B*-Trees: A New Representation for Non-Slicing Floorplans *

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

Abstract

We present in this paper an efficient, flexible, and effective data structure, *B*-trees*, for non-slicing floorplans. B*-trees are based on ordered binary trees and the admissible placement presented in [1]. Inheriting from the nice properties of ordered binary trees, B*-trees are very easy for implementation and can perform the respective primitive tree operations search, insertion, and deletion in only $O(1)$, $O(1)$, and $O(n)$ times while existing representations for non-slicing floorplans need at least $O(n)$ time for each of these operations, where n is the number of modules. The correspondence between an admissible placement and its induced B*-tree is 1-to-1 (i.e., no redundancy); further, the transformation between them takes only linear time. Unlike other representations for non-slicing floorplans that need to construct constraint graphs for cost evaluation, in particular, the evaluation can be performed on B*-trees and their corresponding placements *directly* and *incrementally*. We further show the flexibility of B*-trees by exploring how to handle rotated, pre-placed, soft, and rectilinear modules. Experimental results on MCNC benchmarks show that the B*-tree representation runs about 4.5 times faster, consumes about 60% less memory, and results in smaller silicon area than the O-tree one [1]. We also develop a B*-tree based simulated annealing scheme for floorplan design; the scheme achieves near optimum area utilization even for rectilinear modules.

1 Introduction

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

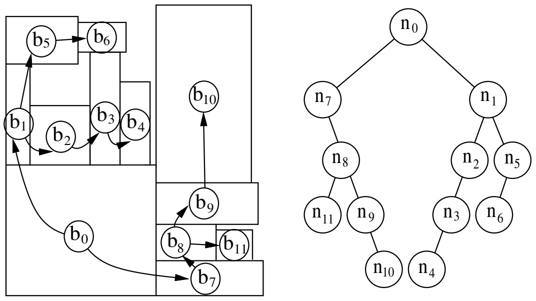
A fundamental problem to floorplanning/placement lies in the representation of geometric relationship among modules. The representation profoundly affects the operations of modules and the complexity of a floorplan/placement design process. It is thus desired to develop an efficient, flexible, and effective representation of geometric relationship for floorplan/placement designs.

1.1 Previous Work

Floorplans can be divided into two categories, the *slicing structure* [12, 15] and the *non-slicing structure* [1, 6, 9, 14]. 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 designs [15]. They presented a normalized Polish expression to represent a slicing structure, enabling the speed-up of its search procedure. However, this representation cannot handle non-slicing floorplans. Recently, researchers have proposed several representations for non-slicing floorplans, such as sequence pair [6], bounded slicing grid (BSG) [9], and O-tree [1].

Onodera et al. in [11] used the branch-and-bound method to solve the module placement problem. This approach has very high time complexity and is thus feasible for only small problem sizes. Murata et al. in [6] proposed the sequence-pair representation for rectangular module placement. The main idea is to use a sequence pair to represent the geometric relation of modules, place the modules on a grid structure, and construct corresponding constraint graphs to evaluate cost. This representation requires $2n\lceil \lg n \rceil$ space to encode a sequence pair and there are $(n!)^2$ combinations in total, where n is the number of modules. Further, the transformation between a sequence pair and a placement takes $O(n \lg n)$ time. Nakatake el al. in [9] presented a grid based representation—BSG. The BSG structure utilizes a set of horizontal and vertical bounded-length lines to cut the plane into rooms and represents a placement by these lines and rooms. BSG incurs redundancies. Its complexity is similar to that of the sequence pair. The sequence pair and BSG were then extended to handle pre-placed modules and soft modules in [2, 7, 8, 10].

Guo et al. in [1] proposed a tree based representation, called *O-trees*. The number of O-tree combinations is only $O(n!2^{2n-2}/n^{1.5})$. Further, the transformation between the representation and a floorplan takes only $O(n)$ time. However, the tree structure is irregular, and thus some primitive tree operations, such as search and insertion, are not efficient. To reduce the operation complexity, the tree is encoded by a sequence of $2n$ bits and a permutation of $n\lceil \lg n \rceil$ bits.



(a) (b)

Figure 1: (a) An admissible placement. (b) The (horizontal) B*-tree representing the placement.

1.2 Our Contributions

To handle non-slicing floorplans, we propose in this paper an ordered binary-tree based representation, called *B*-trees*. Given an *admissible placement* [1], we can represent it by a unique *horizontal* and a unique *vertical* B*-trees. (See Figure 1(b) for the horizontal B*-tree for the admissible placement shown in Figure 1(a).) The admissible placement here means that it is compacted and can neither move down nor move left. Inheriting from the particular characteristics of the ordered binary tree, the B*-tree has many advantages compared with other representations. We summarize the advantages of B*-tree as follows:

- Based on ordered binary trees, the B*-tree is very fast and easy for implementation. Since the number of branches in a B*-tree is fixed (i.e., two branches), it can be implemented by either a static data structure or a dynamic one. By using static memory, we can search and insert a node in $O(1)$ time while the O-tree needs $O(n)$ time. Note that an O-tree is irregular and its number of branches is unpredictable (see Figure 2(b)); thus, it incurs higher operation complexity and/or significant encoding cost.
- The B*-tree is very flexible for handling the floorplanning problems with various types of modules (e.g., hard, pre-placed, soft, and rectilinear modules) *directly* and *efficiently*. In contrast, other representation cannot handle some special modules well. To deal with soft modules using sequence pair, for example, previous work in [8] transformed it to the convex problem (which is NP-hard [13]), processed the soft modules, and then transformed back to the sequence pair.
- B*-trees do not need to construct constraint graphs for area cost evaluation. In contrast, the sequence pair and O-tree representa-

*This work was partially supported by the National Science Council of Taiwan under Grant No. NSC-89-2215-E-009-055. E-mail: {gis87512, ywchang, gis85815, gis8867555}@cis.nctu.edu.tw.

tions require encoding module sequences for tree operations and constructing constraint graphs for cost computation. Note that the complexity of the operations on a binary tree is smaller than that on a constraint graph. Besides, we can also save the time for constructing a constraint graph, which takes linear time for both the O-tree and the sequence pair.

- The area cost after exchanging two modules can be recomputed incrementally on a B*-tree. Specifically, the modules ahead the exchanged modules in the depth-first search (DFS) of a B*-tree remained unchanged. Therefore, we need to consider only the modules behind the exchanged ones for cost update.
- Except for handling soft modules, we only need to transform from a B*-tree to its corresponding placement during processing, which takes only linear time. In contrast, O-trees need to transform between a representation and its placement; sequence pairs only need to transform from a representation to a constraint graph, but it takes $O(n \lg n)$ time.
- Like the O-tree, there are only $O(n! 2^{2n-2}/n^{1.5})$ combinations for a B*-tree while a sequence pair has $(n!)^2$ combinations.

We summarize the advantages and disadvantages of the aforementioned representations in Table 1.

Represent.	Advantages (a) and Disadvantages (d)
Binary tree	a1. efficient
	a2. flexible to deal with hard, pre-placed, soft, and rectilinear modules, etc
	a3. smaller encoding cost
	a4. can operate on the tree directly, no need to do transformation during processing
	a5. can evaluate area cost incrementally
	a6. transformation between representation and placement takes linear time
	d1. can handle only the slicing structure
Sequence pair/ BSG	a1. can handle non-slicing structure
	a2. very flexible in representation
	d1. time-consuming
	d2. the solution space is large
	d3. sequence encoding cost is high
	d4. harder to transform between a sequence pair and a placement
	d5. sequence pair cannot handle soft modules directly
	d6. BSG incurs redundancies
O-tree	a1. can handle non-slicing structure
	a2. the solution space is smaller
	a3. transformation between representation and placement takes only linear time
	a4. encoded by fewer bits than sequence pair and BSG
	d1. less flexible than BSG/sequence pair in representation
	d2. tree structure is irregular, harder for implementation
	d3. need to encode and operate on module sequence
	d4. need to transform between the tree and its placement during processing
	d5. inserting positions are limited, might deviate from the optimal during solution perturbation
B*-tree	a1. can handle non-slicing structure
	a2. binary-tree based, efficient
	a3. flexible to deal with hard, pre-placed, soft, and rectilinear modules, etc
	a4. smaller encoding cost
	a5. except for handling soft modules, only need to transform from a tree to its placement during processing, which takes only linear time
	a6. can evaluate area cost incrementally
	a7. the solution space is smaller
	d1. less flexible than BSG/sequence pair in representation

Table 1: Representation comparison.

The quality of a representation can be evaluated based on three criteria: flexibility, efficiency, and effectiveness. We show the flexibility of the B*-tree representation by applying it to handle hard, pre-placed, soft, and rectilinear modules. To explore the efficiency and effectiveness, we compared with the O-tree [1] which is the fastest representation for non-slicing floorplans in the literature. We implemented the same iterative, deterministic algorithm used in [1]. (The work in [1] considers only hard modules.) Experimental results based on the MCNC benchmarks used in [1] show that the B*-tree runs about 4.5 times faster, consumes about 60% less memory, and results in smaller silicon areas than the O-tree. We also develop a B*-tree based simulated annealing scheme for floorplan design; the scheme achieves near optimum area utilization even for rectilinear modules.

The remainder of this paper is organized as follows. Section 2 formulates the floorplanning/placement problem. Section 3 proposes the B*-tree representation. Section 4 presents our approaches for handling hard, pre-placed, soft, and rectilinear modules. Section 5 describes our algorithm. Finally, experimental results are reported in Section 6.

2 Problem Definition

Let $B = \{b_1, b_2, ..., b_n\}$ be a set of n rectangular modules, and w_i, h_i, and a_i be the width, height, and area of b_i, $1 \leq i \leq n$, respectively. The aspect ratio of b_i is given by h_i/w_i. Let r_{min} and r_{max} be the minimum and maximum aspect ratios, i.e., $h_i/w_i \in [r_{min}, r_{max}]$. A placement $P = \{(x_i, y_i) | 1 \leq i \leq n\}$ is an assignment of the rectangular modules b_i's with the coordinates of their bottom-left corners being assigned to (x_i, y_i)'s so that no two modules overlap. We consider in this paper four kinds of modules: *hard, pre-placed, soft*, and *rectilinear modules*. A hard module is not flexible in its shape but free to rotate. A pre-placed module is inflexible in both its shape and coordinate. It must be located at a fixed position. A soft module is free to move and change its shape within the range $[r_{min}, r_{max}]$. A rectilinear module has an arbitrary shape. Usually, the goal of placement/floorplanning is to minimize the area and wirelength induced by the assignment of b_i's, where area is typically measured by the final enclosing rectangle of P and wirelength the summation of all module center-to-center interconnections. It will be clear later that traditional approaches for wirelength optimization readily apply to the B*-tree representation. We shall thus focus on area optimization in this paper.

3 Representations

In this section, we shall first introduce the O-tree representation [1] since we will adopt the concept of the admissible placement for O-trees and follow some of its notations. Then, we shall present the B*-tree representation.

3.1 The O-tree Representation

An O-tree is induced from an *admissible placement* defined in [1]. A placement is said to be *admissible* if and only if no module can shift left or down with other modules being fixed; i.e., all modules are compacted in both x and y directions. See Figure 2(a) for an admissible placement.

An O-tree is an ordered tree structure with an arbitrary number of branches (children) for each node. (Figure 2(b) shows the O-tree for the placement shown in Figure 2(a).) As shown in Figure 2(b), the branches are irregular; for example, node n_1 has only one child while n_0 has four. The irregular structure complicates tree operations and incurs sequence encoding. For example, the O-tree given in Figure 2(b) is encoded by the two tuple $(00100110101011000011110011, b_0 b_7 b_8 b_{11} b_9 b_{10} b_1 b_2 b_3 b_4 b_5 b_6)$ obtained by the DFS on the O-tree.

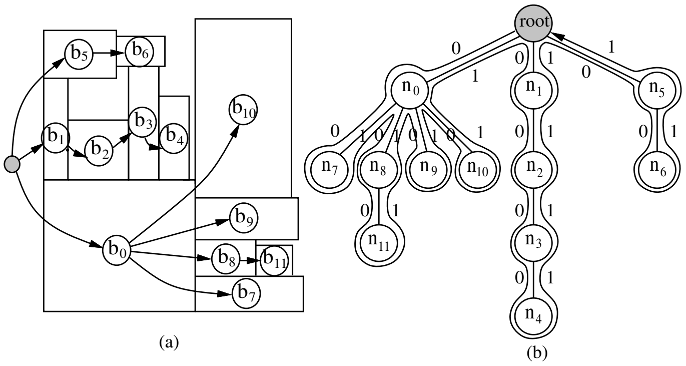


Figure 2: (a) An admissible placement. (b) The O-tree for the placement shown in (a).

The floorplan design algorithm presented in [1] is deterministic by perturbing an O-tree systematically. The perturbation procedure is to delete a module from the O-tree, and then insert it into the best position based on its evaluation. Due to the irregular structure, the candidate positions for inserting a node are limited to the external nodes (see Figure 3) to facilitate the update of the encoding tuple. Inserting a node to a position other than external positions makes the update of the encoding tuple difficult and time-consuming. The inflexibility might cause the O-tree to deviate from the optimal during solution perturbations, and thus inevitably limit the quality of a floorplan design. We consider this inflexibility a major drawback of the O-tree representation. It will be clear in the next subsection that this deficiency can be fixed by using B*-trees.

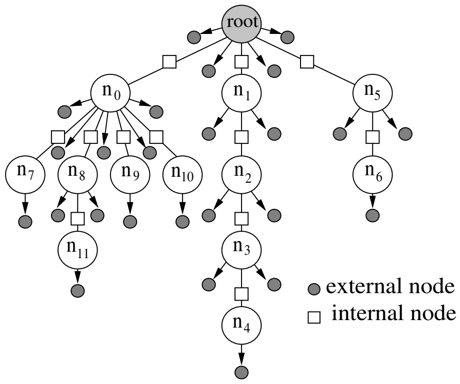


Figure 3: The internal and external insertion positions. To facilitate the update of the encoding tuple, O-trees allow a node to be inserted only at the external positions.

3.2 The B*-Tree Representation

Given an admissible placement P, we can represent it by a unique (horizontal) B*-tree T. (See Figure 1(b) for the B*-tree representing the placement shown in Figure 1(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 lowest module located above 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.

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 adjacent to 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 $(x_{root}, y_{root}) = (0, 0)$.

As shown in Figure 1, 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_7 the left child of n_0. Since the left child of n_7 does not exist, we then construct the right subtree of n_7 (which is rooted by n_8). 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 1(b) illustrates the resulting B*-tree for the placement shown in Figure 1(a). The construction takes only linear time.

Note that a B*-tree can also be constructed by transforming an O-tree to a binary tree, under several additional construction rules for some special module placements to make the theorem to be presented below hold. However, constructing a B*-tree in this way is not as efficient as the aforementioned procedure.

We have the following lemma which leads to Theorem 1.

Lemma 1 *For a module b_i in an admissible placement, the corresponding node n_i in its induced B*-tree has a unique parent if n_i is not the root.*

Theorem 1 *There is a 1-to-1 correspondence between an admissible placement and its induced B*-tree.*

In other words, for an admissible placement, we can construct a unique B*-tree, and vice versa. The nice property of the 1-to-1 correspondence between an admissible placement and its induced B*-tree prevents the search space from being enlarged with duplicate solutions.

4 Operations on Modules

We adopt the *contour* data structure presented in [1] to facilitate the operations on modules. The contour structure is a doubly linked list for modules, describing the contour curve in the current compaction direction. A *horizontal contour* (see Figure 4) can be used to reduce the running time for finding the y-coordinate of a newly inserted module. Without the contour, the running time for placing a new module would be linear to the number of modules. By maintaining the contour structure, however, the y-coordinate of a module can be computed in $O(1)$ time [1]. Figure 4 illustrates how to update the horizontal contour after inserting a new module.

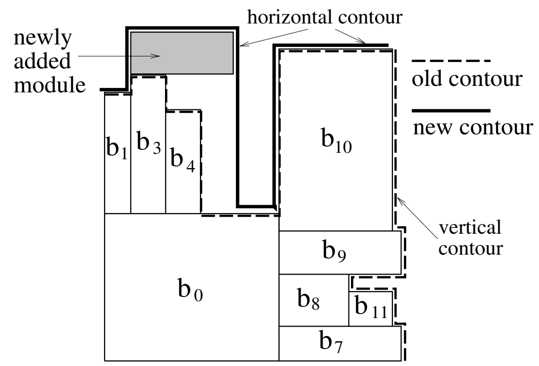


Figure 4: Adding a new module on top, we search the horizontal contour from left to right and update it with the top boundary of the new module.

To cope with rotated modules, when inserting a deleted node into a B*-tree, we can perform the operation twice at each position to find a better solution, one for the original orientation, and the other for the rotated one. In the following, we shall discuss the floorplanning problem with pre-placed, soft, and rectilinear modules.

4.1 Pre-placed Modules

Intuitively, if there exists a pre-placed module which cannot be located on its fixed positions during compaction, we would discard this solution. Unfortunately, this approach is not effective. A better approach is to change an infeasible solution to be feasible. Therefore, we handle the situation in this way: if there exists a pre-placed module that cannot be located on its fixed positions during compaction, we exchange the pre-placed module with another so that the pre-placed module can be located on its fixed positions. There are two subproblems to be solved: (1) how to choose the module that swaps with the pre-placed module, and (2) how to locate the pre-placed one on its fixed position.

We say a module to be *ahead* (*behind*) another if its bottom-left x-coordinate is smaller (larger) than that of another. Similarly, a module is *lower* (*higher*) than another if its bottom-left y-coordinate is smaller (larger) than that of another. A coordinate (x, y) *dominates* another coordinate (x', y'), denoted by $(x, y) \geq (x', y')$, iff $x \geq x'$ and $y \geq y'$. Let b_i be a pre-placed module, (x_i^f, y_i^f) denote its fixed coordinate, and $D_i = \{b_j | (x_i^f, y_i^f) \geq (x_j, y_j)\}$. If there are modules ahead or lower than b_i so that b_i cannot be placed at (x_i^f, y_i^f), we would exchange b_i with the module in D_i that is most close to (x_i^f, y_i^f).

To identify the modules in D_i, we first trace back T from the node n_i until detecting a node n_j whose corresponding module b_j is dominated by (x_i^f, y_i^f). We add n_j into D_i. Then, we trace down the subtree of n_j and add all modules dominated by (x_i^f, y_i^f) into D_i. Note that when we trace down the subtree of n_j, once we find a node n_l that is not dominated by (x_i^f, y_i^f), all the nodes in the subtree of n_l will not be dominated by (x_i^f, y_i^f). After D_i is determined, the module that is most close to (x_i^f, y_i^f) is chosen for exchange.

Figure 5(a) shows a placement with a pre-placed module b_6 and its fixed position (x_6^f, y_6^f). b_6 cannot be placed at (x_6^f, y_6^f) since the position has been occupied by module b_2. Therefore, we must choose a module for exchange with b_6. Following the aforementioned procedure, we have

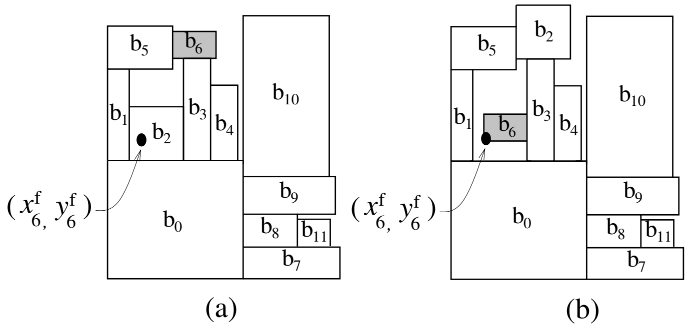


Figure 5: (a) A placement with a pre-placed module. (b) Place b_6 at its fixed position (x_6^f, y_6^f).

$D_6 = \{b_1, b_2\}$. Since b_2 is most close to (x_6^f, y_6^f), we exchange n_2 and n_6 in T and transform T to its corresponding placement. Figure 5(b) shows the resulting exchange of b_2 and b_6, with b_6 being placed at its fixed position (x_6^f, y_6^f). Note that, to save the execution time, we do not identify all feasible modules (e.g., b_0) for the exchange.

A nice property of the B*-tree lies in the incremental cost update. The DFS order of T before the exchange is ($n_0, n_7, n_8, n_{11}, n_9, n_{10}, n_1, n_2, n_3, n_4, n_5, n_6$). The positions of $b_0, b_7, b_8, b_{11}, b_9, b_{10}$, and b_1 remain unchanged after the exchange since they are in the front of b_2 in the DFS order. Therefore, we do not need to recompute the area cost from scratch.

4.2 Soft Modules

With its area being fixed, the width and height of a soft module is free to be changed under its aspect ratio constraint. Our method for handling soft modules consists of two stages: the first stage picks a soft module for processing (deleting and inserting a node associated with the module), and the second stage adjusts the shapes of the other soft modules. To insert a node, most existing work tries all possible combinations of widths and heights. To avoid the time-consuming process of trying all possible combinations, we shall find the best shape directly.

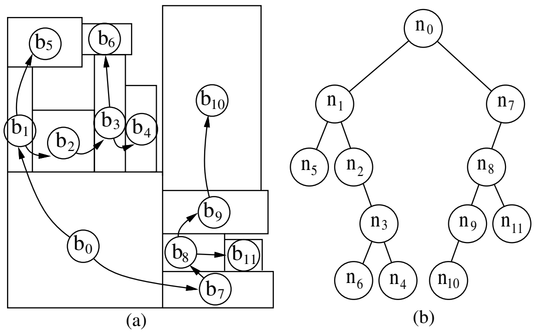


Figure 6: (a) An admissible placement. (b) The vertical B*-tree representing the placement.

To compact along the y direction, we introduce the *vertical B*-tree* T_v, and call the B*-tree described in Section 3.2 the *horizontal B*-tree*, denoted by T_h. Given an admissible placement P, we can construct a unique T_v. Let U_i denote the set of modules located above and adjacent to b_i. The construction is similar to constructing a T_h described in Section 3.2 with the only difference in direction. The left child of the node n_i corresponds to the leftmost module in U_i that is unvisited. The right child of n_i represents the module located on the right-hand side and adjacent to b_i, with its y-coordinate equal to that of b_i and its x-coordinate less than that of the right boundary of the module below and adjacent to b_i, if any. Figure 6(b) gives *the* vertical B*-tree for the admissible placement shown in Figure 6(a).

After inserting a node associated with a soft module, we compute its surplus space in the x and y directions for stretching its width and height under its aspect ratio constraint. Let A_i^H (A_i^V) denote the set of modules affected by shifting the soft module b_i in the horizontal (vertical) direction and b_i itself. (Note that we can shift a module either up or right in an admissible placement.) The modules in A_i^H are those associated with the nodes in the left subtree of b_i in T_h and b_i itself. The modules in A_i^V can be similarly defined in T_v.

Let $[y_{i,min}, y_{i,max}]$ ($[x_{i,min}, x_{i,max}]$) be the range on the vertical (horizontal) contour that could be affected by pushing b_i forward in the horizontal (vertical) direction. We have

$$y_{i,max} = \max\{y_k + h_k | b_k \in A_i^H\} \quad (1)$$

$$y_{i,min} = \min\{y_k | b_k \in A_i^H\} \quad (2)$$

and

$$x_{i,max} = \max\{x_k + w_k | b_k \in A_i^V\} \quad (3)$$

$$x_{i,min} = \min\{x_k | b_k \in A_i^V\}. \quad (4)$$

Let $\hat{x}_{I_i}$ ($\hat{y}_{I_i}$) be the maximum x (y) coordinate on the vertical (horizontal) contour in the range $I_i = [y_{i,min}, y_{i,max}]$ ($[x_{i,min}, x_{i,max}]$), and $\delta_H(i)$ ($\delta_V(i)$) denote the distance between $\hat{x}_{I_i}$ ($\hat{y}_{I_i}$) and the maximum x (y) coordinate on the vertical (horizontal) contour. Also, let O_i^R (O_i^U) be the set of modules on the right-hand (upper) side of b_i and their vertical (horizontal) boundaries overlap with the interval $[y_i, y_i + h_i]$ ($[x_i, x_i + w_i]$), $x_i^* = \min\{x_j | b_j \in O_i^R\}$ ($y_i^* = \min\{y_j | b_j \in O_i^U\}$) denote the minimum x (y) coordinate of the modules in O_i^R (O_i^U), and

$$\alpha_H(i) = \begin{cases} \delta_H(i) + x_i^* - x_i - w_i, & if\ O_i^R \neq \phi \\ \delta_H(i), & \text{otherwise,} \end{cases}$$

$$\alpha_V(i) = \begin{cases} \delta_V(i) + y_i^* - y_i - h_i, & if\ O_i^U \neq \phi \\ \delta_V(i), & \text{otherwise.} \end{cases}$$

We have

$$w_i = \begin{cases} w_i + \alpha_H(i), & if\ r_{min} \leq \frac{a_i}{(w_i + \alpha_H(i))^2} \leq r_{max} \\ \sqrt{\frac{a_i}{r_{min}}}, & if\ \frac{a_i}{(w_i + \alpha_H(i))^2} < r_{min} \\ \sqrt{\frac{a_i}{r_{max}}}, & if\ \frac{a_i}{(w_i + \alpha_H(i))^2} > r_{max}, \end{cases}$$

$$h_i = \begin{cases} h_i + \alpha_V(i), & if\ r_{min} \leq \frac{(h_i + \alpha_V(i))^2}{a_i} \leq r_{max} \\ \sqrt{a_i r_{min}}, & if\ \frac{(h_i + \alpha_V(i))^2}{a_i} < r_{min} \\ \sqrt{a_i r_{max}}, & if\ \frac{(h_i + \alpha_V(i))^2}{a_i} > r_{max}. \end{cases}$$

Modifying the shape of a module, we change its width first, then change its height after compacting along the y direction, and compact along the x direction at last. Changing width and height at the same time may cause confusion as there may be surplus space in both of the x and y directions.

After changing the shape of the inserted soft module, we then change those of other soft modules. We first compute the surplus space in the x and y directions for each soft module. Instead of computing the surplus space from the contour, we find the space between the soft module and its neighboring ones. After inserting a soft module b_i, we compute the surplus space of a soft module b_j in the x and y directions from O_j^R and O_j^U respectively.

Let $\beta_H(j)$ ($\beta_V(j)$) be the surplus space of b_j in the horizontal (vertical) direction, and $\hat{x}$ ($\hat{y}$) be the maximum x (y) coordinate on the vertical (horizontal) contour. We have

$$\beta_H(j) = \begin{cases} x_j^* - x_j - w_j, & if\ O_j^R \neq \phi \\ \hat{x} - x_j - w_j, & \text{otherwise,} \end{cases}$$

$$\beta_V(j) = \begin{cases} y_j^* - y_j - h_j, & if\ O_j^U \neq \phi \\ \hat{y} - y_j - h_j, & \text{otherwise.} \end{cases}$$

Substituting $\alpha_H(i)$ for $\beta_H(j)$ and $\alpha_V(i)$ for $\beta_V(j)$, we can obtain the new w_j and h_j similar to the equations shown above.

Similar to the processing of the inserted soft module, we change the widths and heights of the soft modules one at a time. After changing their widths, we compact in the vertical direction. Then we continue to change their heights and compact in the horizontal direction at last.

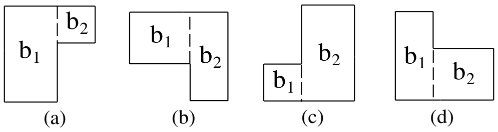


Figure 7: Four cases of an L-shaped module after rotation. Each is partitioned into two parts by slicing it along the middle vertical boundary.

4.3 Rectilinear Modules

We can partition a rectilinear module into several rectangular sub-modules by slicing from left to right along each vertical boundary. We first consider L-shaped modules and then extend to general rectilinear modules.

An L-shaped module b_L can be partitioned into two rectangular sub-modules by slicing b_L along its middle vertical boundary. As shown in Figure 7, b_1 and b_2 are the sub-modules of b_L, and we say $b_1, b_2 \in b_L$. During placement, we must guarantee that the two sub-modules b_1 and b_2 abut, and b_L maintains its original shape. To ensure that the left sub-module b_1 and the right sub-module b_2 of 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.

Since modules are placed in the DFS order, keeping b_2 as b_1's left child in the B*-tree guarantees that b_1 and b_2 are placed in sequence. The LC relation ensures that the x-coordinate of b_2's left boundary equals that of b_1's right boundary. For example, the two sets of sub-modules b_1, b_2 and b_3, b_4 shown in Figure 8(a) abut. The sub-modules b_3 and b_4 are placed at the correct positions while b_1 and b_2 are not since b_1 and b_2 do not conform to their original L shape. We say b_1 and b_2 to be *mis-aligned*. To slove the mis-aligned problem, we must adjust the y coordinate of b_1 or b_2. For the b_1 and b_2 shown in Figure 8(a), for example, b_1 must be pulled up until the y coordinate of b_1's top (bottom) boundary equals that of b_2's for the L shape shown in Figure 7(a) (Figure 7(d)).

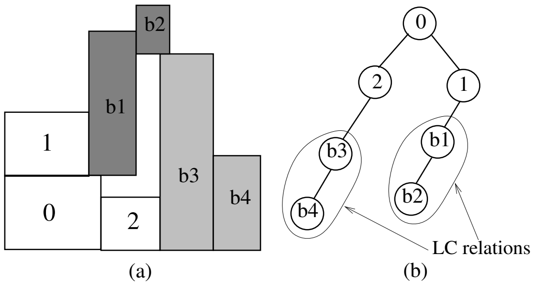


Figure 8: Suppose that b_1, b_2 and b_3, b_4 are two sets of sub-modules corresponding to two L-shaped modules. (a) A placement in which b_1, b_2 and b_3, b_4 abut. However, b_1 and b_2 are mis-aligned. (b) Their corresponding nodes in the B*-tree keep the LC relation between b_1 and b_2 (as well as b_3 and b_4).

For each L-shaped module, there are four orientations after rotation, as shown in Figure 7. Whenever we perform a rotation on an L-shaped module, we must repartition it into two sub-modules and maintain the LC relation between them. See Figure 7 for the sub-modules after repartitioning.

A rectilinear module is *convex* if any two points within the module can be connected by the shortest Manhattan path which also lies within the module; it is *concave* otherwise. A convex module b_C can be partitioned into a set of sub-modules $b_1, b_2, \ldots, b_n$ by slicing b_C from left to right along each vertical boundary. Considering the LC relation, we keep the sub-module 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 may need to adjust the y coordinates of the sub-modules, like what we did for L-shaped modules. For a concave module, we can fill the concave holes of the module and make it convex. Then, the operations remain the same as those for a convex module.

5 The Algorithm

Our floorplan design algorithm is based on the simulated annealing method [4]. The algorithm can consider not only hard modules, but also pre-placed, soft, and rectilinear ones.

We perturb a B*-tree (a feasible solution) to another B*-tree by using the following four operations.

- Op1: Rotate a module.
- Op2: Move a module to another place.
- Op3: Swap two modules.
- Op4: Remove a soft module and insert it into the best internal or external position.

We have discussed Op1 in Section 4. Op2 deletes and inserts a module. If the deleted node is associated with a rectangular module, we simply delete the node from the B*-tree. Otherwise, there will be several nodes associated with a rectilinear module, and we treat them as a whole and maintain their LC relations. Op4 deletes a soft module, tries all possible internal and external positions, inserts it into the best position, changes its shape and the shapes of the other soft modules (see Section 4.2). Op2, Op3, and Op4 need to apply the $Insert$ and $Delete$ operations for inserting and deleting a node to and from a B*-tree. We explain the two operations in the following.

5.1 Deletion

There are three cases for the deletion operation.

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

In Case 1, we simply delete the target leaf node. In Case 2, we remove the target node and then place its only child at the position of the removed node. The tree update can be performed in $O(1)$ time. In Case 3, we replace the target node 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. It is obvious that such a deletion operation requires $O(h)$ time, where h is the height of the B*-tree. Note that in Cases 2 and 3, the relative positions of the modules might be changed after the operation, and thus we might need to reconstruct a corresponding placement for further processing. Also, if the deleted node corresponds to a sub-module of a rectilinear module b_R, we should also delete other sub-modules of b_R.

5.2 Insertion

When adding a module, we may place it around some module, but not between the sub-modules that belong to a rectilinear module. We define three types of positions as follows.

- *Inseparable position:* A position between two nodes associated with two sub-modules of a rectilinear module.
- *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 module, we can insert it into an internal or an external position directly. For a rectilinear module b_R consisting of the sub-modules $b_1, b_2, \ldots, b_n$ ordered from left to right, the sub-modules must be inserted simultaneously, and b_{i+1} must be the left child of b_i to satisfy the LC relation.

The simulated annealing algorithm starts by randomly choosing an initial B*-tree. Then it perturbs a B*-tree (a feasible solution) to another B*-tree based on the aforementioned Op1–Op4 until a predefined "frozen" state is reached. At last, we transform the resulting B*-tree to the corresponding final admissible placement.

6 Experimental Results

We implemented two algorithms in the C++ programming language on a 200 MHz SUN Sparc Ultra-I workstation with 256 MB memory. One is the iterative, deterministic algorithm used in [1], and the other is the aforementioned simulated annealing algorithm. The deterministic one is served for the purpose of fair comparison with the O-tree representation which is the fastest for non-slicing floorplans in the literature.

We performed two sets of experiments: one was based on the MCNC benchmark circuits used in [1], and the other on some artificial rectilinear modules. Table 2 lists the names of the circuits, the numbers of modules, the runtimes *per iteration* for the iterative, deterministic algorithm, memory requirements, and chip areas for the cluster refinement algorithm [16], the O-tree, the B*-tree based on the iterative algorithm. We also tested the total runtimes and areas resulted from using the B*-tree based simulated annealing algorithm. The results show that the B*-tree achieved average speedups (area reductions) of 458 and 4.5 (1.0% and

Circuit	# of modules	Iterative algorithm									Simulated annealing	
		Cluster refinement			O-tree			B*-tree			B*-tree	
		Time/ite.* (sec)	Mem. (MB)	Area* (mm^2)	Time/ite. (sec)	Mem. (MB)	Area (mm^2)	Time/ite. (sec)	Mem. (MB)	Area (mm^2)	Tot. time (sec)	Area (mm^2)
apte	9	224.0	NA	48.40	0.63	4.18	47.76	0.11	1.67	46.92	7	46.92
xerox	10	18.8	NA	20.30	1.68	4.31	20.18	0.39	1.78	20.06	25	19.83
hp	11	18.0	NA	9.58	1.09	4.21	9.49	0.21	1.67	9.17	55	8.95
ami33	33	603.0	NA	1.21	25.40	4.67	1.25	7.83	1.82	1.27	3417	1.27
ami49	49	1860.0	NA	37.70	154.71	5.28	38.60	39.24	1.94	37.43	4752	36.80
Comp.		458	NA	1.010	4.512	2.546	1.014	1.000	1.000	1.000	-	0.989

Table 2: Comparisons for runtime per iteration and memory and area requirements among cluster refinement, O-tree, and B*-tree based on iterative algorithms (and overall runtimes and areas for the B*-tree based simulated annealing algorithm). **NA:** Not Available.

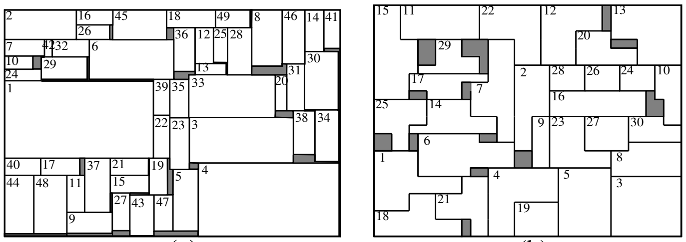
Circuit	#Rectangular modules	#L-shaped modules	#T-shaped modules	Optimum area	Resulting area	Dead space (%)	Runtime (sec)
test1	5	5	0	100	100	0.00	373
test2	10	10	0	400	405	1.25	1625
test3	15	15	0	900	928	3.11	3578
test4	7	7	6	400	414	3.50	2584
test5	10	10	10	900	945	5.00	6834

Table 3: Results for rectilinear modules using the B*-tree based simulated annealing method.

1.4%) over cluster refinement and O-trees, respectively. Further, the B*-tree reduced the average memory requirement by 60.4%, compared with O-trees. The results show the efficiency and effectiveness of the B*-tree representation.

Note that the results were based on hard modules since the work in [1] did not consider soft, pre-placed, or rectilinear modules. Also, the results were obtained by optimizing area alone for both the O-tree and B*-tree. The runtimes and areas for cluster refinement are directly taken from [1] whose experiments were performed on a 200 MHz SUN Sparc Ultra-I workstation with 512 MB memory; the configuration is almost the same as ours except that our machine has only 256 MB memory.

We also tested on several cases generated by cutting a rectangle into a set of rectangular, L-shaped, and T-shaped modules. For these cases, the optimum areas are given by their original rectangles. The results listed in Table 3 show that the B*-tree based simulated annealing algorithm obtained the optimum area for test1 and near optimum areas for test2, test3, test4, and test5 with areas only 1.25%, 3.11%, 3.50%, and 5.00% away from the optima, respectively. The runtimes for achieving the results ranged from about 6 minutes to 114 minutes. Note that the test cases are much more complicated than those used in recent work [2, 3, 17], and the algorithm is much more efficient. Figures 9(a) and (b) show the resulting placements for ami49 and test5.



(a) (b)

Figure 9: (a) Placement of ami49 (which consisting of only hard modules). The shaded regions represent dead spaces. (b) Placement of test5 with 10 rectangular, 10 L-shaped, and 10 T-shaped modules.

7 Concluding Remarks

We have presented the B*-tree representation for non-slicing floorplans and shown its superior efficiency, flexibility, and effectiveness. A key significance of this work lies in minimizing the gap between the representations for slicing and non-slicing floorplans. Compared with the binary trees for slicing floorplans, existing representations for non-slicing floorplans are much harder for implementation and operation and incur more restrictions. In contrast, B*-trees inherit most nice properties from binary trees and thus are very simple, efficient, and flexible for manipulating various types of modules and constraints directly and incrementally. More importantly, the properties make B*-trees a promising alternative to the floorplan design with new, more complicated considerations induced from the deep submicron technology, such as floorplanning with buffer module consideration, etc. Research in interconnect-driven floorplanning using B*-trees is ongoing.

Acknowledgments

We would like to thank Prof. Chung-Kuan Cheng and Dr. Pei-Ning Guo for providing us with the O-tree package and the benchmark circuits. Their work on the O-tree paves the way to the development of the B*-tree. Special thanks also go to the anonymous reviewers for their constructive comments.

References

[1] P.-N. Guo, C.-K. Cheng, and T. Yoshimura, "An O-Tree Representation of Non-Slicing Floorplan and Its Applications,"*Proc. DAC*, pp. 268–273, 1999.

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

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

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

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

[6] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani, "Rectangle-Packing Based Module Placement," *Proc. ICCAD*, pp. 472–479, 1995.

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

[8] H. Murata, Ernest S. Kuh, "Sequence Pair Based Placement Method for Hard/Soft/Pre-placed Modules," *Proc. ISPD*, pp. 167–172, 1998.

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

[10] 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.

[11] H. Onodera, Y. Taniquchi, K. Tamaru, "Branch-and-Bound Placement for Building Block Layout," *Proc. DAC*, pp. 433–439, 1991.

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

[13] C. H. Papadimitriou, and K. Steiglitz, *Combinatorial Optimization,* Prentice Hall, 1982.

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

[15] D. F. Wong, and C. L. Liu, "A New Algorithm for Floorplan Design," *Proc. DAC*, pp. 101–107, 1986.

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

[17] J. Xu, P.-N. Guo, and C.-K. Cheng, "Rectilinear Block Placement Using Sequence-Pair," *Proc. ISPD*, pp. 173–178, 1998.