Rectilinear Block Placement Using B*-Trees *

Guang-Ming Wu, Yun-Chih Chang, and Yao-Wen Chang
Department of Computer and Information Science, National Chiao Tung University, Hsinchu 300, Taiwan

Abstract

Due to the layout complexity in deep sub-micron technology, integrated circuit blocks are often not rectangular. However, literature on general rectilinear block placement is still quite limited. In this paper, we present approaches for handling the placement for arbitrarily shaped rectilinear blocks, based on a newly developed data structure called *B*-trees* [1]. Experimental results show that our algorithm achieves optimal or near optimal block placement for benchmarks with multiple shaped blocks.

1 Introduction

Due to the growth in design complexity, circuit size is getting larger. To cope with the increasing design complexity, hierarchical design and IP modules are widely used. This trend makes block floorplanning/placement much more critical to the quality of a design.

Floorplans can be divided into two categories, the *slicing structure* [13, 17] and the *non-slicing structure* [1, 2, 8, 11, 16]. A slicing structure can be represented by a binary tree whose leaves denote modules, and internal nodes specify horizontal or vertical cut lines. Wong and Liu proposed an algorithm for slicing floorplan design [17]. They presented a normalized Polish expression to represent a slicing structure, enabling the speed-up of the search procedure. However, this representation cannot handle non-slicing floorplans. Recently, researchers have proposed several representations for non-slicing floorplans, such as sequence pair [8], bounded slicing grid (BSG) [11], O-tree [2], and B*-tree [1].

In deep sub-micron technology, the blocks are often not rectangular. Most existing floorplanning/placement algorithms only deal with rectangles and cannot apply to arbitrarily shaped rectilinear block placement directly. New approaches which can handle arbitrary shaped blocks are essential to optimize area utilization.

Preas et al. in [14] proposed a graph model for the topological relationship among rectangular and arbitrarily shaped rectilinear blocks. Wong and Liu in [18] extended the Polish expression to represent slicing floorplans with rectangular and L-shaped blocks. Lee in [7] extended the zone refinement technique to rectilinear blocks. A bounded 2D contour searching algorithm is proposed to find the best position for a block.

Kang and Dai in [4] proposed a BSG-based method to solve the packing of rectangular, L-shaped, T-shaped, and soft blocks. The algorithm combines simulated annealing and a genetic algorithm for general non-slicing floorplans.

Xu, Guo, and Cheng in [19] presented an approach extending the sequence-pair approach for rectangular block placement to arbitrarily sized and shaped rectilinear blocks. The properties of L-shaped blocks are examined first, and then arbitrarily shaped rectilinear blocks are decomposed into a set of L-shaped blocks.

Kang and Dai in [5] proposed a method based on the sequence-pair structure for the rectilinear block placement. Three necessary and sufficient conditions for a sequence pair to be feasible are derived. A stochastic search is applied on the optimization of convex block floorplanning.

Chang et al. recently proposed the *B*-tree* representation for non-slicing floorplans in [1], which is based on block compaction and ordered binary trees. Inheriting from the nice properties of ordered binary trees, B*-trees are very easy for implementation and require only constant time for tree search and insertion, and linear time for deletion. Unlike the sequence pair, BSG, and O-tree representations, in particular, no extra encoding (except the tree itself) is needed for a B*-tree, and cost evaluation can be performed on a B*-tree directly. Besides, the ordered property of a B*-tree makes the incremental cost evaluation of its corresponding placement possible. Further, given a B*-tree, it takes only linear time to construct the placement, and vice versa. All these nice properties make the B*-trees an efficient and flexible representation for non-slicing floorplans. Empirical results show that the B*-tree representation is about 4.5 times faster and consumes about 60% less memory than the O-tree one [1].

In this paper, we extend the B*-tree approach to arbitrarily shaped rectilinear blocks. First, we explore the properties of L-shaped blocks and then extend the properties to general rectilinear blocks. We construct a set of benchmarks with rectangular and L-shaped (and T-shaped) blocks and apply simulated annealing as a vehicle to test the effectiveness of our approaches. Experiment results show that our approaches lead to placements with optimal or near optimal area utilization.

The remainder of this paper is organized as follows. Section 2 formulates the rectilinear block placement problem. Section 3 introduces the B*-tree representation. Section 4 describes the method for the L-shaped blocks in a B*-tree. Section 5 describes our algorithm. Section 6 extends the algorithm to arbitrary rectilinear blocks. Experimental results are reported in Section 7. Finally, we give conclusions in Section 8.

2 Formulation

Let $B = \{b_1, b_2, \ldots, b_n\}$ denote a set of rectilinear blocks. A block is not flexible in its shape but free to rotate and flip. A packing of a set of blocks is a non-overlapping placement of the blocks.

A rectilinear block can be represented by four profile sequences, namely *the top profile sequence, the bottom profile sequence, the left profile sequence*, and *the right profile sequence*, specifying the profiles viewed from the top side, the bottom side, the left side, and the right side of the block, respectively. The top (bottom) profile sequence of a rectilinear block uses the leftmost horizontal segment on the top (bottom) boundary of the block as a *base* and records the length of the succeeding horizontal segments on the top (bottom) boundary and the relative height. Specifically, the top profile sequence that is composed of the length of the base, followed by a sequence of two-tuples composed of the lengths of the succeeding horizontal segments and their relative heights to the base (could be negative). For example, Figure 1 shows a rectilinear block with the top profile sequence (4, [5, 7], [7, 4], [6, -1], [8, 4]). The base of the sequence is segment a which has the length of 4 units. The second horizontal segment is c which has the length of 5 units and is 7 units higher than the base a. Similarly, the third horizontal segment is e which has the length of 7 units and is 4 units higher than the base a, and so on. The other three profile sequences are similarly defined.

Definition 1 *A rectilinear block placement is* feasible *if and only if no two blocks overlap with each other, and all profile sequences remain unchanged after placement (i.e., all blocks keep their original shapes).*

The goal of the rectilinear placement problem is to minimize the area

*This work was partially supported by the National Science Council of Taiwan under Grant No. NSC-89-2215-E-009-055. E-mail: {gis85815, gis87512, ywchang}@cis.nctu.edu.tw.

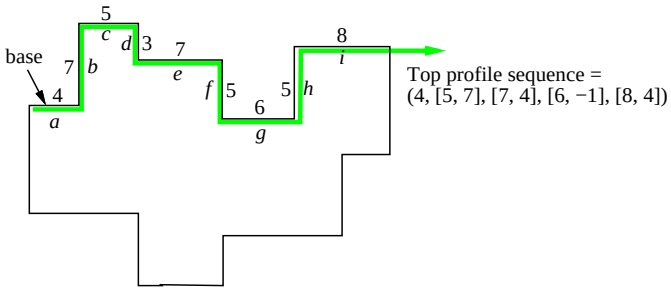


Figure 1: The top profile sequence consists of the length of the base, followed by a sequence of two-tuples that is composed of the lengths of the succeeding horizontal segments and their relative heights to the base.

induced by the assignment of b_i's, where area is measured by the final enclosing rectangle of B.

3 Overview of the B*-Tree Representation

Given an admissible placement P, we can represent it by a unique (horizontal) B*-tree [1] T. (See Figure 2(b) for the B*-tree representing the placement shown in Figure 2(a).) A B*-tree is an ordered binary tree whose root corresponds to the module on the bottom-left corner. Similar to the DFS procedure, we construct the B*-tree T for an admissible placement P in a recursive fashion: Starting from the root, we first recursively construct the left subtree and then the right subtree. Let R_i denote the set of modules located on the right-hand side and adjacent to b_i. The left child of the node n_i corresponds to the lowest module in R_i that is unvisited. The right child of n_i represents the module located above and visible from b_i, with its x-coordinate equal to that of b_i and its y-coordinate less than that of the top boundary of the module on the left-hand side and adjacent to b_i, if any.

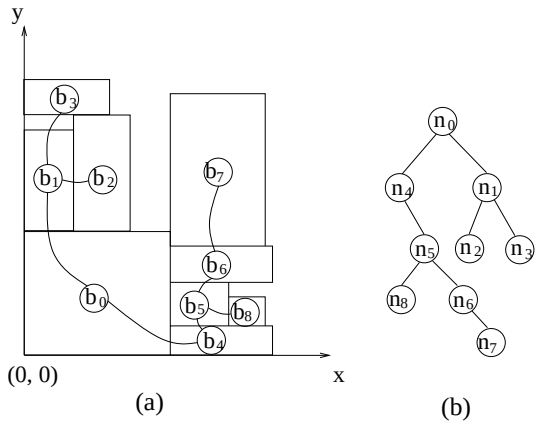


Figure 2: An admissible placement and its corresponding B*-tree.

As shown in Figure 2, we make n_0 the root of T since b_0 is on the bottom-left corner. Constructing the left subtree of n_0 recursively, we make n_4 the left child of n_0. Since the left child of n_4 does not exist, we then construct the right subtree of n_4 (which is rooted by n_5). The construction is recursively performed in the DFS order. After completing the left subtree of n_0, the same procedure applies to the right subtree of n_0. Figure 2(b) illustrates the resulting B*-tree for the placement shown in Figure 2(a). The construction takes only linear time.

The B*-tree keeps the geometric relationship between two modules as follows. If node n_j is the left child of node n_i, module b_j must be located on the right-hand side and adjacent to module b_i in the admissible placement; i.e., $x_j = x_i + w_i$. Besides, if node n_j is the right child of n_i, module b_j must be located above and visible from module b_i, with the x-coordinate of b_j equal to that of b_i; i.e., $x_j = x_i$. Also, since the root of T represents the bottom-left module, the x- and y-coordinates of the module associated with the root $(x_{root}, y_{root}) = (0, 0)$.

A *contour* structure (see Figure 3), which is originally proposed in [2], can be used to reduce the run time of finding the y-coordinate of a newly inserted block. The contour structure is a double linked list of blocks, which describes the contour line in the current compaction direction. Without the contour structure, the run time for placing a new block is linear to the number of blocks. By maintaining the contour structure, however, the y-coordinate of a block can be computed in $O(1)$ time. Figure 3 illustrates how to update the contour when we add a new block to the placement.

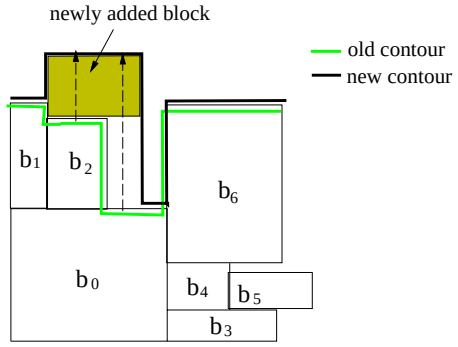


Figure 3: A contour and its update: when adding a new block to the placement, we search the contour from left to right and update it with the top boundary of the new block.

4 L-shaped Blocks

In this section, we apply the B*-tree approach to find a feasible placement with L-shaped blocks. Let b_L denote an L-shaped block. b_L can be partitioned into two rectangular *sub-blocks* by slicing b_L along its middle vertical boundary. As shown in Figure 4(a), b_1 and b_2 are the sub-blocks of b_L, and we say $b_1, b_2 \in b_L$.

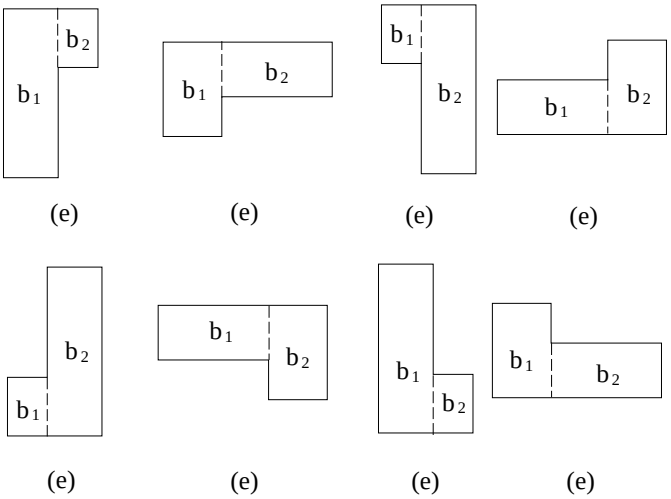


Figure 4: Eight situations of an L-shaped block. Each is partitioned into two parts by slicing it along the middle vertical boundary.

After partitioning and placement, the rectilinear block b_L might

not conform to its top profile sequence, as illustrated in Figure 5. Figure 5(a) shows a B*-tree and its corresponding placement. We can pull sub-block b_2 up to align with the sub-block b_1, so that the block b_L can maintain its top profile sequence without changing the overall topology of the blocks. Oppositely, there might not be enough space to do so; see Figure 5(b) for such an example. It is obvious that a feasible placement can be generated from the B*-tree shown in Figure 5(a) with a local adjustment, but it is impossible for the case shown in Figure 5(b). Therefore, if we represent an L-shaped block by two sub-blocks, we must guarantee that the two sub-blocks abut. To ensure that the left sub-block b_1 and the right sub-block b_2 of an L-shaped block b_L abut, we impose the following *location constraint* (LC for short) for b_1 and b_2:

LC: Keep b_2 as b_1's left child in the B*-tree.

The LC relation ensures that the x-coordinate of the left boundary of b_2 is equal to that of the right boundary of b_1. For example, the two sets of sub-blocks b_1, b_2 and b_3, b_4 shown in Figure 6(a) do not abut while those shown in Figure 6(b) do. In Figure 6(b), the sub-blocks b_3 and b_4 are placed at the right locations while the sub-blocks b_1 and b_2 are not since the y-coordinates of b_1 and b_2 are not equal. We say b_1 and b_2 are *mis-aligned*.

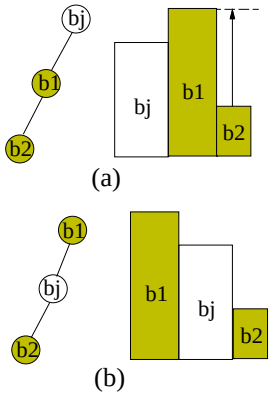


Figure 5: Placing the L-shaped block shown in Figure 4(a) by two sub-blocks: (a) a feasible placement; (b) an infeasible placement.

In the following, we adopt the contour data structure to solve the mis-alignment problem. When transforming a B*-tree to its corresponding placement, we update the contour to maintain its top profile sequence as follows. Assume that b_1 and b_2 are the respective left and right sub-blocks of an L-shaped block b_L, and they are mis-aligned. When processing b_2, b_1 must have been placed. We can classify the mis-alignment into two categories and adjust them as follows:

1. *Basin:* The contour is lower than the top profile sequence at the position of the current sub-block b_2. (See Figure 7(a).) In this case, we pull b_2 up to conform to the top profile sequence of the L-shaped block b_L.
2. *Plateau*: The contour is higher than the top profile sequence at the position of the current sub-block b_2. (See Figure 7(b).) In this case, we pull b_1 up to conform to the top profile sequence of b_L. (Note that b_2 cannot be moved down because the compaction operation makes b_2 be placed right above another block.)

It is clear that each of the adjustment can be performed in constant time with the contour data structure.

In the following, we discuss the rotation and flip operations of an L-shaped block. For each L-shaped block b_i, there are eight orientations by rotation and flip, as shown in Figure 4. To preserve the LC relation and keep it in the B*-tree, we repartition b_i into two sub-blocks after it is rotated or flipped and keep the LC relation between them. Figure 4 shows the sub-blocks after repartitioning. As shown

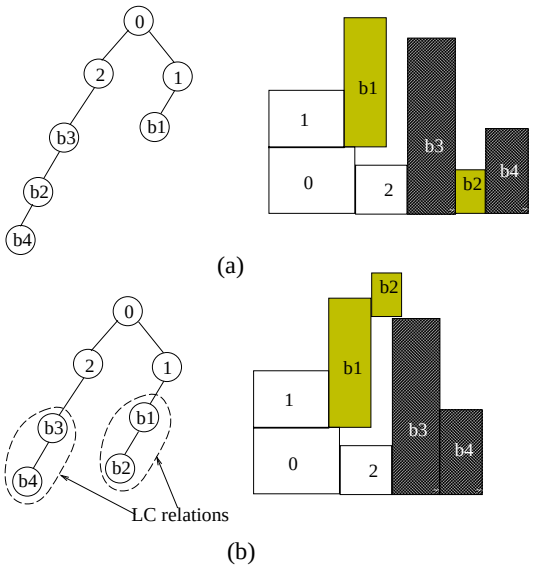


Figure 6: Suppose that b_1, b_2 and b_3, b_4 are two sets of sub-blocks corresponding to two L-shaped blocks. (a) A placement in which b_1, b_2 and b_3, b_4 do not abut. Their corresponding nodes in the B*-tree may not be related. (b) Another placement in which b_1, b_2 and b_3, b_4 abut. Their corresponding nodes in the B*-tree keep the LC relation between b_1 and b_2 (as well as b_3 and b_4).

in the figure, an L-shaped block is always partitioned by slicing it along the middle vertical boundary. After repartitioning, we should update the top profile sequence for the block.

5 Floorplan Algorithm

Our rectilinear floorplan design algorithm is based on the simulated annealing method [6, 15] and the B*-tree described in Section 3. We perturb a B*-tree (a feasible solution) to another B*-tree by using the following four operations.

Op1: Rotate a block.

Op2: Flip a block.

Op3: Move a block to another place.

Op4: Swap two blocks.

The Op1 and Op2 operations have been described in Section 4. The Op3 operation deletes and inserts a block into a B*-tree. If the deleted

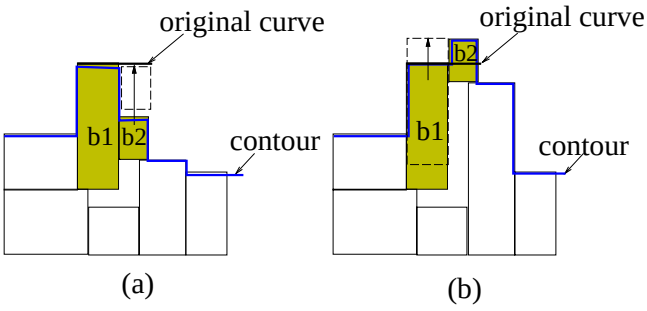


Figure 7: Placing two sub-blocks b_1 and b_2 of an L-shaped block. (a) If the contour is lower than the top profile sequence at b_2, then we pull b_2 up to meet the top profile sequence. (b) If the contour is higher than the top profile sequence at b_2, then we pull b_1 up to meet the top profile sequence.

node is associated with a rectangular block, we simply delete the node from the B*-tree. Otherwise, there will be two nodes associated with an L-shaped block, and we must delete the two nodes from the B*-tree and insert them to other places. Note that the LC relations must hold. Both of the Op3 and Op4 operations need to apply the $Insert(n_i)$ and $Delete(n_i)$ operations, where $Insert(n_i)$ ($Delete(n_i)$) is the operation for inserting (deleting) a node n_i to (from) a B*-tree. The B*-tree must remain a binary tree after deletion or insertion. We detail the deletion and insertion operations in the following.

5.1 Deletion

The deletion can be categorized into three cases:

- Case 1: A leaf node.
- Case 2: A node with one child.
- Case 3: A node with two children.

In Case 1, we can just delete the target leaf node directly, and the tree will still be a B*-tree. As shown in Figure 8(a), to delete the node n_7 from the B*-tree of Figure 2, we set the left child field of its parent n_6 to be *NULL* and free the node n_7.

In Case 2, we remove the target node and then place the single child at the position of the removed node. For example, after deleting the node n_4 from the B*-tree of Figure 2, we move n_5 to the original position of n_4 and obtain the tree shown in Figure 8(b). This tree update can be performed in $O(1)$ time. Note that the relative positions of the blocks might be changed after the operation, and thus we might need to reconstruct a corresponding placement for further processing.

In Case 3, when deleting a target node n_t with two children, we replace n_t by either its right child or left child n_c. Then we move a child of n_c to the original position of n_c. The process proceeds until the corresponding leaf node is handled. For instance, suppose that we delete the node n_0 from the B*-tree of Figure 2. We can use the right child n_1 to replace it, and then use n_3 to replace n_1. (The resulting tree is shown in Figure 8(c).) It is obvious that such a deletion operation requires $O(h)$ time, where h is the height of the B*-tree. Again the relative positions of the blocks might be changed after the operation, and thus we might need to reconstruct a corresponding placement for further processing.

Note that if the deleted node n_i is a sub-block of an L-shaped b_L, we should also delete the other sub-block of b_L.

5.2 Insertion

When adding a block to a placement, we may place the block around certain block, but not between two sub-blocks that belong to an L-shaped block. For a B*-tree, we define three types of positions as follows. (See Figure 9 for an illustration.)

- *Inseparable position:* A position between two nodes associated with the two sub-blocks of an L-shaped block.
- *Internal position:* A position between two nodes in a B*-tree, but is not an inseparable one.
- *External position:* A position pointed by a NULL pointer.

Only internal and external positions can be used for inserting a new node.

For a rectangular block, we can insert it into an internal or an external position directly. For any L-shaped block b_L consisting of two sub-blocks b_1 and b_2, with b_1 on the left-hand side of b_2, the two sub-blocks must be inserted to a B*-tree simultaneously, and b_2 must be the left child of b_1 (according to the LC relation).

In the following, we discuss three cases of for inserting an L-shaped block to an internal position. As shown in Figure 10, if we insert two nodes b_1 and b_2 of an L-shaped block to an internal position between nodes b_i and b_j, with b_j being a child of b_i, b_j can be placed at the position that is the left child of b_2, the right child of b_2, or the right child of b_1.

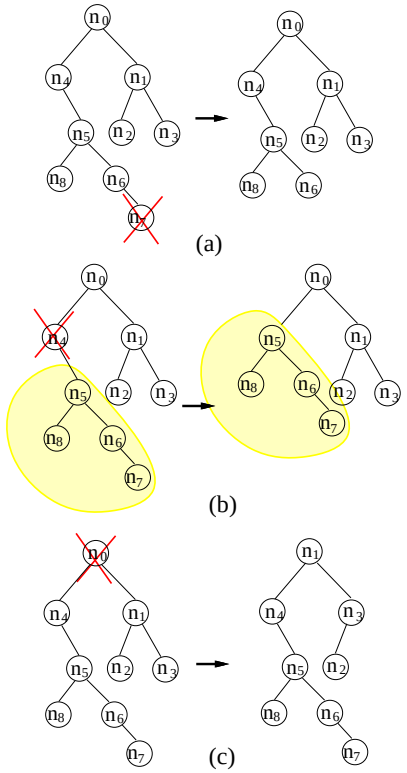


Figure 8: Deletion. (a) Deleting a leaf node, (b) Deleting a node with only one child, (c) Deleting a node with two children.

6 Extension to General Rectilinear Blocks

In this section, we extend the techniques described in previous sections to handle general rectilinear blocks. In general, a rectilinear block can be partitioned into a set of rectangular sub-blocks. Let b_i denote an arbitrarily shaped rectilinear block. b_i can be partitioned into a set of rectangular sub-blocks by slicing b_i from left to right along every vertical boundary of b_i, as shown in Figure 11(a).

After perturbing the Op1 and Op2 operations, we repartition a rectilinear block when it is rotated or flipped. Figure 11(b) shows the block of Figure 11(a) after rotating by 90° clockwise; there are six sub-blocks in it after the repartition.

There are two types of rectilinear blocks: *convex* and *concave blocks*. A rectilinear block is convex if any two points within the block can be connected by a shortest Manhattan path which also lies within the block; the block is concave, otherwise. Figure 11 and Figure 12 show two convex and a concave blocks, respectively. A convex block b_C can be partitioned into a set of sub-blocks $b_1, b_2, \ldots, b_n$ ordered from left to right. Considering the LC relation, we keep the sub-block b_{i+1} as b_i's left child in the B*-tree to ensure that they are placed side by side along the x-direction, where $1 \leq i \leq n-1$. To ensure that $b_1, b_2, \ldots, b_n$ are not mis-aligned, we modify the processing for *Basin* and *Plateau* as follows.

- *Basin:* The contour is lower than the top profile sequence at the position of a sub-block. We pull the sub-block up to conform to the top profile sequence.
- *Plateau*: The top boundary of a sub-block b_i ($1 \leq i \leq n$) in the contour is higher than the top profile sequence at the position of b_i. Assume that b_i has the largest top boundary. We pull all sub-blocks, except b_i, up to conform to the top profile sequence.

Moreover, all sub-blocks must be deleted (or inserted) together for the OP3 and OP4 operations.

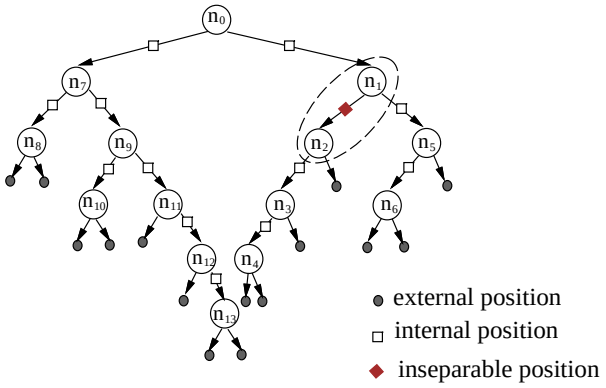


Figure 9: The inseparable, internal, and external positions of a B*-tree. (Assume that n_1 and n_2 are associated with the same L-shaped block.) A node can be inserted at either an internal or an external position.

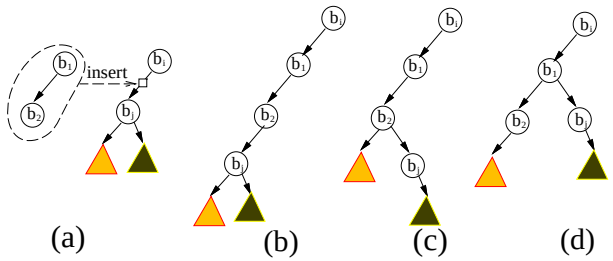


Figure 10: Three cases of inserting an L-shaped block to an internal position.

For a concave block, there might be empty space between two sub-blocks. As shown in Figure 12, the sub-block b_1 is placed above the sub-block b_2, which cannot be characterized by an LC relation in the B*-tree. Nevertheless, we can fill the concave holes of a concave block and make it a convex block. We call this operation a *filling approximation* for the rectilinear block. For any concave block, we treat it as a convex block after applying appropriate filling.

7 Experimental Results

We implemented our algorithm in the C++ programming language on a 450MHz SUN Ultra Sparc-I workstation with 1 GB memory. Since the benchmarks in previous work are artificial cases and unavailable to us, we generate some general benchmarks for experiments in this paper. Our test cases were generated by cutting a rectangle into a set of blocks. Therefore, the optimum area is given by the original block.

As shown in Table 1, Columns 2, 3 and 4 list the numbers of rectangular, L-shaped, and T-shaped blocks. RL10, RL20, and RL30 consist of only rectangular and L-shaped blocks. There are five rectangular and five L-shaped blocks in RL10, ten rectangular and ten L-shaped blocks in RL20, and fifteen rectangular and fifteen L-shaped blocks in RL30, respectively. RLT10, RLT20 and RLT30 consist of not only rectangular and L-shaped blocks, but also T-shaped ones. RLT10 is composed of four rectangular, three L-shaped, and three T-shaped blocks, RLT20 is composed of seven rectangular, seven L-shaped, and six T-shaped blocks, and RLT30 is composed of ten rectangular, ten L-shaped, and ten T-shaped blocks. The original area of each test case is shown in Column 5. Columns 6 and 7 list the resulting area and the dead space (%). The results show that our algorithm obtains the optimum area for RL10 and near optimum areas for RL20, RL30, RLT10, RLT20, and RLT30 with areas only 2.00%, 4.00%, 2.00%, 3.50%, and 5.00% away from the optima, respectively. The runtimes for achieving the results ranged from about 8 seconds to 50 minutes (see Column 8). Figures 13 and 14 show the optimum and the resulting placement for RL10 and RLT30, respectively.

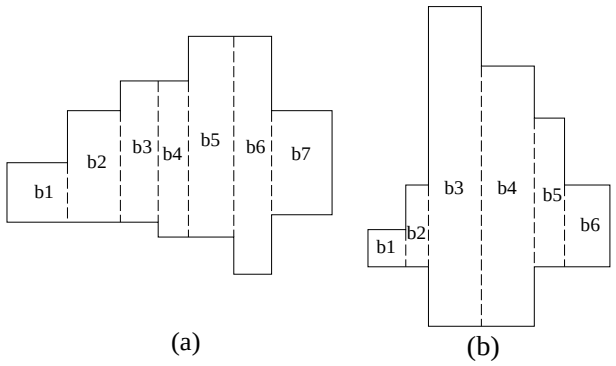


Figure 11: (a) Partition a convex block along every vertical boundary from left to right. (b) Repartition the block of (a) after it rotates.

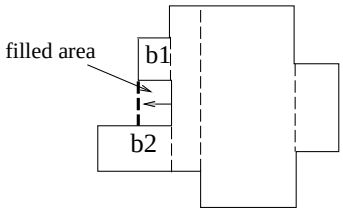


Figure 12: Filling approximation for a rectilinear block.

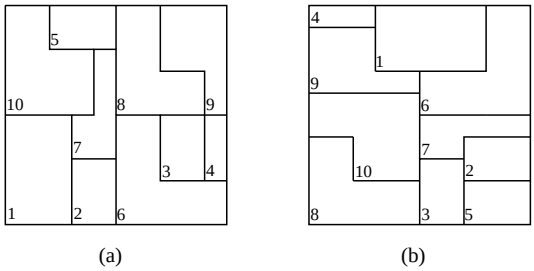


Figure 13: Placement for RL10: 5 rectangular and 5 L-shaped blocks. (a) The optimum placement (10 x 10); (b) The resulting placement (10 x 10).

8 Conclusions

In this paper, we have extended the B*-tree approach introduced in [1] to arbitrarily shaped rectilinear blocks. Rectilinear blocks were partitioned into a set of rectangular sub-blocks, each of them is individually represented by a node in the B*-tree. The LC relations and the *basin* and *plateau* operations were used to ensure that each block keeps its original shape. The experiment results have shown that our approach is very effective in area utilization.

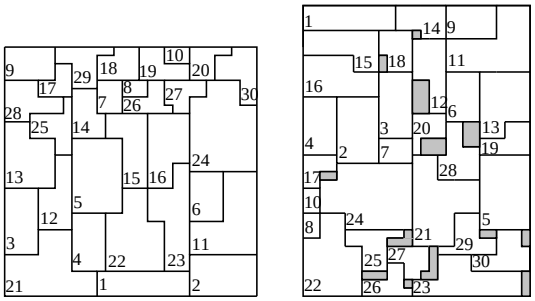
References

[1] Yun-Chih Chang, Yao-Wen Chang, Guang-Ming Wu and Su-Wei Wu, "B*-Trees: A New Representation for Non-Slicing Floorplans," *Proc. IEEE/ACM Design Automation Conf.*, pp. 458–463, 2000.

[2] Pei-Ning Guo, Chung-Kuan Cheng, and Takeshi Yoshimura, "An O-Tree Representation of Non-Slicing Floorplan and Its

Circuits	#Rectangular blocks	#L-shaped blocks	#T-shaped blocks	Optimum area	Resulting area	Dead space (%)	Runtime (sec)
RL10	5	5	0	100 (10 x 10)	100 (10 x 10)	0.00	8
RL20	10	10	0	400 (20 x 20)	408 (15 x 27)	2.00	307
RL30	15	15	0	900 (30 x 30)	936 (29 x 32)	4.00	1636
RLT10	4	3	3	100 (10 x 10)	102 (6 x 17)	2.00	41
RLT20	7	7	6	400 (20 x 20)	414 (18 x 23)	3.50	1096
RLT30	10	10	10	900 (30 x 30)	945 (27 x 35)	5.00	3007

Table 1: The experimental results.



(a) (b)

Figure 14: Placement for RLT30: 10 rectangular, 10 L-shaped, and 10 T-shaped blocks. (a) The optimum placement (30 x 30); (b) The resulting placement (27 x 35).

Applications,"*Proc. IEEE/ACM Design Automation Conf.*, pp. 268–273, 1999.

[3] Christos H. Papadimitriou, and Kenneth Steiglitz, *Combinatorial Optimization,* prentice Hall, 1982.

[4] M. Z. Kang and W. Dai., "General floorplanning with L-shaped, T-shaped and soft blocks based on bounded slicing grid structure," *Proc. Asia and South Pacific Physical Design Automation Conf.*, pp. 265–270, 1997.

[5] M. Z. Kang and W. Dai., "Arbitrary Rectilinear Block Packing Based on Sequence Pair," *Proc. International Conference on Computer-Aided-Design*, pp. 259–266, 1998.

[6] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671–680, May 13, 1983.

[7] T. C. Lee, "An Bounded 2D Contour Searching Algorithm for Floorplan Design with Arbitrarily Shaped Rectilinear and Soft Modules, *"Proc. IEEE/ACM Design Automation Conf.*, pp. 525–530, 1993.

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

[9] H. Murata, K. Fujiyoshi, and M. Kaneko, "VLSI/PCB Placement with Obstacles Based Sequence Pair," *Proc. Internal Symposium on Physical Design*, pp. 26–31, 1997.

[10] H. Murata, Ernest S. Kuh, "Sequence Pair Based Placement Method for Hard/ Soft/Pre-placed Modules," *Proc. Internal Symposium on Physical Design*, pp. 167–172, 1998.

[11] S. Nakatake, K. Fujiyoshi, H. Murata, and Y. Kajitani, "Module Placement on BSG-Structure and IC Layout Applications," *Proc. IEEE International Conf. on Computer-Aided-Design*, pp. 484–491, 1996.

[12] S. Nakatake, M. Furuya, and Y. Kajitani, "Module Placement on BSG-Structure with Pre-Placed Modules and Rectilinear Modules," *Proc. Asia and South Pacific Physical Design Automation Conf.*, pp. 571–576, 1998.

[13] R. H. J. M. Otten, "Automatic Floorplan Design, *"Proc. Design Automation Conf.*, pp. 261–267, 1992.

[14] B. T. Preas, and W. M. vanCleemput, "Placement Algorithms for Arbitrarily Shaped Blocks," *Proc. IEEE/ACM Design Automation Conf.*, pp. 474–480, 1979.

[15] C. Sechen, and A. Sangiovanni-Vincentelli, "The TimberWolf Placement and Routing Package," *IEEE Journal of Solid-State Circuits*, vol. 20, no. 2, pp. 510–522, Apr. 1985.

[16] T. C. Wang, and D. F. Wong, "An Optimal Algorithm for Floorplan and Area Optimization, *"Proc. IEEE/ACM Design Automation Conf.*, pp. 180–186, 1990.

[17] D. F. Wong, and C. L. Liu, "A New Algorithm for Floorplan Design," *Proc. IEEE/ACM Design Automation Conf.*, pp. 101–107, 1986.

[18] D. F. Wong, and C. L. Liu, "Floorplan Design for Rectangular and L-shaped Modules," *Proc. IEEE International Conf. on Computer-Aided-Design*, pp. 520–523, 1987.

[19] Jin Xu, Pei-Ning Guo, and Chung-Kuan Cheng, "Rectilinear Block Placement Using Sequence-Pair," *Proc. Internal Symposium on Physical Design*, pp. 173–178, 1998.