achieves excellent area utilization while minimizing interconnection wire length at the same time.

2 Problem Description

Inputs of the placement problem are
- a set of blocks with fixed geometries and fixed pin positions
- a set of nets specifying the interconnections between pins of blocks
- a set of pads (external pins) with fixed positions
- a set of user constraints, e.g., block positions/orientations, critical nets, if any

Given the inputs, the objective of the problem is to:
find the positions and orientations of each block, so that the chip area and interconnection wire length between blocks are minimized while satisfying all the given constraints.

We take wire length into account simultaneously in the optimization process. Since it is impossible to calculate the exact wire length at this stage where detailed routing has not yet been carried out, we estimate the length of each net as one-half of the perimeter of the bounding box of the net.

The objective function, which measures the quality of the resulting placement, can be expressed as follows,

$$E = C_1 \times ChipArea + C_2 \times WireLength$$

where C_1, C_2 are the corresponding weights.

3 Some properties of rectilinear blocks for placement

We introduce here some properties of rectilinear convex blocks for placement. By convex, we mean that within the block any two points with the same x- coordinates (or y- coordinates) can be connected via a horizontal or a vertical line, which is also contained by the block. Therefore, for example, the block in Fig. 1(a) is convex, while the one in Fig. 1(b) is not.

(a) (b)

Fig. 1: Rectilinear convex. (a) convex; (b) concave.

A rectilinear convex block has four sets of edges viewed from the four corners of the block. The number of edges in the four sets gives the shape information of the block. Let n_{ne}, n_{se}, n_{sw}, n_{nw} be the number of edges in the four corners: northeast (horizontal edges), southeast (vertical edges), southwest (horizontal edges), northwest (vertical edges) respectively. A rectilinear convex block can be expressed as a 4-tuple (n_{ne}, n_{se}, n_{sw}, n_{nw}). Fig. 2(a) shows an example where n_{ne}=3, n_{se}=1, n_{sw}=2, n_{nw}=1.

When a block is rectangular, as a special case, the 4-tuple is simply (1, 1, 1, 1).

This research was funded in part by grants from the NSF project number MIP-9529077 and California MICRO Program.

DAC 97, Anaheim, California

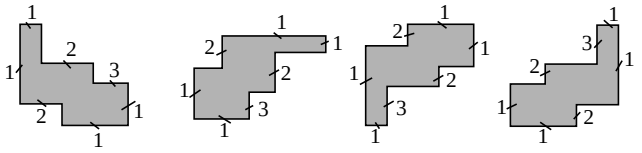

Fig. 2: Express rectilinear convex block as a 4-tuple. (a) original; (b) rotated 90 degree clockwise; (c) reflected up/down; (d) reflected left/right.

When a block is rotated 90 degree clockwise, the 4-tuple is also 'rotated' clockwise, becoming (n_{nw}, n_{ne}, n_{se}, n_{sw}); when it is reflected up/down along its x- axis or left/right along its y- axis, the 4-tuple is also 'reflected', becoming (n_{se}, n_{ne}, n_{nw}, n_{sw}) or (n_{nw}, n_{sw}, n_{se}, n_{ne}), as illustrated in Fig. 2(b), (c) and (d) respectively.

When two rectangular blocks *A*, *B* are placed adjacent to each other, there are four possible topological relationships between them([5]), i.e., *B* is above, right to, below, left to *A*. In the case of rectilinear convex blocks, the situation is much more complicated.

In general, when we are placing two stairlines adjacent to each other, one with *m* stairs, the other with *n* stairs, the total number of topological relationships between them is $m \times n$ in the worst case, as shown in Fig. 3, where the number is $3 \times 4 = 12$.

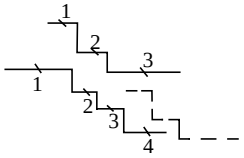

Fig. 3: Topological relationships between two stairlines

The following observation gives the general relationships between two rectilinear convex blocks in the worst case.

Observation: The total number of possible topological relationships between two rectilinear convex blocks *A*, *B* without rotation and reflection is

$$A_{ne} \times B_{sw} + A_{se} \times B_{nw} + A_{sw} \times B_{ne} + A_{nw} \times B_{se}$$

When both blocks are rectangular, as a special case, the total number is simply $1 \times 1 + 1 \times 1 + 1 \times 1 + 1 \times 1 = 4$.

If one block is rotated or reflected, then the corresponding number of possible additional relationships is simply the 'rotated' or 'reflected' 4-tuple of that block multiplied by the 4-tuple of the other block respectively.

4 Overall Algorithm

Zone refinement is a technique used in the purification process of crystal ingots. It provides a general framework for reducing the total number of blocks to the degree which can be handled at one time.

Fig. 4(a) shows the situation when cluster refinement is in progress. A placement consists of two regions of blocks, separated by a zone called a 'gap'. The lower bound of the top region forms the 'ceiling' profile, while the upper bound of the bottom region forms the 'floor' profile, as illustrated by the bold lines. The 'gap' profile is obtained by 'deducting' the ceiling from the floor profile.

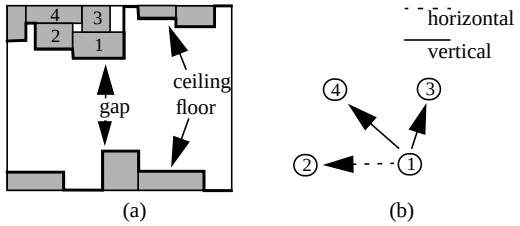

Fig. 4: Cluster refinement. (a) Cluster selected; (b) Constraint graphs

First a cluster of blocks is selected and 'peeled off' from the ceiling. Then the branch-and-bound algorithm is applied to the cluster, and the cluster is placed onto the floor based on the results obtained. The algorithm loops until the ceiling is empty.

```
ClusterRefinementPlacement {
        Select a compaction direction;
        for(n=0; n<IterationNumber; n++) {
              Construct ceiling;
              Initialize floor and gap;
              while(ceiling not empty) {
                    Select cluster;
                    Update ceiling;
                    Branch-and-Bound placement;
                    Update floor;}
              Alternate compaction direction;}}
```

Fig. 5: Outline of the algorithm

The figure above shows the outline of the algorithm. The algorithm improves an initial placement along one direction, alternates the optimizing direction and begins the next iteration. It may iterate many times until no improvement is achieved, a given number of iterations is reached or a computation time limit is passed.

4.1 Cluster Selection

The selection is based on compaction constraint graphs([6]). We first give the definition of neighbors, then present the algorithm.

Definition: Two blocks are *neighbors* if they are adjacent to each other in either horizontal or vertical constraint graph.

For example, in Fig. 4(a), the neighbors of block *1* are block *2*(left), block *3*(above) and block *4*(above), based on the constraint graphs shown in Fig. 4(b).

The algorithm selects a cluster adaptively and sequentially, according to the current partial placement. It identifies the block which dominates the minimum gap distance to be the 'critical' block, then build up a cluster of size *k* with this block and its *k-1* neighbors.

In Fig. 4(a), if *k*=4, block *1* is identified first and a cluster is built up with it and its three neighbors, blocks *2*, *3* and *4*.

4.2 Branch-and-Bound Placement

In order to find an optimal solution, we evaluate all of the possible combinations within the whole solution space on the selected cluster. Because the number of possible combinations increases greatly with the number of blocks examined, we employ a branch-and-bound technique to explore the solution space effectively.

4.3 Branching Operations

We enumerate the state space as a tree whose nodes correspond to partial placements for some blocks in the cluster. The successors of a node correspond to the implementation of the blocks to be considered next. A path from the root to a leaf represents a complete placement for the cluster.

4.3.1 order

Suppose *k* blocks are selected in the cluster and are numbered *1, 2,..., k*. To find the best order of blocks to be placed, we try all combinations of the blocks, i.e., *k!* permutations.

4.3.2 rotation

Each block may be specified in any one of eight orientations, i.e., two rotations and four reflections. We specify one rotation out of two for each block when placing each block, because rotation is closely related to the final chip area. Since reflection is only directly related to interconnection wire length, we try the four reflections for each block only when making estimates to reduce the total wire length.

4.3.3 virtual grids

When we search for the optimal position for a block in the cluster, we have to try all of the virtual grids created by the blocks

which have been placed onto the floor, and which are in the ceiling, as illustrated in Fig. 6.

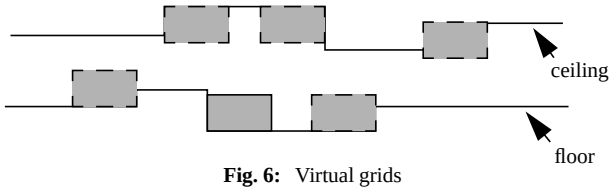

Fig. 6: Virtual grids

4.4 Bounding Operations

The efficiency of the branch-and-bound method depends a lot on the bounding technique. By pruning unpromising branches in the decision tree, we can explore the solution space much more efficiently to find the optimal solution.

4.4.1 recursive permutation tree

We can form the *k!* permutations as a recursive permutation tree. Therefore, when trying different permutations of the ith blocks, we only have to place the (i-1)th block once.

4.4.2 corners only

A permutation of *k* blocks can be expressed as

$$p(1), p(2),..., p(i),..., p(k)$$

where *p(i)* is the number of the ith block in the permutation. When we are placing two neighboring blocks p_1, p_2 in a permutation, we have to evaluate all of the virtual grids for the block p_1 first, then for the p_2. When they swap their order to be p_2, p_1 in another permutation, after the block p_2 has been placed in the floor, it creates some new grids. Then to place the block p_1, the only virtual grids we have to evaluate are those new grids, because they are the only grids that could not be evaluated by other permutations. Therefore, the following important theorem holds.

Theorem. When searching for the optimal position for the ith block in a permutation, we only have to evaluate the corners created by the (i-1)th block if

$$p(i) < p(i\text{-}1)$$

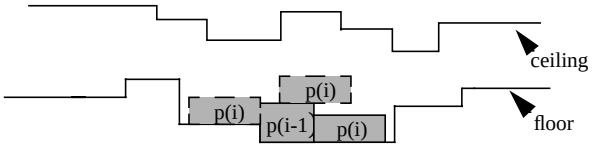

Fig. 7: Corners only

In general, we have to search for all of the virtual grids to place a block. The theorem above suggests that we only need to evaluate those virtual grids in the range from the leftmost to the rightmost position of the previous block(Fig. 7). This greatly reduces the search space and let the algorithm prune unpromising branches and reach the optimal solution much more efficiently.

4.4.3 lower bounds

We first place each block in the cluster to its original x- position, and calculate the cost functions and record their values along with the x- positions as the current best known solution. Those values along with the constraints given, provide lower bounds for the placements obtained in the search process later. The cost functions used could be some or all of the following: gap distance, chip width, chip length, aspect ratio, dead space created, interconnection wire length, etc..

Each node in the decision tree corresponds to a partial problem in which only rotations or positions of some blocks in the cluster are determined, and only permutation among those blocks is established, while those of the others have not been yet. Associated with the nodes, are the partial cost values of the corresponding placement. If any one of them exceeds the corresponding lower bound, we can say that the branch is not promising, the search process along the branch will be terminated.

When the search process along a branch reaches a leaf node, a complete placement for the cluster is obtained. If the cost values of the placement are better than the current best known solution, we update the values and save the current x- positions as the best known solution.

4.5 Rectilinear block placement

In the case of rectilinear blocks, we also have to try all the different orders of blocks, and also the rotations and all the virtual grids for each block. And since reflection can result in different placements now, we also have to try different reflections for each block.

In general, the floor profile might not be overall 'convex'. We can divide the floor profile to several segments, so that each segment is 'convex'(Fig. 8). Then we can apply the rules of relationships between rectilinear convex blocks to place rectilinear blocks.

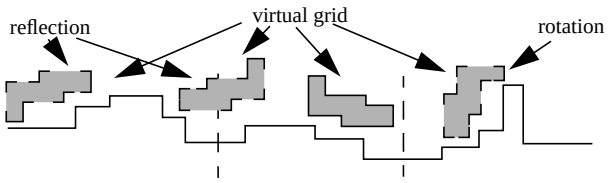

Fig. 8: Dividing floor profile to 'convex' segments

5 Complexity Analysis

We will roughly investigate the complexity of our algorithm which is adopted from the well-known analysis of zone-refinement algorithm. First we consider the complexity of constraint graphs and pure Z-R algorithm, and then analyze our extension parts for clustering and branch-and-bound algorithm.

5.1 Complexity of the constraint graphs and Z-R algorithm

Traditional Z-R algorithm deals with *n* blocks and utilizes two necessary data structures: constraint graphs and ceiling-floor-gap relations. These structures are maintained throughout the program and their updations are the key operations of entire algorithm.

Shin et al.[6] gave a detail proof that the initial construction of structures takes $O(n^2)$ and the updation needs $O(n)$ for each block moving. For a complete pass of Z-R, it needs $O(n)$ updation and results in an overall complexity $O(n^2)$.

5.2 Complexity of cluster refinement algorithm

In Section 4. 3, we assume *k* blocks per cluster and *m* virtual grids for every cluster moving. The bounding operations give a great amount of pruning from the original solution space $O(k!m^k)$. The following is the complexity analysis of the cluster refinement algorithm.

Lemma 1: Given a permutation $(p_1, p_2, \ldots, p_k)$, the branches at level *i* is $b_i = \begin{cases} m, \text{ if } i = 1 \vee p_{i-1} < p_i \\ 2, otherwise \end{cases}$

The number of branches in each node is either *m* for new pattern of permutation, or 2 when it is corner-only because of some old permutation covering most locations of virtual grids already.

Lemma 2: Given a permutation $(p_1, p_2, \ldots, p_k)$, the total number of branches is $B(p_1, p_2, \ldots, p_k) = \prod_{i=1}^{k} b_i$.

Lemma 3: Given a cluster $(1, 2, \ldots, k)$, the total number of branches in its decision tree is

$$\sum_{(p_1, p_2, \ldots, p_k) \in permutation} B(p_1, p_2, \ldots, p_k) .$$

For most cases, the useful size k of cluster is equal to or less than 5 and m is about $O(\sqrt{n})$. Table 1 gives the complexity for k less or equal to 5.

Table 1: Cluster size and complexity

k	number of branches
1	m
2	$m^2 + 2 \bullet m$
3	$m^3 + 4 \bullet 2 \bullet m^2 + 2^2 \bullet m$
4	$m^4 + 11 \bullet 2 \bullet m^3 + 11 \bullet 2^2 \bullet m^2 + 2^3 \bullet m$
5	$m^5 + 26 \bullet 2 \bullet m^4 + 66 \bullet 2^2 \bullet m^3 + 26 \bullet 2^3 \bullet m^2 + 2^4 \bullet m$

Thus, the overall complexity for cluster refinement algorithm will be $O(2^k m^k n^2)$. And for small k, it becomes $O(n^{2+k/2})$.

5.3 Comparison to other approaches

Cluster refinement takes the advantages of the Z-R algorithm and utilizes exhaustive search in local area that improves Z-R algorithm's weak point. Extra branch-and-bound searching of best placement for clusters increases the complexity by the factor of $O(2^k m^k)$ to the original Z-R algorithm's $O(n^2)$.

Other approaches like Onodera's topological relationship[5] taking $O(8^n n!)$ to find the optimal solution which can only handle six blocks at most mentioned by the author. Another approach, Murata's sequence-pair[4] has solution space in the size of $O(8^n (n!)^2)$ and using the simulated annealing method to search only a small fraction of the whole space. Fig. 9 shows the comparison with other approaches.

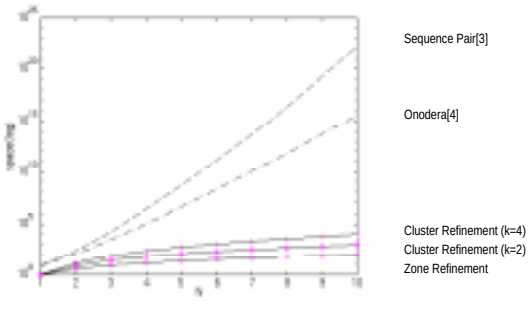

Fig. 9: Comparisons

6 Experiments

To examine the efficiency of the proposed algorithm, we apply our algorithm to the MCNC benchmark circuits. The algorithm is implemented in C and executed on a Sun Sparc20 workstation.

Table 2 shows the final results when four blocks are selected in a cluster. Fig. 10 shows the final placements for the two biggest benchmark circuits *ami33* and *ami49*.

Table 2: Experimental Results on MCNC Benchmark Circuits

circuit	*apte*	*xerox*	*hp*	*ami33*	*ami49*
area(mm^2)	48.42	20.30	9.575	1.207	37.69
dead space(%)	3.83	4.69	7.77	4.15	5.95
wire length(mm)	321	477	185	64	764
CPU (sec)	223.8	18.8	18.0	603.4	1861.7

Table 3: CPU time required when using different cluster sizes

cluster size	1	2	3	4	5
ami33	11.8	25.4	109.9	893.7	7931.9
ami49	31.7	60.5	219.1	2141.2	41298.2

Table 4: Placements on *ami49* using different cluster sizes

cluster size	1	2	3	4	5
area (mm^2)	39.51	38.99	38.38	37.69	37.26
dead space (%)	10.29	9.08	7.64	5.95	4.87

The algorithm can employ different cluster sizes. Obviously, the more blocks are selected in the cluster, the more CPU time the algorithm will take, but it is also expected that, the better results the algorithm will achieve, because the branch-and-bound algorithm will search for the larger solution space. Table 3 shows the CPU time, when different cluster sizes are used, to iterate the algorithm 100 times on the two biggest benchmark circuits. Table 4 shows the results by using different cluster sizes on benchmark circuit *ami49*.

It is difficult to fairly compare our algorithm with the other approaches reported, because they include routing space in their placements, which is not necessary any more because of recent technology progress. However, it is estimated that the chip area and wire length would increase around 10% and 5% respectively, if introducing routing space by the technology factor T they used([4], [5]). So our approach still outperforms others a lot.

References

[1] C.K. Cheng, E.S. Kuh, "Module Placement based on Resistive Network Optimization", *IEEE Trans. Computer-Aided Design*, vol. CAD-3, pp. 218-225, July 1984.

[2] W. M. Dai, E. S. Kuh, "Simultaneous Floor Planning and Global Routing for Hierarchical Building-Block Layout", *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 828-837, Sept. 1987.

[3] T. C. Lee, "A Bounded 2D Contour Searching Algorithm for Floorplan Design with Arbitrarily Shaped Rectilinear and Soft Modules", *Proc. 30th Design Automation Conf.*, pp.525-530, 1993.

[4] H. Murata, K. Fujiyoshi, S. Nakatake, Y. Kajitani, "Rectangular-Packing-Based Module Placement", *Proc. IEEE International Conf. on Computer-Aided Design*, pp. 472-479, 1995.

[5] H. Onodera, Y. Taniguchi, K. Tamaru, "Branch-and-Bound Placement for Building Block Layout", *Proc. 28th Design Automation Conf.*, pp. 433-439, 1991.

[6] H. Shin, A. L. Sangiovanni-Vincentelli, C. H. Sequin, "'Zone-Refining' Techniques for IC Layout Compaction", *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 167-178, Feb. 1990.

[7] D. F. Wong, C. L. Liu, "A New Algorithm for Floorplan Design", *Proc. 23rd Design Automation Conf.*, pp. 101-107, 1986.

[8] T. Yamanouchi, K. Tamakashi, T. Kambe, "Hybrid Floorplanning Based on Partial Clustering and Module Restructuring", *Proc. IEEE International Conf. on Computer-Aided Design*, pp. 478-483, 1996.

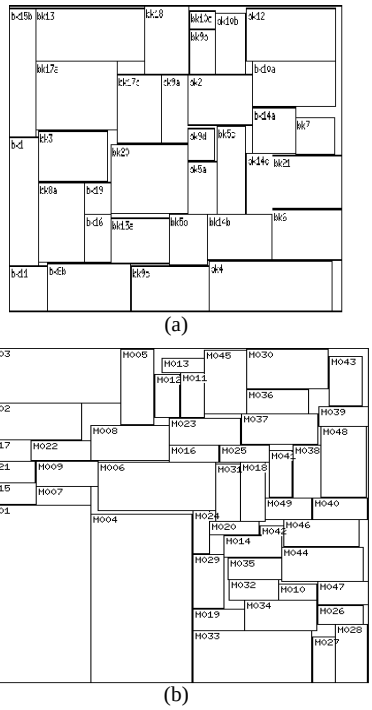

Fig. 10: Final placements. (a) *ami33*; (b) *ami49*