Floorplanning with Soft Rectilinear Blocks Using Corner Block List*

Yijie Zeng[1,2], Sheqin Dong[2], Xianlong Hong[2]

[1]Graduate School at Shenzhen, Tsinghua University, Shenzhen 518057, China
[2]Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China

Abstract

*In this paper, we propose a CBL-based floorplanning algorithm to handle arbitrary soft rectilinear blocks. The rectilinear blocks are efficiently handled by an **obstacle-based** method and a **replace-upon-overlap** mechanism. Compared with previous methods, our algorithm has two major advantages. 1) **Efficiency**: it can obtain a fast convergence to a sub-optimal solution when combined with simulated annealing; and 2) **Generality**: it can efficiently handle soft rectilinear blocks as well as hard rectilinear blocks. We have tested our algorithm on several modified **MCNC** benchmarks, and the experimental results show that our approach is quite promising.*

1. Introduction

With the advent of deep submicro technology, the sizes of **VLSI** circuits are growing dramatically. To reduce the design complexity, new design schemes such as hierarchical design with **IP** blocks and Mixed Mode Placement (**MMP**) are widely used. In these schemes, the circuit components are not always rectangle, giving rise to the importance of floorplanning/placement with rectilinear blocks.

Several methods have been proposed to handle rectilinear blocks based on well-known representations such as **SP** [1-3], **BSG** [4-5], **B***-tree [6], **O**-tree [7], **TCG** [8], and **CBL** [9-10]. Most of these methods partition each rectilinear block into a set of rectangular subblocks, and handle the subblocks individually under some constraints derived from their relative positions in the original rectilinear block. To obtain a feasible placement, all the constraints must be satisfied and postprocessing is needed to restore the original shapes of the rectilinear blocks. These methods have two major drawbacks. Firstly, diminishing violation of the constraints during the optimization process is quite time-consuming. Secondly, they can only handle hard rectilinear blocks of fixed shapes. However, some **IP** blocks are "***soft***" in that each of them has several alternative shapes to achieve flexibility. In **MMP**, the standard cells are grouped into a set of macro blocks. To obtain a more compact placement, it is desirable that each macro block has several candidate shapes rather than only one fixed shape.

To address the floorplanning problem with soft rectilinear blocks, we propose a **CBL**-based algorithm. All the subblocks of each rectilinear block are placed as ***obstacles*** at their right relative positions, and a ***replace-upon-overlap*** mechanism is used to avoid any potential overlaps among the blocks. Thus, our algorithm always constructs a feasible floorplan for any given CBL list. Furthermore, we can efficiently handle soft rectilinear blocks by integrating the selection of their shapes into the ***move set*** of the simulated annealing.

The rest of the paper is organized as follows: Section 2 gives a brief review of the **CBL** representation. In Section 3 we discuss the obstacle-based method. We present our floorplanning algorithm in Section 4, and then show the experimental results in Section 5. Finally, some concluding remarks are given in Section 6.

2. Corner Block List Representation

CBL represents a special type of floorplan called mosaic floorplan [11]. In mosaic floorplan, the whole chip area is divided into a set of rectangular rooms, each of which is assigned to exactly one block. The **Corner Block (CB)** is the block at upper-right corner of the floorplan. The left and bottom bounding segments of CB form a T-junction.

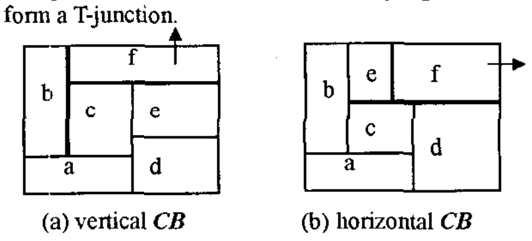


Fig. 1. The orientation of the corner block (CB)

The orientation of CB is defined by the orientation of its T-junction. The T-junction has only two types of orientations: **T** rotated by 90 degrees (Fig. 1(a)) and by 180 degrees (Fig. 1(b)) counterclockwise respectively. If **T** is rotated 90 degrees counterclockwise, the **CB** is vertically oriented, and its corresponding entry in list **L** is set by a "0". Otherwise, the **CB** is horizontally oriented, and the entry in list **L** is set by a "1".

CBL list is constructed by recursive deletion of **CBs** in the floorplan. For each deletion of a **CB**, its name is recorded in list S, its orientation recorded in list **L**, and the number of its attached T-junctions is recorded by the same number of successive "1"s ended by a "0" in a

* This work is supported by National Natural Science Foundation of China (No. 60076016), NSFC and Hongkong RGC Joint Project (No. 60218004), and National Promotion Project (Tsinghua) (No. Jc2001025).

binary list T_i (Fig. 2). At the end of deletions of all **CBs**, three lists are obtained: the block name list **S** = $\{M_n, M_{n-1}, \ldots, M_1\}$, the orientation list **L** = $\{L_n, L_{n-1} \ldots L_2\}$, and the T-junction list **T** = $\{T_n, T_{n-1}, \ldots, T_2\}$. Then each list is reversed and all the items of the T-junction list are combined into a single binary vector **T**.

The triple (S, L, T) is called a ***corner block list***.

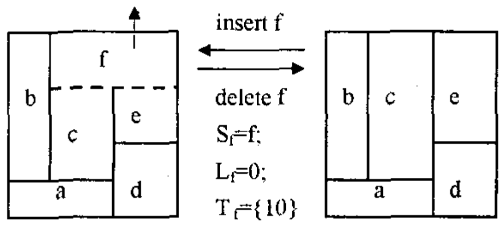
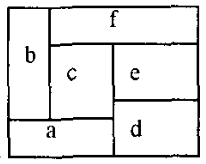


Fig. 2. Deletion/insertion of the corner block f

Construction of the floorplan from a **CBL** is the inverse process. Blocks are inserted in turn either from the right for vertical orientation, or from the top for horizontal orientation, covering required number of T-junctions given by the corresponding entry in list **T**. Fig. 3 illustrates the resultant floorplan of a corner block list.



Corner Block List:
S={a,b,c,d,e,f};
L={0,1,1,0,0};
T={0 10 0 0 10}.

Fig. 3. A CBL list and the resultant floorplan

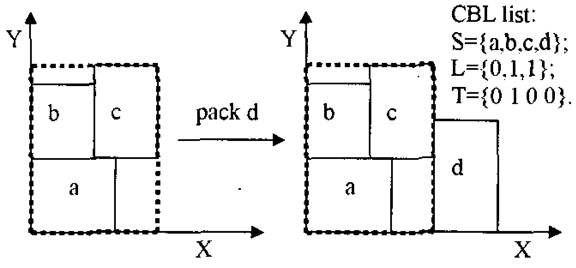
3. The Obstacle-Based Method

3.1 Preliminaries

Definition 1: The **Shielding Region** of a packed corner block M_i is the two-dimensional region defined by $[0, x^R(M_i)] \times [0, y^R(M_i)]$, where $(x^R(M_i), y^R(M_i))$ is the coordinate of the upper right corner of M_i.

According to construction of the floorplan from a **CBL** list, once a corner block has been packed, any successive corner blocks can only be packed above it or at the right of it. Thus the following lemma holds.

Lemma 1: Any successive blocks cannot be packed inside the shielding regions of previous corner blocks.



(a) before *d* is packed (b) after *d* is packed

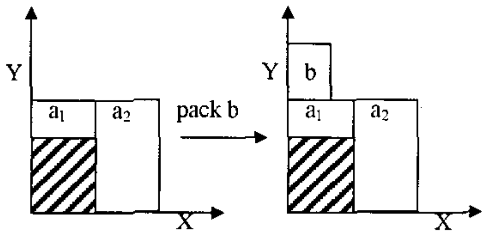
Fig. 4. The shielding region of the corner block *c* is enclosed by the dotted segments.

As illustrated in Fig. 4, after the third corner block *c* is packed, the successive block *d* can only be packed outside the shielding region of *c*.

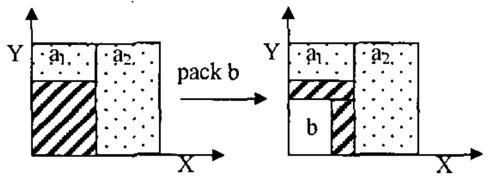
3.2 Obstacles

In our approach, each rectilinear block is partitioned into a set of rectangle subblocks. Once the location of a rectilinear block is determined, the locations of all its subblocks are also fixed. However, if these subblocks are packed as corner blocks, then the empty space within the enclosing rectangle of the rectilinear block cannot be utilized. As shown in Fig. 5(a), once the two subblocks a_1 and a_2 of an L-shaped block are packed as corner blocks in turn, the empty space beneath a_1 cannot be occupied by successive blocks according to Lemma 1.

Therefore, we do not pack the subblocks as corner blocks. Instead, to obtain a more compact floorplan, we place them as ***obstacles*** for successive blocks. The obstacles are maintained by a list called ***obstacle list***, which has one corresponding entry for each obstacle. If the two subblocks are placed as obstacles, then the empty space beneath a_1 can be partially utilized by a rectangle block ***b***, resulting in a better use of the chip area (Fig. 5(b)).



(a) The empty space cannot be utilized



(b) The empty space is partly occupied by **b**

Fig. 5.Obstacles.The darkly and slightly shaded areas denote empty space and obstacles respectively

3.3 Avoiding Overlaps

Once the subblocks are settled down in the chip area as obstacles, obviously they should not be overlapped by any successive blocks. Such a non-overlapping requirement is guaranteed by the ***replace-upon-overlap*** mechanism. In particular, each time before we settle down a new corner block, we first compute its location according to the **CBL** packing, and then ***try placing*** it. If the new corner block overlaps any current obstacles, an overlapping obstacle is selected, and a corresponding shield block is packed in place of the corner block. In that case, we need to re-compute the location of the current corner block.

Definition 2: The **Shield Block** of an obstacle is a false rectangle block of variable dimensions, whose function is to shield the obstacle from being overlapped by successive blocks.

When we pack the shield block, we determine its dimensions such that its upper right corner coincides with that of its corresponding obstacle.

Lemma 2: After the shield block is packed as a corner block, the corresponding obstacle will never be overlapped by any successive blocks.

Proof: According to the definition of ***shielding region***, the obstacle locates inside the shielding region of its corresponding shield block. Besides, Lemma 1 guarantees that any successive blocks cannot be packed into the shield region of the shield block. Therefore, the obstacle will never be overlapped by any successive blocks.

Suppose a_1 and a_2 are two obstacles, and block b is higher than the empty space beneath block a_1. When we compute the location of block b and try placing it, it overlaps one of the two obstacles a_1 (Fig. 6(a)). Thus, the corresponding shield block a^S_1 is packed as a new corner block in place of b, and the upper right corners of both a^S_1 and a_1 coincide precisely (Fig. 6(b)). The boundaries of the shield block a^S_1 are shown with the thick black segments in Fig. 6(b). It can be seen that a_1 is shielded by its shield block a^S_1.

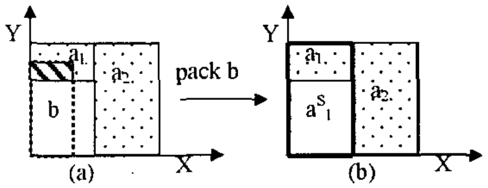


Fig. 6. The shield block a^S_1 is packed in place of block b. The darkly shaded area denotes the overlapping area.

According to Lemma 2, after its corresponding shield block is packed, the obstacle will not be overlapped by any successive blocks, so we update the obstacle list by deleting the corresponding entry in it.

3.3 Resolving Multiple Overlaps

If the new corner block overlaps more than one obstacle, we select one particular overlapping obstacle according to the following criterion, and pack the corresponding shield block.

Criterion of Selection: if the corner block is vertically oriented, then we select the overlapping obstacle with the ***lowest*** upper boundary; otherwise, we select the one with the ***leftmost*** right boundary.

The criterion is designed in a straightforward way. Intuitively, selecting the overlapping obstacle and thus packing the corresponding shield block with the lowest (leftmost) upper (right) boundary will result in a more compact floorplan.

4. The Floorplanning Algorithm

In our approach, all the blocks are represented in the list S in a uniform way. Each block has exactly one corresponding entry in list S, regardless of its geometric shape. For soft rectilinear blocks, the selection of their shapes is integrated in the ***move set*** of the simulated annealing. As shown in Fig. 7, three blocks are to be placed. The resultant list S consists of only three entries (Fig. 7(d)), even though both block a and block c are partitioned into more than one subblock.

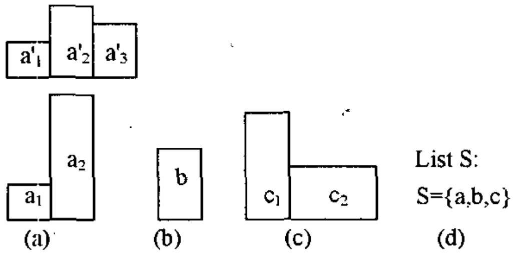


Fig. 7. Three blocks and the resultant list S.
(a) Soft rectilinear block a; (b) Rectangle block b;
(c) Hard rectilinear block c; (d) Resultant list S.

4.1 Outline of the Algorithm

Given a **CBL** list, we construct the floorplan by processing the blocks in the list S in turn. Based on the construction of floorplan from a **CBL** list, we compute the location for each corner block. For a rectilinear block, its location is represented in terms of the loaction of its enclosing rectangle. Then we ***try placing*** the corner block. If it does not overlap any obstacles, we pack it in case of a rectangle block, or settle down all its subblocks as obstacles if it is a rectilinear block. Otherwise it overlaps some obstacles, then one particular overlapping obstacle is selected according to the ***criterion of selection***, and its corresponding shield block is packed. In that case, we need to re-compute the location of the current corner block and try placing it again. The algorithm is outlined as the following:

```
ALGORITHM Floorplan
Input: CBL list and partitions of rectilinear blocks.
Output: locations for all blocks.
BEGIN
    Initialize the obstacle list empty, and set $i$ to 1;
    WHILE $i$ is not greater than $n$ DO
        Compute the location of $M_i$ and try placing it;
        IF $M_i$ overlaps any current obstacles THEN
            Select one overlapping obstacle
            according to the criterion of selection;
            Pack its shield block in place of $M_i$;
            Update the obstacle list;
        END IF;
        ELSE
            Pack $M_i$ if it is a rectangle block;
            Otherwise place all its subblocks as
            obstacles, and update the obstacle list;
            Increase $i$ by 1;
    END WHILE;
END ALGORITHM;
```

4.2 Handling Soft Rectilinear Blocks

The floorplanning algorithm is embedded in the inner loop of simulated annealing. To handle soft rectilinear blocks, a new category of ***move*** is integrated into the original ***move set*** suggested by [11]. In particular, we randomly select a soft rectilinear block, and randomly select an alternative shape to substitute the original one. By this means, we can efficiently deal with any soft rectilinear blocks.

5. Experimental Results

We have implemented our algorithm using **C** programming language, and have performed several modified **MCNC** benchmarks on the **SUN** Sparc20 workstation. All the rectangle blocks are flexible in aspect ratio distributing between 0.2 and 5 discretely, and area is the objective of the optimization. The same annealing schedule was used for all the experiments.

To test the efficiency of our algorithm to deal with hard rectilinear blocks, we expanded several blocks in the benchmarks into L-/T-shaped block. For each benchmark, we randomly generated 100 different cases with the same number of L-/T-shaped blocks. The average data are listed in Tab. 1. The second column in Tab. 1 is the number of rectangle blocks. We can see that our algorithm is quite fast in obtaining good solutions.

In addition, to test its capability to handle soft rectilinear blocks, we expanded several blocks into soft rectilinear blocks. Each soft rectilinear block has four alternative rectilinear shapes: L-, T-, Z-, and |_|-shaped respectively. Similarly, 100 randomly generated cases were tested for each benchmark. The average data are shown in Tab. 2, and the resultant floorplan of a randomly generated case is illustrated in Fig. 8. The third column in Tab. 2 is the number of the soft rectilinear blocks. The results demonstrate that our algorithm can efficiently handle soft rectilinear blocks.

6. Conclusion

In this paper, we have proposed an efficient floorplanning algorithm to handle arbitrary soft rectilinear blocks. The rectilinear blocks are handled by an ***obstacle-based*** method and a ***replace-upon-overlap*** mechanism. We handle soft rectilinear blocks by integrating the selection of their shapes into the ***move set*** of simulated annealing. The experimental results demonstrate that our algorithm can efficiently deal with both hard and soft rectilinear blocks.

References:

[1] J. Xu, P. -N. Guo, and C. -K. Cheng, "Rectilinear Block Placement Using Sequence-Pair, "IEEE TCAD, pp. 484–493, 1999.

[2] M. Kang, and W. W. Dai. , "Arbitrary Rectilinear Block Packing Based on Sequence Pair, " Proc. ICCAD, pp. 259–266, 1998.

[3] K. Fujiyoshi and H. Murata, "Arbitrary Convex and Concave rectilinear Block Packing Using Sequence-Pair, " IEEE TCAD, vol. 19, no. 2, pp. 224–233, Feb.2000.

[4] M. Kang and W. Dai. , "General Floorplanning with L-shaped, T-shaped and Soft Blocks Based on Bounded Slicing Grid Structure, " Proc. ASP-DAC, pp. 265–270, 1997.

[5] S. Nakatake, M. Furuya, and Y. Kajitani, "Module Placement on BSG-Structure with Pre-Placed Modules and Rectilinear Modules, " Proc. ASP-DAC, pp. 571–576, 1998.

[6] G. -M. Wu, Y. -C. Chang, and Y. -W. Chang, "Rectilinear Block Placement Using B*-Trees, " Proc. ICCD, 2000.

[7] Y. Pang, C. K. Cheng, K. Lampasert, and W. Xie, "Rectilinear Block Packing Using O-tree Representation, " Proc. ISPD, pp. 156-161, 2001.

[8] J. Lin, H. Chen, and Y. Chang "Arbitrary convex and concave rectilinear module packing using TCG" Proc. DATE , pp. 69-75, 2002

[9] Y. Ma, X. Hong, S. Dong, Y. Cai, C. -K. Cheng, and Jun Gu, "Floorplanning with Abutment Constraints and L-shaped/T-shaped Blocks based on Corner Block List" Proc. DAC, pp. 770-775, 2001.

[10] —, "Stairway compaction using corner block list and its applications with rectilinear blocks"Proc. ASP-DAC, pp. 387-392, 2002

[11] X. Hong, G. Huang et al. "Corner Block List: An Effective and Efficient Topological Representation of Nonslicing Floorplan" Proc. ICCAD, pp. 8-12, 2000.

Tab. 1. Floorplanning results with L-shaped and T-shaped blocks

examples	#|M|	#|L|	#|T|	dead space(%)	CPU time(sec.)
apte_lt	5	3	1	4.9	13
xerox_lt	6	3	1	6.4	13
hp_lt	7	3	1	5.6	13
ami33_lt	23	8	2	6.3	46
ami49_lt	35	11	3	9.3	63

Tab. 2. Floorplanning results with soft rectilinear blocks

examples	#|M|	#|R|	dead space(%)	CPU time(sec.)
apte_sr	5	4	3.7	13
xerox_sr	6	4	5.4	13
hp_sr	7	4	4.4	13
ami33_sr	23	10	5.4	46
ami49_sr	35	14	7.9	63

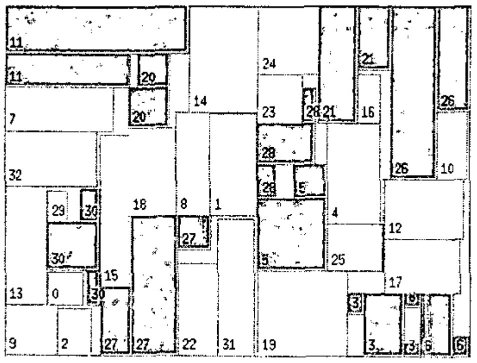


Fig. 8. A resultant floorplan of ami33 with 10 rectilinear blocks: the area usage is 94.3% and the CPU time is 46 seconds.