Module placement with boundary constraints using B*-trees

J.-M. Lin, H.-E. Yi and Y.-W. Chang

Abstract: The module placement problem is to determine the co-ordinates of logic modules in a chip such that no two modules overlap and some cost (e.g. silicon area, interconnection length, etc.) is optimised. To shorten connections between inputs and outputs and/or make related modules adjacent, it is desired to place some modules along the specific boundaries of a chip. To deal with such boundary constraints, we explore the feasibility conditions of a B*-tree with boundary constraints and develop a simulated annealing-based algorithm using B*-trees. Unlike most previous work, the proposed algorithm guarantees a feasible B*-tree with boundary constraints for each perturbation. Experimental results show that the algorithm can obtain a smaller silicon area than the most recent work based on sequence pairs.

1 Introduction

As circuit complexity increases dramatically, hierarchical design and IP modules are widely used in modern VLSI design. This trend makes floorplanning/placement much more critical than ever. Floorplanning/placement is used to determine shapes and positions of modules to optimise circuit performance. Since geometric relations among modules are determined during floorplanning/placement, the results have a great impact on the quality and flexibility of a design, such as layout area, global routing structure, power consumption, etc. To facilitate floorplan design, we need a representation that can record the geometric relationship among modules, which can be manipulated efficiently, and can handle various constraints. Among the floorplanning/placement constraints, boundary constraints, which require some modules to be placed along the chip boundaries, are often concerned in the real design. There are at least two situations that motivate the consideration of boundary constraints:

- To shorten connections between inputs and outputs, it is desirable to place some modules along the specific boundaries of a chip.
- To deal with large circuits hierarchically, modules are grouped into units and module placement is performed independently for each unit. It is helpful if some modules are constrained to be placed along boundaries in each unit such that they can be adjacent to some other modules in the neighbouring units.

Therefore, it is desired to develop an efficient and effective approach to the floorplanning/placement problem with boundary constraints.

IEE Proceedings online no. 20020433

DOI: 10.1049/ip-cds:20020433

Paper first received 19th November 2001 and in revised form 25th March 2002

J.-M. Lin and H.-E. Yi are with the Department of Computer and Information Science, National Chiao Tung University, Hsinchu 300, Taiwan, Republic of China

Y.-W. Chang is with the Department of Electrical Engineering & Graduate Institute of Electronics Engineering, National Taiwan University, Taipei 106, Taiwan, Republic of China

1.1 Previous work

Floorplans are often handled based on their structures, the slicing structure [1, 2] and the non-slicing structure [3–8]. A slicing structure can be obtained by recursively cutting rectangles horizontally or vertically into smaller rectangles; otherwise, it is a non-slicing structure. For the slicing structure, Otten in [1] first used a binary tree to represent the slicing floorplan. Wong and Liu later in [2] proposed a normalised Polish expression to improve the binary tree-based representation. The slicing structure has smaller solution space, resulting in faster running time. However, the non-slicing structure is more general and often leads to a more area-efficient placement than the slicing one.

There are a few new non-slicing floorplan representations in the literature, e.g. sequence pair (SP) [7], bounded-sliceline grid (BSG) [8], O-tree [4], B*-tree [3], corner block list (CBL) [5] and transitive closure graph (TCG) [6]. Murata *et al.* in [7] used two sequences of module names, called SP, to represent the geometric relations among modules. Another representation, called BSG, was later proposed by Nakatake *et al.* [8]. Guo *et al.* [4] first proposed a tree-based representation, O-tree, for non-slicing floorplans. Chang *et al.* in [3] presented a binary tree-based representation, called B*-tree, and showed its superior properties for operations. Hong *et al.* in [5] proposed the CBL representation for non-slicing floorplans. Recently, Lin and Chang in [6] proposed a new representation, called TCG, by using a pair of transitive closure graphs.

The floorplan design with boundary constraints was first studied by Young and Wong [9]. They applied the normalised polished expression to handle the problem with a slicing floorplan. Recently, several approaches to the problem of non-slicing floorplans have been presented. Tang and Wong in [10] handled the constraint by adding dummy edges into the constraint graphs of SP; however, they just presented their idea without implementing their approach. Based on CBL, Ma *et al.* in [11] assigned a penalty to a misplaced boundary module and perturbed CBL to reduce the penalty. All the previous works in [9–11] cannot guarantee a feasible solution after solution perturbation and their final placements. Unlike the previous works, Lai *et al.* in [12] explored the feasibility conditions for SP with boundary constraints and transformed an infeasible solution into a feasible one to guarantee a feasible solution

in each perturbation. However, the method is very complex, and many rules are needed to cope with the constraints.

1.2 Our contribution

In this paper, we deal with the floorplan design with boundary constraints using the B*-tree representation because it has been proved to be a superior representation due to its simple, yet effective, binary tree structure. We first explore the feasibility conditions of a B*-tree with boundary constraints and develop a simulated annealing based algorithm using B*-trees. Unlike the previous works proposed by Young and Wong [9], Tang and Wong [10], and Ma *et al.* [11], our algorithm guarantees a feasible B*-tree with boundary constraints for each perturbation. Unlike the complicated rules using SP proposed by Lai *et al.* [12] to guarantee a feasible solution for each perturbation, our method is very simple and easy to implement. Experimental results show that our algorithm can obtain a smaller silicon area than the most recent work by Lai *et al.* [12].

2 Problem formulation

Given a set of modules $M=\{m_1, m_2, \ldots, m_n\}$, where each module m_i, $1 \leq i \leq n$, has a fixed area, and its width, height and area are denoted by w_i, h_i and A_i, respectively. Each module m_i is free to rotate. Let F denote those modules without any boundary constraints (i.e. modules in F are free to be placed anywhere in the final placement). Let T (L, B or R) denote a set of modules that are demanded to be placed along the top (left, bottom or right) boundary in the final placement. Therefore, we can divide M into five disjoint subsets F, T, B, L and R (i.e. $M=F\cup T\cup B\cup L\cup R$). T, B, L or R may be an empty set if there exists no module with such a boundary constraint in placement.

Fig. 1 gives an example of a placement with boundary-constrained modules. Module m_1 (m_2) denotes a left (top) boundary module. Fig. 1*a* shows an infeasible placement since m_1 is not placed along the left boundary while Fig. 1*b* gives a feasible placement since m_1 and m_2 are placed at the designated boundaries.

Let (x_i, y_i) denote the co-ordinate of the bottom-left corner of m_i, $1 \leq i \leq n$, on a chip. A placement P with boundary-constrained modules is an assignment of (x_i, y_i) for each m_i such that no two modules overlap and the modules in $T\cup B\cup L\cup R$ satisfy designated boundary constraints. The goal of a placement with boundary constraints is to minimise the total area (i.e. the minimum bounding rectangle of P). We do not consider the optimisation of interconnect wirelength in this paper.

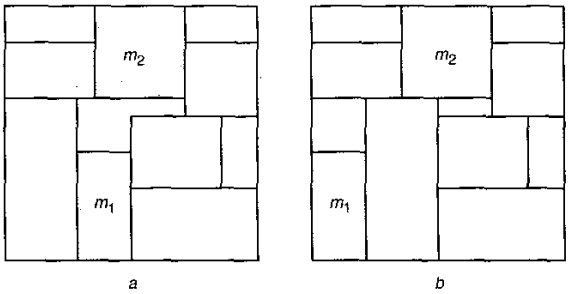

Fig. 1 *Placement with boundary-constrained modules*
Module m_1 (m_2) denotes a left (top) boundary module
a Infeasible placement because there exists a module at the left-hand side of m_1
b Feasible placement

However, it can be done easily by adding it into the cost function of our algorithm.

3 B*-tree representation

Before introducing our method, we shall first review the *B*-tree representation*. Chang *et al.* in [3] presented a binary tree-based representation for a left and bottom compacted placement, called B*-tree, and showed its superior properties for operations. Given a placement P, we can construct a unique (horizontal) B*-tree in linear time by using a recursive procedure similar to the depth first search (DFS) algorithm. (See Fig. 2*b* for the corresponding B*-tree of the placement shown in Fig. 2*a*.) Each node n_i in a B*-tree denotes a module. The root of a B*-tree corresponds to the module on the bottom-left corner. The left child n_j of a node n_i denotes the module m_j that is the lowest adjacent module on the right-hand side of m_i (i.e. $x_j=x_i+w_i$). The right child n_k of a node n_i denotes module m_k that is the lowest visible module above m_i and with the same x co-ordinate as m_i (i.e. $x_k=x_i$).

Figs. 2*a* and *b* show a placement and its corresponding B*-tree, respectively. The root n_0 of the B*-tree in Fig. 2*b* denotes that m_0 is the module on the bottom-left corner of the placement. For node n_3 in the B*-tree, n_3 has a left child n_4, which means that module m_4 is the lowest adjacent module in the right-hand side of module m_3 (i.e. $x_4=x_3+w_3$). n_7 is the right child of n_3 since module m_7 is the visible module over module m_3 and the two modules have the same x co-ordinate ($x_7=x_3$).

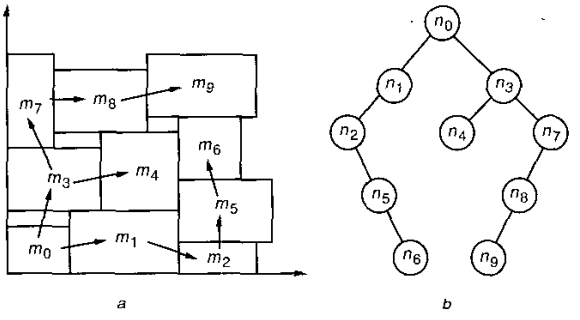

Fig. 2 *Placement (a) and corresponding B*-tree (b)*

We now show the procedure to get the placement from a B*-tree. We first introduce a contour structure, which is used by Guo *et al.* in [4]. The contour structure is a doubly linked list of modules, which describes the contour line in the current compaction direction. Without the contour structure, the runtime for placing a new module is linear to the number of modules. By maintaining the contour structure, the y co-ordinate for a newly inserted module can be computed in $O(1)$ time. Fig. 3 illustrates how to update the contour when we add a new module m_8 to the placement. The old contour is composed of modules m_7, m_3, m_4, m_6 and m_5. After m_8 is placed, the new contour becomes m_7, m_8, m_4, m_6 and m_5. Note that we only need to search modules m_3 and m_4 to get its y co-ordinate y_8 with the contour structure.

4 B*-tree for boundary-constrained modules

In this Section, we first explore the properties of a B*-tree with boundary constraints. We then present the feasibility conditions of a B*-tree with the constraint.

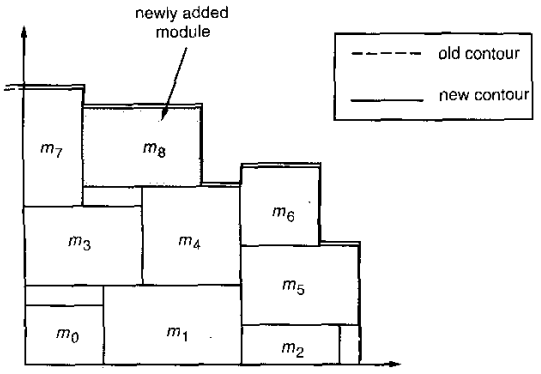

Fig. 3 *Adding a new module on top*
We search the contour from left to right and update it with the top boundary of the new module

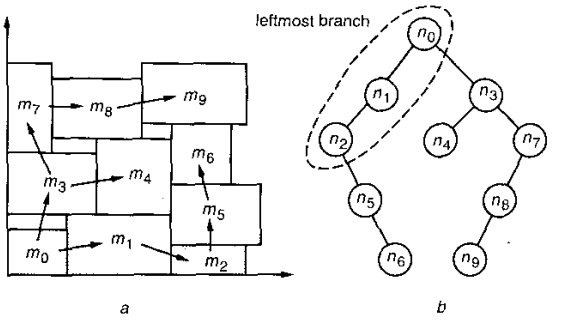

Fig. 4 *Bottom boundary modules (a) and corresponding nodes in the leftmost branch (b)*

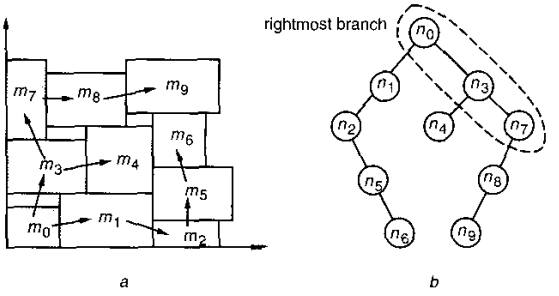

Fig. 5 *Left boundary modules (a) and corresponding nodes in the rightmost branch (b)*

4.1 Properties of B*-tree

The boundary-constrained modules are those modules that must be placed along boundaries in the final placement. A module can be placed along the bottom (left) boundary if there exists no module below (left to) the module in the final placement. Similarly, a module can be placed along the top (right) boundary if there exists no module above (right to) the module in the final placement. By the definition of a B*-tree, the left child n_j of a node n_i represents the lowest adjacent module b_j to the right of b_i (i.e. $x_j = x_i + w_i$). The right child n_k of n_i represents the lowest visible module b_k above b_i and with the same x co-ordinate as b_i (i.e. $x_k = x_i$). Therefore, we have the following four properties to guarantee that there exists no module below, left to, right to and above the module along the bottom, left, right and top boundaries, respectively.

Property 1: In a B*-tree, we have the properties for boundary constraints.

(i) The node corresponding to a bottom boundary module cannot be the right child of others.

(ii) The node corresponding to a left boundary module cannot be the left child of others.

(iii) The node corresponding to a right boundary module cannot have a left child.

(iv) The node corresponding to a top boundary module cannot have a right child.

4.2 Feasibility conditions of a B*-tree

The properties mentioned in the preceding Section must be satisfied to guarantee a feasible B*-tree with boundary-constrained modules. However, they only describe the necessary conditions for a B*-tree with the boundary constraints, that is, a module may not be placed along the designated boundary if the corresponding property is satisfied (see Fig. 2 for an example). Although node n_4 in Fig. 2*b* does not have a left (right) child, module m_4 is not placed at the right (top) boundary in Fig. 2*a*. To guarantee that modules are placed at designated boundaries, we propose sufficient conditions for a B*-tree with boundary constraints.

Let the leftmost branch (rightmost branch) of a B*-tree denote the path formed by the root and its leftmost (rightmost) descendants. See Fig. 4*b* (Fig. 5*b*) for the leftmost (rightmost) branch in a B*-tree. All nodes in the leftmost (rightmost) branch are not right (left) children of others; therefore, the nodes in the leftmost (rightmost) branch satisfy property 1 (property 2), which means that there exists no module below (left to) the corresponding module. By the definition of a B*-tree, module m_j should be adjacent and right to m_i if n_j is left child of the node n_i. The modules corresponding to the nodes in the leftmost branch should be placed at the bottom-left corner or right to the module placed at the bottom-left corner. Therefore, these modules must be placed along the bottom boundary. Similarly, m_k is the lowest visible module above m_i and with the same x co-ordinate as m_i if n_k is the right child of n_i. The modules corresponding to the nodes in the rightmost branch should be placed at the bottom-left corner or above and with the same x co-ordinate as the module placed at the bottom-left corner. Therefore, these modules must be placed along the left boundary. We thus have the following theorem for the feasibility conditions of a B*-tree with the bottom and the left constraints.

Theorem 1: Feasibility conditions

- Bottom-boundary condition: The nodes corresponding to the bottom boundary modules must be in the leftmost branch of a B*-tree.
- Left-boundary condition: The nodes corresponding to the left boundary modules must be in the rightmost branch of a B*-tree.

Figs. 4*a* and *b* show a placement and its corresponding B*-tree. Modules m_0, m_1 and m_2 in Fig. 4*a* are bottom boundary modules and the corresponding nodes n_0, n_1 and n_2 are in the leftmost branch of a B*-tree of Fig. 4*b*. Similarly, Figs. 5*a* and *b* show a placement with left boundary modules m_0, m_3 and m_7 and the corresponding B*-tree with nodes n_0, n_3 and n_7 in the rightmost branch.

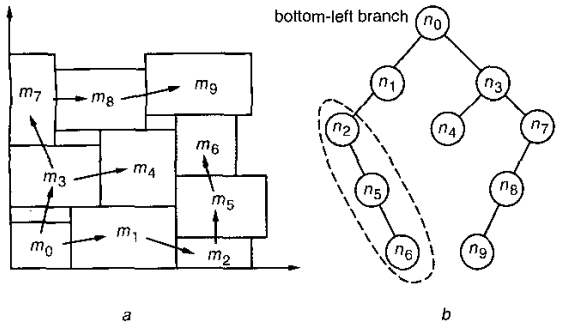

Fig. 6 *Right boundary modules (a) and corresponding nodes in the bottom-left branch (b)*

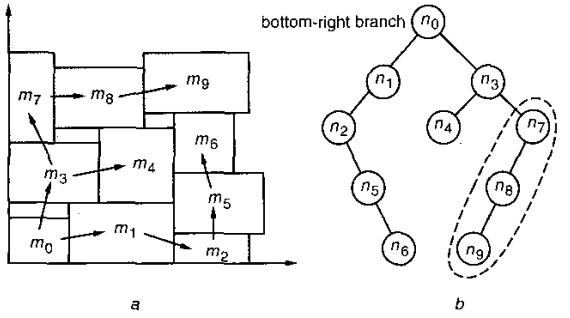

Fig. 7 *Top boundary modules (a) and corresponding nodes in the bottom-right branch (b)*

Let the bottom-left branch (bottom-right branch) of a B*-tree denote the path formed by the end of the leftmost (rightmost) branch and its rightmost (leftmost) descendants. See Fig. 6*b* (Fig. 7*b*) for the bottom-left (bottom-right) branch of a B*-tree. However, there may exist left (right) children for the nodes in the bottom-left (bottom-right) branch; therefore, the nodes in the branch may not satisfy property 3 (property 4). To guarantee that modules can be placed along the right (top) boundary, their left (right) children are deleted. By the definition of a B*-tree, the modules corresponding to the nodes in the bottom-left branch are placed at the bottom-right corner or above and with the same x co-ordinate as the module placed at the bottom-right corner. Further, no module is placed right to these modules since the left children for the nodes in the bottom-left branch are deleted. Similarly, the modules corresponding to the nodes in the bottom-right branch are placed at the top-left corner or right to the module at the top-left corner. Further, no module is placed above these modules since the right children of the nodes in the bottom-right branch are deleted. We thus have the following theorem for the feasibility conditions of a B*-tree with the right and the top boundary constraints.

Theorem 2: Feasibility conditions

- Right-boundary condition: For the right boundary modules, their corresponding nodes are in the bottom-left branch of a B*-tree with the left child for each node in the path being deleted.
- Top-boundary condition: For the top boundary modules, their corresponding nodes are in the bottom-right branch of a B*-tree with the right child for each node in the path being deleted.

Figs. 6*a* and *b* show a placement and the corresponding B*-tree. Modules m_2, m_5 and m_6 in Fig. 6*a* denote the right boundary modules and the corresponding nodes n_2, n_5 and n_6 are in the bottom-left branch of the B*-tree in Fig. 6*b*. Besides, n_2, n_5 and n_6 have no left child. Similarly, Figs. 7*a* and *b* show a placement with top boundary modules m_7, m_8 and m_9 and the corresponding B*-tree with nodes n_7, n_8 and n_9 in the bottom-right branch. It should be noted that m_9 is also a module along the right boundary, which cannot be identified by the right-boundary condition. To identify it, we shall find the last node in the bottom-right branch, which corresponds to the module at the top right corner.

5 The placement algorithm

Based on B*-trees, we develop a simulated annealing based algorithm [13] for handling the placement with boundary constraints. Given an initial B*-tree, the algorithm perturbs the B*-tree to get a new one. Then, the four feasibility conditions of B*-trees are checked. We transform an infeasible B*-tree into a feasible one if any condition is violated. The perturbation process repeats until predefined termination conditions are met. See Fig. 8 for the flow diagram of our algorithm.

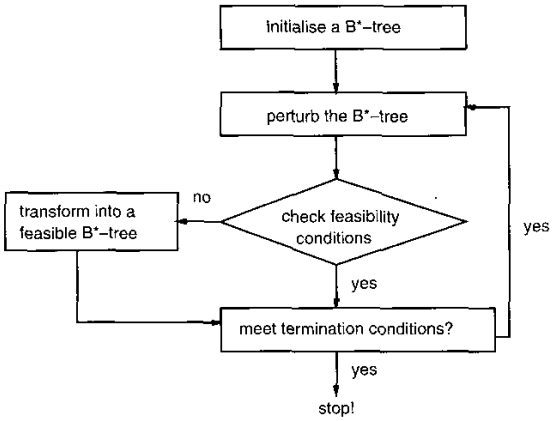

Fig. 8 *Flow diagram of the algorithm*

5.1 Solution perturbation

We apply the following three operations to perturb a B*-tree:

- Op1: rotate a module
- Op2: swap two modules
- Op3: move a module to another place.

Op1 only exchanges the width and height of a module without changing a B*-tree while Op2 and Op3 do. Only two nodes in a B*-tree are exchanged for Op2. The time complexities of Op1 and Op2 both take $O(1)$ time. However, the topology of a B*-tree is changed for Op3 since we need to delete and insert nodes into the B*-tree. The operations for deleting and inserting nodes are described in the following.

For node deletion, three types of nodes must be considered: leaf nodes, nodes with one child and nodes with two children. For a leaf node, it can be removed from a B*-tree directly without affecting other nodes. For a node with one child, it is replaced by its child. The subtree rooted by the child remains unchanged after the deletion. This tree

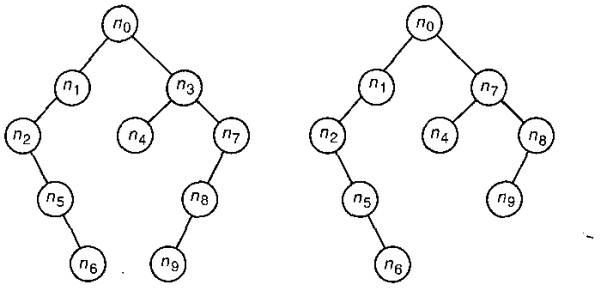

Fig. 9 *Example of deleting a node with two children*
a Node n_3 has two children
b B*-tree after deleting n_3

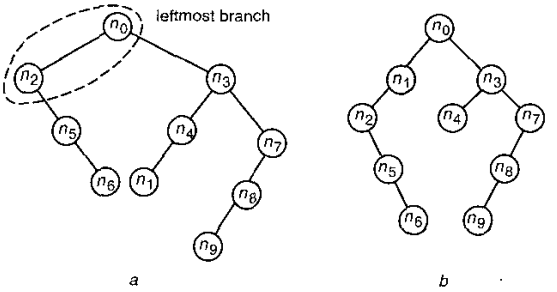

Fig. 10 *Example of handling a bottom boundary module*
a Infeasible B*-tree if node n_1 denotes a bottom boundary module but n_1 is not in the leftmost branch
b Feasible B*-tree where n_1 is inserted into the leftmost branch

update can be performed in $O(1)$ time. The process to delete a node with two children is a bit more complex. One of its two children is chosen to replace the target node. Then we move a child of the node to the position of the node. The procedure continues until the corresponding leaf node is processed. This operation takes $O(h)$ time, where h is the height of the B*-tree. See Fig. 9 for an example. The node n_3 in Fig. 9*a* is to be deleted from the B*-tree. Since n_3 has two children n_4 and n_7, we randomly choose node n_7 to replace n_3, and then use the child n_8 of n_7 to replace n_7, and so on. The resulting B*-tree is shown in Fig. 9*b*.

When we insert a node n_i into a B*-tree, we randomly choose a node n_j as its new parent. Then, n_i is inserted as the left (or right) child of n_j and the original left (or right) child of n_j becomes the left (or right) child of n_i. The operation takes $O(1)$ time. According to the above analysis, Op3 takes $O(n)$ time, where n is the number of modules.

5.2 Maintaining a feasible B*-tree

The feasibility condition of a B*-tree may be destroyed after perturbation. Therefore, we transform an infeasible B*-tree into a feasible one after perturbation.

The procedures to transform an infeasible B*-tree into a feasible one are described as follows. For bottom (left) boundary modules, let S_b (S_l) denote the set of the nodes in the leftmost (rightmost) branch in a given B*-tree. Those nodes corresponding to the bottom (left) boundary modules, not in S_b (S_l), are recorded in the set X_b (X_l). If $X_b \neq \emptyset$ ($X_l \neq \emptyset$), each node $n \in X_b$ ($n \in X_l$) will be deleted from the current position and randomly inserted into the leftmost (rightmost) branch, which takes linear time. For example, Fig. 10*a* shows an infeasible B*-tree if node n_1 represents a bottom boundary module but $n_1 \notin S_b$. To get a feasible B*-tree, we delete n_1 from the B*-tree and insert it into the leftmost branch. The resulting B*-tree is shown in Fig. 10*b*.

For right (top) boundary modules, the procedure to transform an infeasible B*-tree is a bit more complex than the two procedures described above. Let S_r (S_t) denote the set of the nodes in the bottom-left (bottom-right) branch of a B*-tree. Let X_r (X_t) denote the set of nodes corresponding to right (top) boundary modules but are not in S_r (S_t). If $X_r \neq \emptyset$ ($X_t \neq \emptyset$), we delete each node $n \in X_r$ ($n \in X_t$) from a B*-tree and insert it into the bottom-left (bottom-right) branch, which takes linear time. Further, to guarantee a feasible B*-tree during perturbation, we do not move nodes to the left (right) children of the nodes in the bottom-left (bottom-right) branch.

6 Experimental results

We implemented our algorithm in the C++ programming language on a 200 MHz SUN Ultra I workstation with 256 MB memory. We compared our algorithm with the sequence-pair-based algorithm used in [12] based on the MCNC benchmark circuits listed in Table 1. Columns 1, 2, 3, 4 and 5 in the Table give the respective names of circuits, numbers of modules, numbers of top-boundary modules (denoted by T), numbers of bottom-boundary modules (denoted by B), numbers of left-boundary modules (denoted by L) and numbers of right-boundary modules (denoted by R). Note that the constrained modules in each circuit are the same as that used in [12] for the purpose of fair comparison.

Table 1: Information on test circuits

Circuit	No. of modules	No. of T modules	No. of B modules	No. of L modules	No. of R modules
apte	9	1	1	1	1
xerox	10	1	1	1	1
hp	11	1	1	1	1
ami33	33	2	2	2	2
ami49	49	3	3	2	3

The area and runtime comparisons between the sequence-pair-based algorithm [12] and ours are listed in Table 2. (Note that the sequence-pair-based algorithm was implemented on a Pentium-II 350 processor with 128 MB RAM.) As shown in Table 2, our algorithm results in an average dead space of 15.37%, compared to 18.24% reported by the sequence-pair-based algorithm. Also, our algorithm is quite efficient. Figs. 11 and 12 show the resulting placements for xerox and ami33 with the boundary-constrained modules shaded.

7 Conclusions

We have explored the feasibility conditions of a B*-tree with boundary constraints and developed a simulated annealing based algorithm using B*-trees. Also, we have proposed an efficient procedure to transform an infeasible solution into feasible one if the feasibility constraints are violated. Unlike most previous works, our algorithm guarantees a feasible B*-tree with boundary constraints in each perturbation.

Table 2: Area and runtime comparisons between the sequence-pair-based algorithm (on a Pentium-II 350 PC with 128 MB RAM) and our algorithm (on a 200 MHz SUN Ultra I workstation with 256 MB)

Circuit	Total area of modules	Sequence-pair			B*-tree		
		Resulting area mm^2	Dead space %	Runtime s	Resulting area mm^2	Dead space %	Runtime s
apte	46.56	46.92	0.77	15	46.92	0.77	19
xerox	19.32	20.96	5.59	19	19.91	3.05	22
hp	8.92	9.24	3.59	23	9.27	3.92	38
ami33	1.16	1.21	4.31	287	1.20	3.45	144
ami49	35.43	36.84	3.98	584	36.91	4.18	324
Total			18.24			15.37	

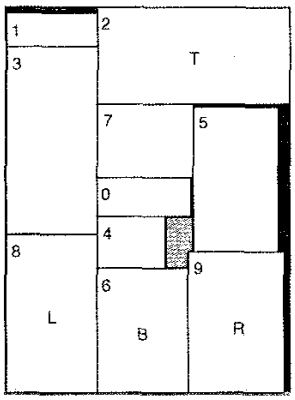

Fig. 11 *Placement result of xerox, where $T = \{2\}$, $B = \{6\}$, $L = \{8\}$ and $R = \{9\}$*
Area is 19.91 mm^2, and the dead space is 3.05%

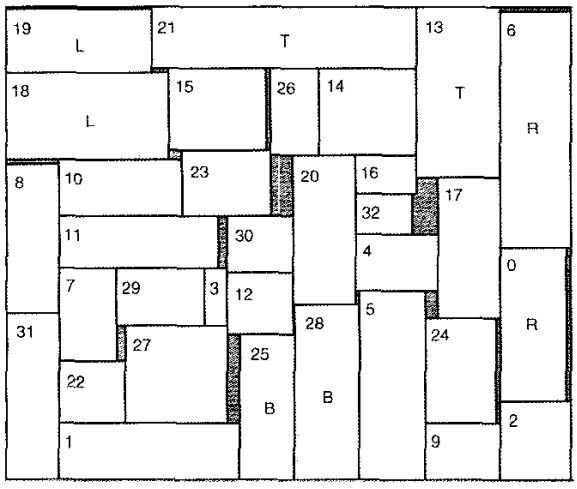

Fig. 12 *Placement result of ami33, where $T = \{13, 21\}$, $B = \{25, 28\}$, $L = \{18, 19\}$ and $R = \{0, 6\}$*
Area is 1.2 mm^2 and the dead space is 3.45%

Further, our algorithm is very simple and can be implemented easily. In particular, the operations and packing of a B*-tree take only linear time. Experimental results have shown that our algorithm can obtain smaller silicon area than the most recent work based on sequence pairs and consumes less running time.

8 Acknowledgment

This research was partially supported by the National Science Council of Taiwan ROC under grant NSC-89-2215-E-009 117.

9 References

1 OTTEN, R.H.J.M.: 'Automatic floorplan design'. Proceedings of 19th Design Automation Conference, DAC'82, USA, June 1982, pp. 261–267
2 WONG, D.F., and LIU, C.L.: 'A new algorithm for floorplan design'. Proceedings of 23rd Design Automation Conference, DAC'86, USA, June 1986, pp. 101–107
3 CHANG, Y.-C., CHANG, Y.-W., WU, G.-M., and WU, S.-W.: 'B*-Trees: A new representation for non-slicing floorplans'. Proceedings of 37th Design Automation Conference, DAC'00, CA, USA, June 2000, pp. 458–463
4 GUO, P.-N., CHENG, C.-K., and YOSHIMURA, T.: 'An O-tree representation of non-slicing floorplan and its applications'. Proceedings of 36th Design Automation Conference, DAC'99, LA, CA, USA, June 1999, pp. 268–273
5 HONG, X., HUANG, G., CAI, T., GU, J., DONG, S., CHENG, C.-K., and GU, J.: 'Corner block list: An effective and efficient topological representation of non-slicing floorplan'. Proceedings of International Conference on Computer aided design, ICCAD'00, CA, USA, Nov. 2000, pp. 8–12
6 LIN, J.-M., and CHANG, Y.-W.: 'TCG: A transitive closure graph-based representation for non-slicing floorplans'. Proceedings of 38th Design Automation Conference, DAC'01, NV, USA, June 2001, pp. 764–769
7 MURATA, H., FUJIYOSHI, K., NAKATAKE, S., and KAJITANI, Y.: 'Rectangle-packing based module placement'. Proceedings of International Conference on Computer aided design, ICCAD'95, CA, USA, Nov. 1995, pp. 472–479
8 NAKATAKE, S., FUJIYOSHI, K., MURATA, H., and KAJITANI, Y.: 'Module placement on BSG-structure and IC layout applications'. Proceedings of International Conference on Computer aided design, ICCAD'96, CA, USA, Nov. 1996, pp. 484–491
9 YOUNG, F.Y., and WONG, D.F.: 'Slicing floorplans with boundary constraint'. Proceedings of Asia and South Pacific Design Automation Conference, ASP-DAC'99, Yokohama, Japan, Feb. 1999, pp. 17–20
10 TANG, X., and WONG, D.F.: 'FAST-SP: A fast algorithm for block placement based on sequence pair'. Proceedings of Asia and South Pacific Design Automation Conference, ASP-DAC'01, Yokohama, Japan, Jan. 2001, pp. 521–526
11 MA, Y., DONG, S., HONG, X., CAI, Y., CHENG, C.-K., and GU, J.: 'VLSI floorplanning with boundary constraints based on corner block list'. Proceedings of Asia and South Pacific Design Automation Conference, ASP-DAC'01, Yokohama, Japan, Jan. 2001, pp. 509–514
12 LAI, J., LIN, M.-S., WANG, T.-C., and WANG, LI.-C.: 'Module placement with boundary constraints using the sequence-pair representation'. Proceedings of Asia and South Pacific Design Automation Conference, ASP-DAC'01, Yokohama, Japan, Jan. 2001, pp. 515–520
13 KIRKPATRICK, S., GELATT, C.D., and VECCHI, M.P.: 'Optimization by simulated annealing', *Science*, 1983, **220**, (4598), pp. 671–680