Rectilinear Block Placement Using B*-Trees

GUANG-MING WU
Nan-Hua University
YUN-CHIH CHANG
Realtek Semiconductor Corp.
and
YAO-WEN CHANG
National Taiwan University

Due to the layout complexity in modern VLSI designs, integrated circuit blocks may not be rectangular. However, literature on general rectilinear block placement is still quite limited. In this article, we present approaches for handling the placement for arbitrarily shaped rectilinear blocks using B*-trees [Chang et al. 2000]. We derive the feasibility conditions of B*-trees to guide the placement of rectilinear blocks. Experimental results show that our algorithm achieves optimal or near-optimal block placement for benchmarks with various shaped blocks.

Categories and Subject Descriptors: B7.2 [**Integrated Circuits**]: Design Aids—*placement and routing*; J.6 [**Computer Applications**]: Computer-Aided Engineering

General Terms: Algorithms, Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Computer-aided design of VLSI, floorplanning, layout, placement

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.

Floorplanning is often studied based on two floorplan structures, the *slicing structure* [Otten 1982; Wong and Liu 1986] and the *nonslicing structure* [Chang

The work of G.-M. Wu was partially supported by the National Science Council of Taiwan ROC under Grant No. NSC-91-2215-E-343-001. The work of Y.-W. Chang was partially supported by the National Science Council of Taiwan ROC under Grant No. NSC-91-2215-E-002-038.
Authors' addresses: G.-M. Wu, Department of Information Management, Nan-Hua University, Chiayi, Taiwan; email: gmwu@mail.nhu.edu.tw; Y.-C. Chang, Realtek Semiconductor Corp., No. 2, Industry E. Rd. IX, Science-Based Industrial Park, Hsinchu, Taiwan; Y.-W. Chang, Graduate Institute of Electronics Engineering and Department of Electrical Engineering, National Taiwan University, Rm. 548, Electrical Engineering Building #1, Taipei 106, Taiwan; email: ywchang@cc.ee.ntu.edu.tw.

et al. 2000; Guo et al. 1999; Murata et al. 1997; Nakatake et al. 1996; Wang and Wong 1990]. 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 [1986] proposed an algorithm for slicing floorplan design. They presented a normalized Polish expression to represent a slicing structure, enabling the speedup of the search procedure. However, this representation cannot handle nonslicing floorplans. Recently, researchers have proposed several representations for nonslicing floorplans, such as the sequence pair [Murata et al. 1997], bounded slicing grid (BSG) [Nakatake et al. 1996], O-tree [Guo et al. 1999], and B*-tree [Chang et al. 2000].

In modern VLSI design, the blocks may not be rectangular. Most existing floorplanning/placement algorithms, however, only deal with rectangles and cannot apply to arbitrarily shaped rectilinear block placement directly. New approaches that can handle arbitrarily shaped blocks are essential to optimize resource (e.g., area) utilization.

Preas and van Cleemput [1979] proposed a graph model for the topological relationships among rectangular and arbitrarily shaped rectilinear blocks. Wong and Liu [1987] extended the Polish expression to represent slicing floorplans with rectangular and L-shaped blocks. Lee [1993] extended the zone refinement technique to rectilinear blocks. A bounded 2-D contour searching algorithm was proposed to find the best position for a block.

Kang and Dai [1997] proposed a BSG-based method to solve the packing of rectangular, L-shaped, T-shaped, and soft blocks. The algorithm combined simulated annealing and a genetic algorithm for general nonslicing floorplans.

Xu et al. [1998] 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 were examined first, and then arbitrarily shaped rectilinear blocks were decomposed into a set of L-shaped blocks.

Kang and Dai [1998] proposed a method based on the sequence-pair structure for rectilinear block placement. Three necessary and sufficient conditions for a sequence pair to be feasible were derived. A stochastic search was applied on the optimization of convex block floorplanning.

Chang et al. [2000] recently proposed the *B*-tree* representation for nonslicing floorplans, which is based on block compaction and ordered binary trees. Inheriting from the nice properties of ordered binary trees, B*-trees are very easy to implement and require only constant time for tree search and insertion, and linear time for deletion. Unlike the O-tree representations, in particular, no extra encoding of the tree itself is needed for a B*-tree, and cost evaluation can be performed directly on a B*-tree. Besides, the ordered property of a B*-tree makes the incremental cost evaluation of its corresponding placement possible. Furthermore, 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 nonslicing floorplans.

In this article, we apply the B*-tree to handle the placement of arbitrarily shaped rectilinear blocks. First, we explore the properties of L-shaped blocks and then extend the properties to general rectilinear blocks. The feasibility conditions are then used to guide the placement of rectilinear blocks. We construct a

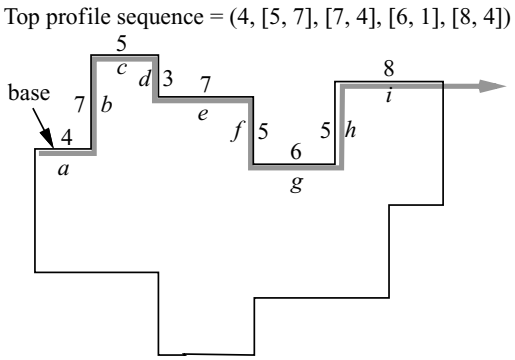

Fig. 1. The top profile sequence consists of the length of the base, followed by a sequence of two-tuples that is composed of the lengths of the succeeding horizontal segments and their relative heights to the base.

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 article 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 arbitrarily 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 nonoverlapping 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 consists 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 a length of 4 units. The second horizontal segment is c which has a length of 5 units and is 7 units higher than the base a. Similarly, the third horizontal segment is e which has a length of 7 units and is 4 units higher

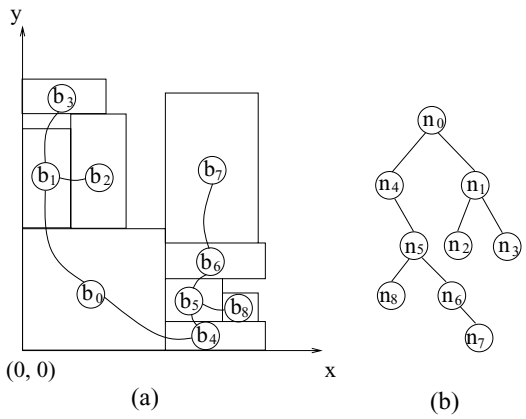

Fig. 2. An admissible placement and its corresponding B*-tree.

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 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 optimize a given cost metric (such as area, wirelength, etc.) induced by the assignment of b_is. (By area here, we mean 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 T [Chang et al. 2000]. (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 in the bottom-left corner. Similar to the depth-first search (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 first module located above and visible from b_i, with its x-coordinate equal to that of b_i (and its y-coordinate less than or equal to that of the top boundary of the module on the left-hand side and adjacent to b_i, if any, to consider the special placement).

As shown in Figure 2, we make n_0 the root of T since b_0 is in 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.

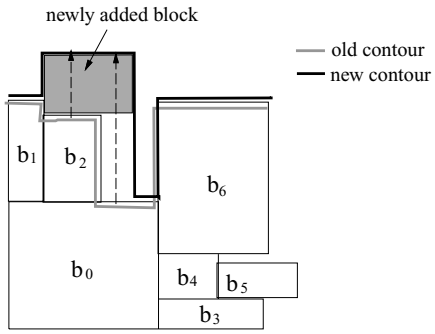

Fig. 3. A contour and its update: when adding a new block to the placement, we search the contour from left to right and update it with the top boundary of the new block.

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; that is, $x_j = x_i + w_i$. 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; that is, $x_j = x_i$. 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 was originally proposed in Guo et al. [1999], can be used to reduce the run-time of finding the y-coordinate of a newly inserted block. The contour structure is a doubly 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.

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 *subblocks* by slicing b_L along its middle vertical boundary. As shown in Figure 4(a), b_1 and b_2 are the subblocks of b_L, and we say $b_1, b_2 \in b_L$.

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 subblock b_2 up to align with the subblock b_1, so that the block b_L can maintain its top profile sequence without changing the overall topology of the blocks. Conversely, 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 subblocks, we must guarantee that the two subblocks abut. To ensure that the left subblock b_1 and the

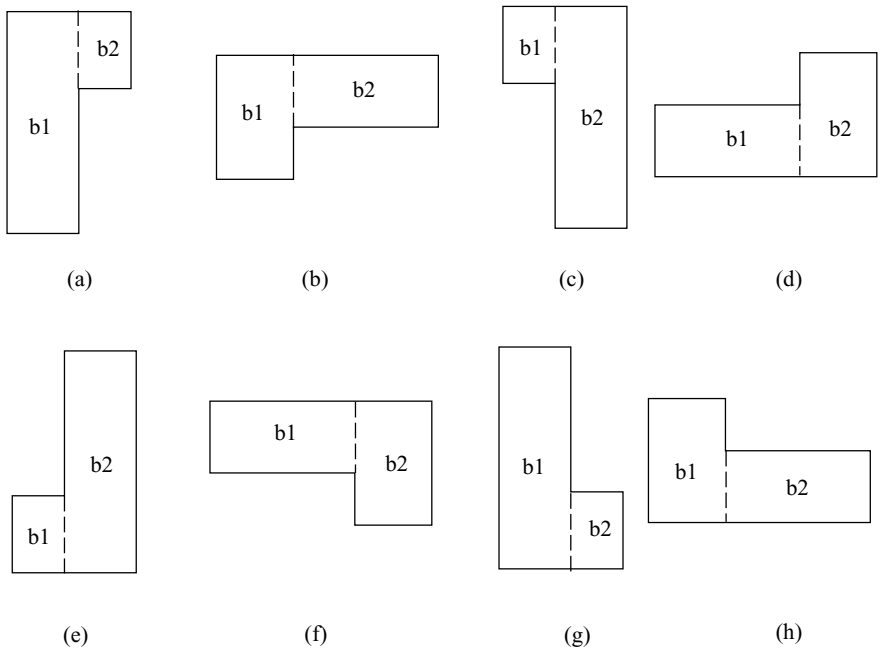

Fig. 4. Eight situations of an L-shaped block. Each is partitioned into two parts by slicing it along the middle vertical boundary.

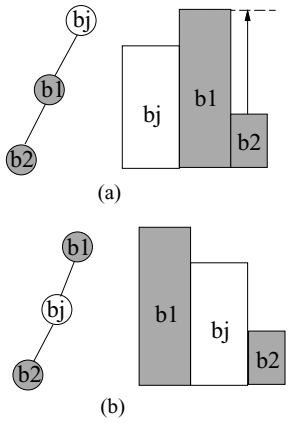

Fig. 5. Placing the L-shaped block shown in Figure 4(a) by two subblocks: (a) a feasible placement; (b) an infeasible placement.

right subblock 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 subblocks b_1, b_2 and b_3, b_4 shown in Figure 6(a) do not abut whereas those shown in Figure 6(b)

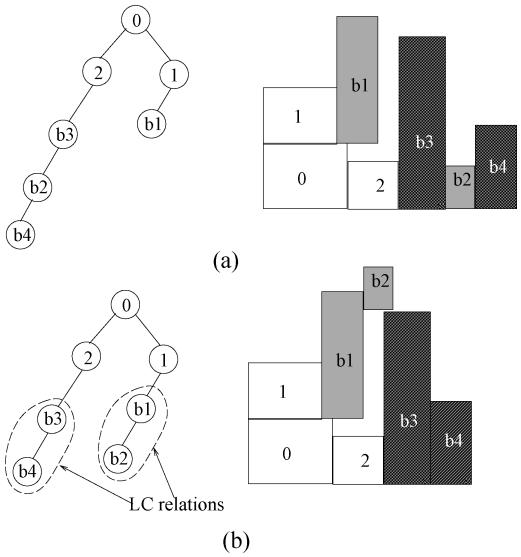

Fig. 6. Suppose that b_1, b_2 and b_3, b_4 are two sets of subblocks corresponding to two L-shaped blocks (a) a placement in which b_1, b_2 and b_3, b_4 do not abut. Their corresponding nodes in the B*-tree may not be related; (b) another placement in which b_1, b_2 and b_3, b_4 abut. Their corresponding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

do. In Figure 6(b), the subblocks b_3 and b_4 are placed at the right locations and the subblocks 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 *misaligned*.

In the following, we adopt the contour data structure to solve the misalignment 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 subblocks of an L-shaped block b_L, and they are misaligned. When processing b_2, b_1 must have been placed. We can classify the misalignment into categories and adjust them as follows.

(1) *Basin:* The contour is lower than the top profile sequence at the position of the current subblock 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. It should be noted that this operation will not pull other blocks up due to the DFS packing order induced by the B*-tree.
(2) *Plateau*: The contour is higher than the top profile sequence at the position of the current subblock 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 adjustments 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

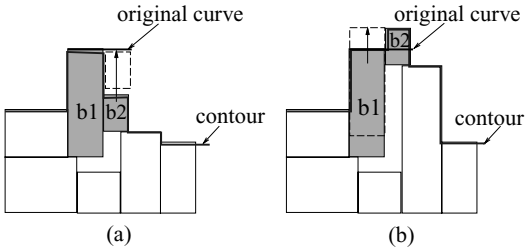

Fig. 7. Placing two subblocks b_1 and b_2 of an L-shaped block: (a) if the contour is lower than the top profile sequence at b_2, then we pull b_2 up to meet the top profile sequence; (b) if the contour is higher than the top profile sequence at b_2, then we pull b_1 up to meet the top profile sequence.

flip, as shown in Figure 4. To preserve the LC relation and keep it in the B*-tree, we repartition b_i into two subblocks after it is rotated or flipped and keep the *LC* relation between them. Figure 4 shows the subblocks after repartitioning. As shown 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 [Kirkpatrick et al. 1983; Sechen and Sangiovanni 1985] 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 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 in 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 these cases:

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

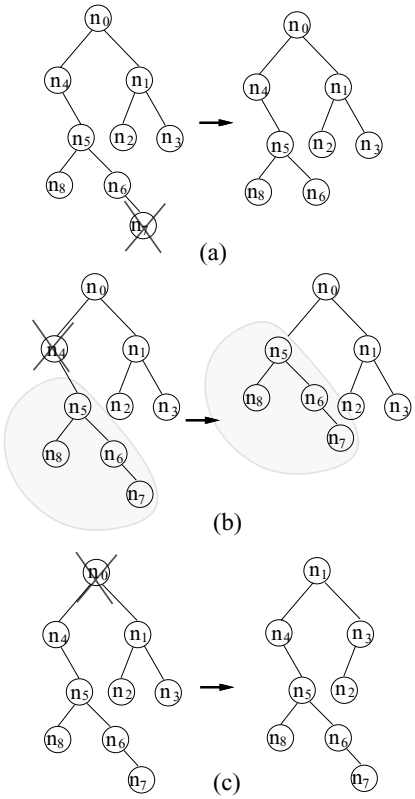

Fig. 8. Deletion: (a) deleting a leaf node; (b) deleting a node with only one child; (c) deleting a node with two children.

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

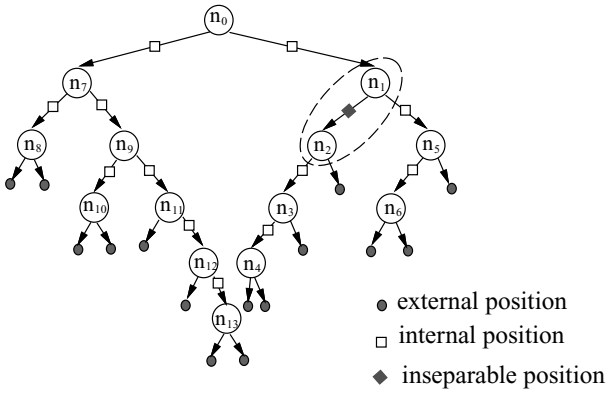

Fig. 9. The inseparable, internal, and external positions of a B*-tree. (Assume that n_1 and n_2 are associated with the same L-shaped block.) A node can be inserted at either an internal or an external position.

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 subblock of an L-shaped b_L, we should also delete the other subblock of b_L.

5.2 Insertion

When adding a block to a placement, we may place the block around a certain block, but not between two subblocks 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 subblocks of an L-shaped block.

—*Internal position:* A position between two nodes in a B*-tree, but not an inseparable one.

—*External position:* A position pointed to 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 subblocks b_1 and b_2, with b_1 on the left-hand side of b_2, the two subblocks must be inserted into 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 inserting an L-shaped block into an internal position. As shown in Figure 10, if we insert two nodes b_1 and b_2 of an L-shaped block into 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.

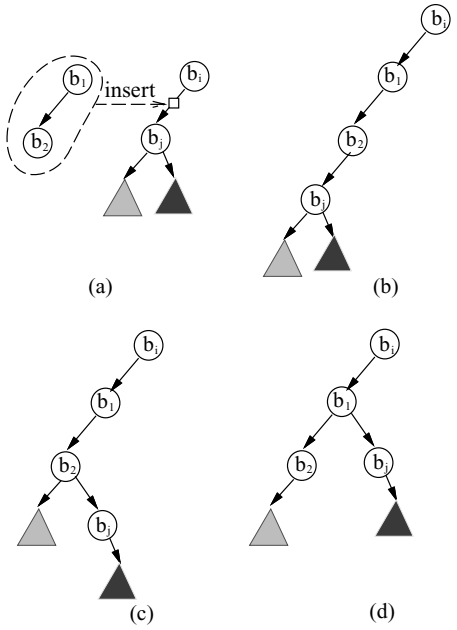

Fig. 10. Three cases of inserting an L-shaped block to an internal position.

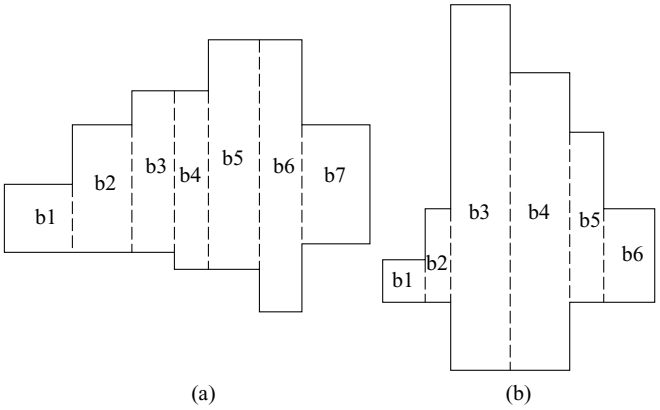

Fig. 11. (a) Partition a convex block along every vertical boundary from left to right; (b) repartition the block of (a) after it rotates.

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 subblocks. Let b_i denote an arbitrarily shaped rectilinear block. b_i can be partitioned into a set of rectangular subblocks 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

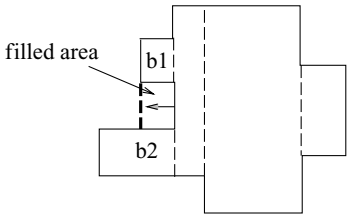

Fig. 12. Filling approximation for a rectilinear block.

Figure 11(a) after rotating 90° clockwise; there are six subblocks in it after the repartition.

There are two types of rectilinear blocks: *convex* and *concave*. A rectilinear block is convex if any two points within the block can be connected by a shortest Manhattan path that also lies within the block; the block is concave, otherwise. Figures 11 and 12 show two convex and a concave block, respectively. A convex block b_C can be partitioned into a set of subblocks $b_1, b_2, \ldots, b_n$ ordered from left to right. Considering the *LC* relation, we keep the subblock 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 misaligned, 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 subblock. We pull the subblock up to conform to the top profile sequence.

—*Plateau*: The top boundary of a subblock 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 subblocks, except b_i, up to conform to the top profile sequence.

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

For a concave block, there might be empty space between two subblocks. As shown in Figure 12, the subblock b_1 is placed above the subblock 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 450 MHz 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 article. 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 I, Columns 2 through 4 list the numbers of rectangular, L-shaped, and T-shaped blocks. RL10, RL20, and RL30 consist of only

Table I. Experimental Results

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

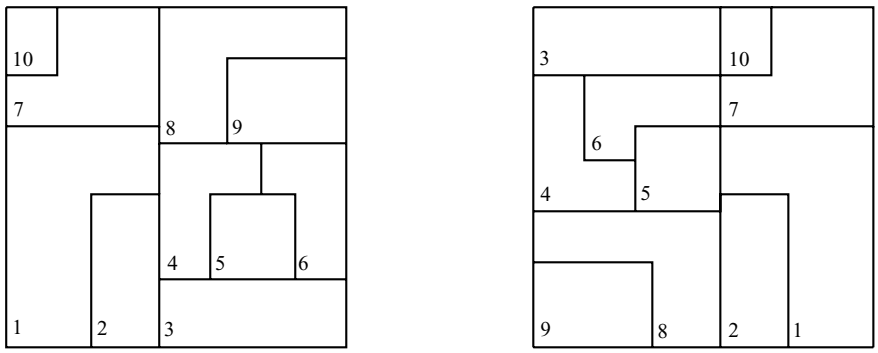

Fig. 13. Placement for RL10: 5 rectangular and 5 L-shaped blocks: (a) the optimum placement (10×10); (b) the resulting placement (10×10).

rectangular and L-shaped blocks. There are 5 rectangular and 5 L-shaped blocks in RL10, 10 rectangular and 10 L-shaped blocks in RL20, and 15 rectangular and 15 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 4 rectangular, 3 L-shaped, and 3 T-shaped blocks; RLT20 is composed of 7 rectangular, 7 L-shaped, and 6 T-shaped blocks; and RLT30 is composed of 10 rectangular, 10 L-shaped, and 10 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 run-times 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.

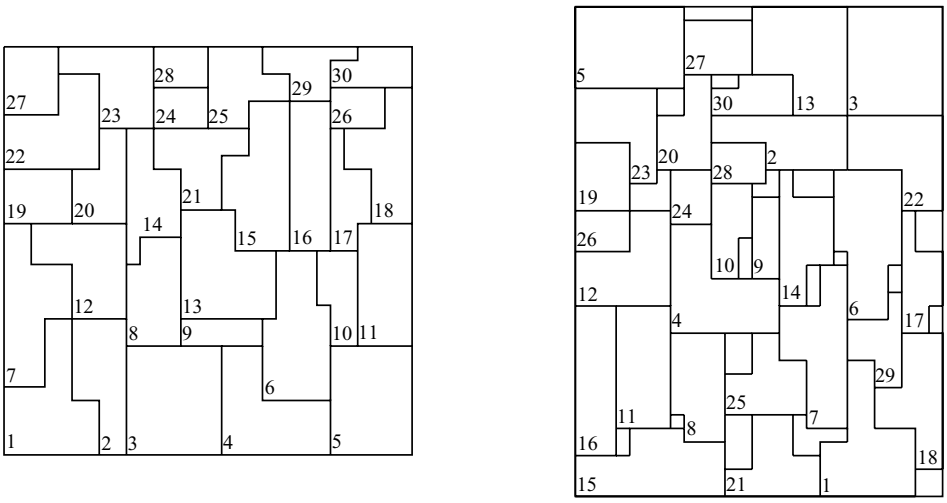

Fig. 14. Placement for RLT30: 10 rectangular, 10 L-shaped, and 10 T-shaped blocks: (a) the optimum placement (30×30); (b) the resulting placement (27×35).

8. CONCLUSIONS

In this article, we have extended the B*-tree approach introduced in Chang et al. [2000] to handle the placement of arbitrarily shaped rectilinear blocks. We partitioned a rectilinear block into a set of rectangular subblocks, each 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 kept its original shape. The experiment results have shown that our approach is very effective in area utilization.

REFERENCES

CHANG, Y.-C., CHANG, Y.-W., WU, G.-M., AND WU, S.-W. 2000. B*-Trees: A new representation for non-slicing floorplans. In *Proceedings of the ACM/IEEE Design Automation Conference*, (Los Angeles, June), 458–463.

GUO, P.-N., CHENG, C.-K., AND YOSHIMURA, T. 1999. An O-tree representation of non-slicing floorplan and its applications. In *Proceedings of the ACM/IEEE Design Automation Conference* (California, June), 268–273.

PAPADIMITRIOU, C. AND STEIGLITZ, K. 1982. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall Inc., Englewood Cliffs, NJ.

KANG, M. Z. AND DAI, W. 1997. General floorplanning with L-shaped, T-shaped and soft blocks based on bounded slicing grid structure. In *Proceedings of the ACM/IEEE Asia and South Pacific Design Automation Conference* (Chiba, Japan, January 28–31), 265–270.

KANG, M. Z. AND DAI, W. 1998. Arbitrary rectilinear block packing based on sequence pair. In *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design* (San Jose, CA, November 8–12), 259–266.

KIRKPATRICK, S., GELATT, C. D., AND VECCHI, M. P. 1983. Optimization by simulated annealing. *Science 220*, 4598, 671–680.

LEE, T. C. 1993. A bounded 2D contour searching algorithm for floorplan design with arbitrarily shaped rectilinear and soft modules. In *Proceedings of the ACM/IEEE Design Automation Conference* (California, June), 525–530.

MURATA, H. AND KUH, E. S. 1998. Sequence pair based placement method for hard/soft/pre-placed modules. In *Proceedings of the ACM International Symposium on Physical Design* (California, April), 167–172.

MURATA, H., FUJIYOSHI, K., NAKATAKE, S., AND KAJITANI, Y. 1995. Rectangle-packing based module placement. In *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design* (California, November), 472–479.

MURATA, H., FUJIYOSHI, K., AND KANEKO, M. 1997. VLSI/PCB placement with obstacles based sequence pair. In *Proceedings of the ACM International Symposium on Physical Design* (California, April), 26–31.

NAKATAKE, S., FUJIYOSHI, K., MURATA, H., AND KAJITANI, Y. 1996. Module placement on BSG-structure and IC layout applications. In *Proceedings of ACM/IEEE International Conference on Computer-Aided Design* (California, November), 484–491.

NAKATAKE, S., FURUYA, M., AND KAJITANI, Y. 1998. Module placement on BSG-structure with pre-placed modules and rectilinear modules. In *Proceedings of the ACM/IEEE Asia and South Pacific Design Automation Conference* (Yokohama, February 10–13), 571–576.

OTTEN, R. H. J. M. 1982. Automatic floorplan design. In *Proceedings of the ACM/IEEE Design Automation Conference* (California, June), 261–267.

PREAS, B. T. AND VANCLEEMPUT, W. M. 1979. Placement algorithms for arbitrarily shaped Blocks. In *Proceedings of the ACM/IEEE Design Automation Conference* (California, June), 474–480.

SECHEN, C., AND SANGIOVANNI-VINCENTELLI, A. 1985. The TimberWolf placement and routing package. *IEEE J. Solid-State Circ. 20*, 2 (April), 510–522.

WANG, T. C. AND WONG, D. F. 1990. An optimal algorithm for floorplan and area optimization. In *Proceedings of the ACM/IEEE Design Automation Conference* (Orlando, FL, June 24–28), 474–480.

WONG, D. F. AND LIU, C. L. 1986. A new algorithm for floorplan design. In *Proceedings of the ACM/IEEE Design Automation Conference* (California, June), 101–107.

WONG, D. F. AND LIU, C. L. 1987. Floorplan design for rectangular and L-shaped modules. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design* (California, November), 520–523.

XU, J., GUO, P.-N., AND CHENG, C.-K. 1998. Rectilinear block placement using sequence-pair. In *Proceedings of the ACM International Symposium on Physical Design* (California, April), 173–178.

Received March 2001; accepted May 2002