Placement with Alignment and Performance Constraints Using the B*-tree Representation [*]

Meng-Chen Wu[1] *and Yao-Wen Chang*[2]

[1] Institute of Electronics, National Chiao Tung University, Hsinchu 300, Taiwan

[2] Department of Electrical Engineering & Graduate Institute of Electronics Engineering, National Taiwan University, Taipei 106, Taiwan

Abstract

To facilitate sequential data transfer (e.g., bus or pipeline signals) and reduce bounded net delay (as well as total wirelength), it is desired to align circuit blocks one by one and constrain the blocks within a certain bounding box. In this paper, we handle the placement with alignment and performance (delay) constraints using the B*-tree representation. We first explore the feasibility conditions with the alignment and performance constraints, and then propose algorithms that can guarantee a feasible placement with alignment constraints and generate a good placement with performance constraints during each operation. In particular, our method is the first algorithm to achieve the amortized linear-time complexity for evaluating a placement with the alignment and performance constraints. Experimental results based on the MCNC benchmark with the constraints show that our method significantly outperforms the previous work.

1 Introduction

Due to the growth in design complexity with continued technology scaling, circuit sizes are getting much larger. To deal with the increasing complexity, hierarchical design and IP blocks are widely used. This trend makes block floorplanning/placement much more critical to the quality of a modern VLSI design. To address particular design requirements, we can impose corresponding position constraints for the circuit blocks in a floorplan/placement. Among the floorplanning/placement constraints, alignment and performance constraints have received increasing attention recently [11, 15]. The alignment constraint requires that blocks abut one by one in an alignment range while the performance constraint place blocks in a given bounding box to reduce critical nets delay (and total wirelength as well). The following give major reasons that motivate the considerations of these constraints:

- Smooth data transfer in a bus structure or a pipeline. Since alignment blocks abut one by one in a pre-defined range, we can set the bus width be the alignment range to facilitate the routing.
- Minimize delay on some critical nets. Block placements are often performed independently for each block. It is helpful if we place some blocks near to each other to reduce critical net delay.

Therefore, it is desirable to find an efficient and effective way to handle the floorplanning/placement problem with the alignment and performance constraints.

1.1 Previous Work

The floorplanning problem is often studied by the structures of floorplans, say the *slicing structure* and the *non-slicing structure*. For a slicing structure, Otten [6] presented a binary tree representation whose leaves denote blocks and internal nodes describe horizontal or vertical merging of the two children. Wong and Liu [12] proposed a normalized Polish expression for slicing floorplans. Although the slicing structure can be handled more easily and efficiently, real designs are often non-slicing.

The modelings for the non-slicing structures (and also the slicing structures) have been proposed recently. Among them, sequence pair [4], BSG [5], and TCG [3] specify the topological relative positions between blocks. From these representations, we can easily identify the geometric relation between two blocks, such as below, above, left-to and right-to. O-tree [2] and B*-tree [1] give partial topological relations among blocks by using ordered trees. In the trees, a node represent a block and an edge describes the relation between the parent node and the child node. Q-sequence [8] and twin binary tree [14] are coding schemes for modeling *mosaic floorplans*, in which each room contains exactly one block.

Problems related to floorplanning/placement with the performance and alignment constraints, such as pre-placed and range constraints and rectilinear blocks, were studied earlier. The floorplan design with the pre-defined range constraint was proposed by Young and Wong [15]. Unlike pre-placed blocks, the constrained blocks are required to place within a pre-defined range. They handle the slicing floorplan design with the range constraint based on the normalized Polished expression.

For rectilinear blocks, most previous works partition a rectilinear block into a set of sub-blocks and then process the sub-blocks one by one and at the same time try to maintain the original rectilinear shape. For example, Chang *et al.* in [1] applied the B*-tree to handle this constraint by keeping a *location constraint* between nodes associated with the sub-blocks of a rectilinear block. Similar operations can be found in Pang *et al.* [7] based on the O-tree and Xu *et al.* [13] based on sequence pair.

Tang and Wong [11] recently extended the range constraint to the performance constraint, and the rectilinear constraint to the alignment constraint. With the performance constraint, blocks are placed in a movable bounding box to reduce the net delay and minimize the total wire length as well. Different from the rectilinear constraint that must maintain the predefined rectilinear shape, the alignment constraint describes that some blocks are aligned in a row without fixed relative positions. They solved the placement with the two constraints using FAST-SP [10].

1.2 Our Contribution

In this paper, we handle the floorplanning/placement with the alignment and performance constraints using the B*-tree representation. We first explore the feasibility conditions with the alignment and performance constraints, and then propose algorithms that can guarantee a feasible placement with alignment constraints and generate a good placement with performance constraints during each operation. In particular, our method is the first algorithm to achieve the amortized $O(n)$-time complexity for evaluating a placement with the alignment and performance constraints (compared to the $O(n \lg \lg n)$-time algorithm proposed by Tang and Wong in [11]), where n is the number of blocks. (Note that both works assume that the number of groups of modules with alignment and performance constraints is constant.) Experimental results based on the MCNC benchmark with the constraints show that our method significantly outperforms the previous work; for example, our method achieved an average deadspace of only 3.2%, compared to that of 5.8% by [11].

The remainder of this paper is organized as follows. Section 2 reviews B*-trees. Section 3 gives the definitions of the alignment and performance constraints. The methods for solving these constraints are proposed in Section 4. Section 6 presents our algorithm. Experimental results are reported in Section 7. Finally, we give conclusions in Section 8.

2 Preliminaries

B*-trees [1] are an ordered binary tree for modeling a compacted floorplan. Given an *admissible placement* [2] (in which no module can move left or bottom), we can construct a *unique* B*-tree in linear time. Further, given a B*-tree, we can obtain a placement by packing the blocks in amortized linear time with a contour structure [2].

Given an admissible placement P, we can represent it by a unique B*-tree T. (See Figure 1(b) for the B*-tree representing the placement of Figure 1(a).) A B*-tree is an ordered binary tree whose root corresponds to the block 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 blocks located on the right-hand side and adjacent to b_i. The left child of the node n_i corresponds to the lowest block in R_i that is unvisited. The right child of n_i represents the lowest block located above and with its x-coordinate equal to that of b_i.

Based on the definition, the root of T represents the block on the bottom-left corner, and thus the x- and y-coordinates of the block associated with the root $(x_{root}, y_{root}) = (0, 0)$. If node n_j is the left child of node n_i, block b_j is placed on the right-hand side and adjacent to block b_i in the placement; i.e., $x_j = x_i + w_i$. Otherwise, if node n_j is the right child of n_i, block b_j is placed above block b_i, with the x-coordinate of b_j equal to that of b_i; i.e., $x_j = x_i$. With the contour structure, we can compute the y-coordinate of a block in amortized constant time.

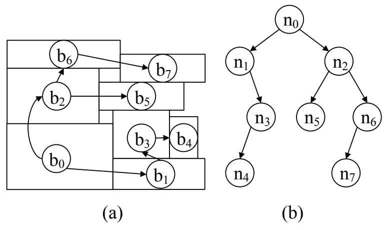

Figure 1: (a) An admissible placement. (b) The B*-tree representing the placement.

3 Problem Formulation

In this section, we first define the alignment and the performance constraints and then formulate the placement problem.

[*] This work was partially supported by the National Science Council of Taiwan under Grant No. NSC-92-2215-E-002-018. E-mail: kenwu@eda.ee.nctu.edu.tw; ywchang@cc.ee.ntu.edu.tw

3.1 Alignment Constraint

The alignment constraint is for placement with bus structures or pipelines. Unlike abutment that only requires the constrained blocks to be adjacent, alignment blocks are placed in an alignment range. Different from rectilinear blocks with fixed relative positions, alignment blocks are flexible to move within the pre-specified alignment range. Specifically, alignment blocks must abut one by one horizontally or vertically and aligned in a pre-defined range like a bus width. Therefore, alignment can be classified into H-alignment and V-alignment according to the orientation of blocks, horizontal and vertical, respectively. Given an alignment range r and a set of m blocks, $b_i, i = 1, 2, ..., m$, whose width, height, and the coordinate of bottom-left corner are denoted by $w_i, h_i, (x_i, y_i)$, $i = 1, 2, ..., m$, respectively. We follow the definitions given in [11] as follows.

Definition 1 (H-alignment) *The m blocks are* H-aligned *iff $x_i + w_i = x_{i+1}, 1 \leq i \leq m-1$ (abutting one by one) and $y_{max} + r \leq y_i + h_i, 1 \leq i \leq m$, where $y_{max} = max\{y_i | i = 1, 2, ...m\}$ (aligning horizontally).*

Definition 2 (V-alignment) *The m blocks are* V-aligned *iff $y_i + h_i = y_{i+1}, 1 \leq i \leq m-1$ (abutting one by one) and $x_{max} + r \leq x_i + w_i, 1 \leq i \leq m$, where $x_{max} = max\{x_i | i = 1, 2, ...m\}$ (aligning vertically).*

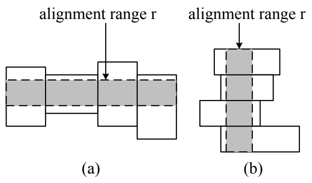

Figure 2: (a) Blocks with the H-alignment constraint. (b) Blocks with the V-alignment constraint.

Figures 2(a) and (b) show two sets of blocks with the H-alignment and the V-alignment constraints, respectively. The shaded regions illustrate the alignment ranges. Due to the similarity between the H-alignment and the V-alignment constraints, we can transform the V-alignment constraint into the H-alignment constraint during processing. We shall focus our discussions on the H-alignment constraint, unless specified otherwise.

3.2 Performance Constraint

For very deep submicron VLSI design, the interconnect delay dominates the circuit performance. Therefore, it is desirable to minimize the critical net delay to optimize circuit performance. The performance constraint is intended for this purpose; the constraint requires designated nets (blocks) to be placed within a predefined bounding box nets. Imposing the performance constraint, we can optimize not only the circuit delay, but also the total wire length. Here gives the definition.

Definition 3 (Performance constraint) *Given a set of blocks and the performance constraints, minimize total wire length and place designated blocks in a bounding box such that the critical net delay satisfy performance constraints at the same time.*

3.3 Problem Definition

Let $B = \{b_1, b_2, ..., b_n\}$ be a set of n rectangular blocks whose respective width, height, and area are denoted by W_i, H_i, and A_i, $1 \leq i \leq n$. Let (x_i, y_i) denote the coordinate of the bottom-left corner of block b_i, $1 \leq i \leq n$, on a chip. A placement $\mathcal{P}$ with the alignment and the performance constraints is an assignment of the coordinate (x_i, y_i) for each b_i, $1 \leq i \leq n$, such that no two blocks overlap and the given constraints are satisfied. The goal of floorplanning/placement is to optimize a predefined cost metric, such as the area (the minimum bounding rectangle of $\mathcal{P}$), induced by the assignment of b_i's on the chip.

4 Placement with Alignment Constraints

In this section, we first present the solutions for alignment constraints with B*-trees. Then, we propose the feasibility conditions for the B*-tree with the constraints.

4.1 B*-trees with Alignment Blocks

The alignment blocks have two properties: (1) Alignment blocks must abut one by one; (2) These blocks have to be located in an alignment range. First, we give solutions for abutment placement. For a B*-tree, the left child n_j of the node n_i represents the lowest adjacent block b_j which is right to block b_i (i.e. $x_j = x_i + w_i$). Therefore, blocks can abut one by one if their corresponding nodes form a left-skewed sub-tree in a B*-tree. An an example shown in Figure 1, the four sets of abutment blocks b_0 and b_1, b_3 and b_4, b_2 and b_5, and b_6 and b_7 correspond to four left-skewed sub-trees.

Property 1 *In a B*-tree, the nodes in a left-skewed sub-tree may correspond to a set of abutment blocks.*

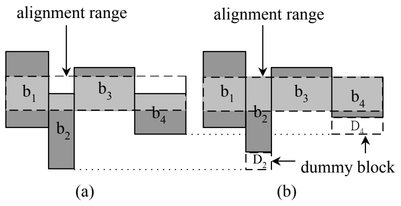

Figure 3: (a) An infeasible placement with blocks falling out of the alignment range; block b_2 and b_4 are not in the alignment range. (b) Inserting dummy blocks, we obtain a feasible placement without any block violating the alignment constraint.

After packing, blocks are compacted to the bottom and left. The blocks associated with a left-skew sub-tree of a B*-tree may be aligned together if no block falls down during packing. To solve the falling down problem, we introduce *dummy blocks* to fix it.

A dummy block comes for an alignment block. The dummy block has the same x-coordinate with the alignment block and right below it. The width of the dummy block is equal to its corresponding alignment block, and its height can be adjusted to make a displaced alignment block shift into the right alignment range. As illustrated in Figure 3(a), the blocks b_2 and b_4 fall out of the alignment range. As shown in Figure 3(b), we adjust the heights of the two dummy blocks to shift the displaced alignment blocks into the correct alignment range. After adjusting the heights of the dummy blocks, we can guarantee that the resulting placement is feasible with the alignment constraints.

Property 2 *Inserting a dummy block of an appropriate height, we can guarantee a feasible placement for the corresponding alignment block.*

4.2 Feasibility Condition for Alignment Constraints

The properties mentioned in the preceding section provide the way to develop the feasibility condition of a B*-tree with the alignment constraints. Consequently, we can take advantage of the feasibility condition to transform an infeasible placement to a feasible one of alignment blocks.

Given a B*-tree, we refer to the node representing an alignment block as an *alignment node*. For each alignment node, we introduce a *dummy node* in the B*-tree and make the alignment node the right child of its corresponding dummy node. By the definition of the B*-tree, this will make a dummy block right under its corresponding alignment block, and we can thus adjust the height of the dummy block to change the y-coordinate of the alignment block, if needed. We refer to a set of an alignment node and its corresponding dummy node as a *cluster node*, i.e., each cluster node consists of an alignment node and a dummy node. To make a set of alignment blocks abut one by one, by Property 1, we further require that the corresponding cluster nodes form a left-skewed sub-tree. We say that the cluster nodes form an *alignment shape* iff they form a left-skewed sub-tree. As an example shown in Figure 4(a), the three cluster nodes c_3, c_4, and c_5 form a left-skewed sub-tree and thus an alignment shape, for which the corresponding placement for the alignment blocks b_3, b_4, and b_5 can abut one by one, as shown in Figure 4(b). In Figure 4(b), the placement is obtained by packing the blocks corresponding to the B*-tree of Figure 4(a) and adjusting the heights of the dummy blocks to satisfy the alignment constraints. We have the following theorem for the feasibility condition of a B*-tree with alignment constraints.

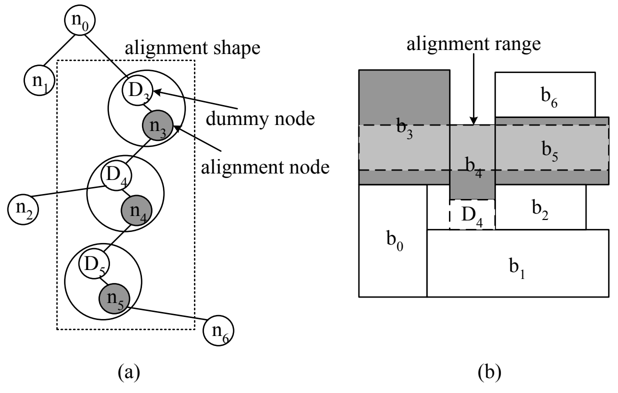

Figure 4: (a) The alignment shape in a B*-tree (b) The corresponding placement of (a)

Theorem 1 *There exists a feasible placement with alignment constraints if the alignment nodes in a B*-tree form an alignment shape.*

To deal with the alignment constraints, we need two passes to pack blocks correctly. In the first pass, we compute the coordinate for each non-dummy block (a regular block or an alignment block). Then, we verify whether every alignment block is in the alignment range. If there is any violation of the alignment constraints, we compute in the second pass the minimum movement (height) for the corresponding alignment (dummy) block to shift into the alignment range.

Given m alignment blocks and the alignment range r, the equation for computing the minimum movement (height) Δ_i for the alignment block $b_i, i = 1, 2, ..., m$, in H-alignment is as follows:

$$\Delta_i = \begin{cases} (y_{max} + r) - (y_i + h_i) & \text{if } (y_{max} + r) > (y_i + h_i) \\ 0 & \text{otherwise} \end{cases}$$

where

$$y_{max} = max\{y_i | i = 1, 2, ..., m\}.$$

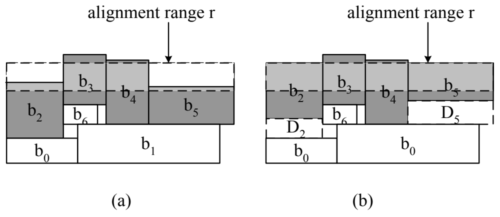

Figure 5: (a) The placement obtained in the 1st pass, where blocks 2 and 5 fall out of the alignment range. (b) The placement obtained in the 2nd pass, where those blocks violating the alignment constraint are adjusted by inserting corresponding dummy blocks with appropriate heights.

After getting the minimum movement for each alignment block, we set the height of the corresponding dummy block to Δ_i to shift the alignment block upward to the alignment range. Using such a two-pass packing scheme, we can guarantee that the final placement is feasible without violating any alignment constraint. As shown in Figure 5(a), the alignment blocks b_2, b_3, b_4, and b_5, abut one by one, but the blocks b_2 and b_5 fall out of the alignment range after the 1st-pass packing. Then, we compute the minimum movement (height) for each alignment (dummy) block i, Δ_i, and shift b_2 and b_5 upward by Δ_2 and δ_5, respectively. Fig 5(b) gives a feasible placement after the adjustment.

5 Placement with Performance Constraints

Traditional floorplanners try to minimize total wire length but cannot guarantee that critical nets meet the delay constraint. In order to make critical net delay satisfy the delay constraint, we need to place the blocks, called *performance blocks*, connected by critical nets near each other. The delay $D_{s,t}$ of a two-pin net from the source at (x_s, y_s) to a sink at (x_t, y_t) at the floorplanning stage can be approximated by the following equation:

$$D_{s,t} = \delta(|x_t - x_s| + |y_t - y_s|),$$

where δ is a constant to scale the distance to timing.

Note that we can use the above linear function to estimate the delay because the actual delay is close to linear to the source-sink distance with appropriate buffer insertions. (Of course, more sophisticated approximation can also be used for this purpose by trading off the running time.) From the above equation and the given delay bound, D_{max}, the distance from the source s to the sink t, $I_{s,t}$, must satisfy the following inequality to meet the performance constraint:

$$I_{s,t} = |x_t - x_s| + |y_t - y_s| = \frac{D_{s,t}}{\delta} \leq \frac{D_{max}}{\delta}.$$

For a net, we use the popular approximation that the distance of pins is given by half of the perimeter of the minimum bounding box of the blocks connected by the net. (Again, more sophisticated approximation can also be used by trading off the running time.) To meet the performance constraints, we shall place the constrained blocks in a bounding box whose half of the perimeter is smaller than the distance with the delay bound.

In Figure 6(a), the bounding box (dotted lines) of the blocks is smaller than the bounding box (dash lines) with the delay bound, so the placement is feasible for the given performance constraint. The placement of Figure 6(b) is infeasible because the bounding box of the blocks is greater than that with the delay bound. In Figure 6(c), we obtain a feasible placement with the performance constraint from Figure 6(a). Then we cluster the blocks as a rectilinear super block and fix the shape of the rectilinear super block. Therefore, the performance constraint will be satisfied afterwards. We can repartition the rectilinear super block into a set of new blocks for further processing with other blocks.

Let I_{bound} denote the *bounding distance* which is half of the perimeter of the maximum bounding box of blocks connected by the net, and I_{max} the *maximum bounding distance* which is the distance with the delay bound. We have the following property:

Property 3 *We can get a feasible placement with performance constraints by placing performance blocks in the bounding box whose bounding distance is smaller than or equal to the maximum distance bound.*

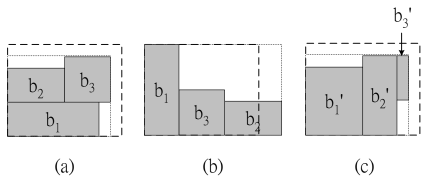

Figure 6: (a) A placement with performance blocks only. The dotted rectangle gives the bounding box of the blocks. The rectangle with dash lines gives the bounding box with the delay bound. The placement is feasible for performance constraints. (b) An infeasible placement with the blocks and the delay bound given in (a). (c) We can cluster the feasible placement in (a) into a new rectilinear block and repartition the rectilinear block into a set of new blocks for further processing with other blocks.

5.1 Feasibility Conditions for Performance Constraints

Given a set of blocks and performance constraints, we call the nodes representing performance blocks as *performance nodes* in a B*-tree. To meet the performance constraint, the performance blocks shall be located near each other. We take advantage of the processing for a rectilinear block presented in [7] to guarantee a feasible placement with the performance constraint. Given a placement $\mathcal{P}$ of k performance blocks whose areas are $A_i, i = 1, 2, ..., k$,, the width u, the height v, and the dead space S_{perf} of the placement $\mathcal{P}$, the sub-placement of the performance blocks must satisfy the following inequality:

$$u + v = I_{bound} \leq I_{max}$$

We do not restrict u or v so that the ratio of the bounding box can be adjusted. If the bounding distance of sub-placement is greater than the distance bound, the sub-placement cannot meet the performance constraints and thus the placement with the sub-placement either. Thus, we shall modify the sub-placement until it is smaller than the distance bound. By doing so, we obtain a set of feasible sub-placements for the performance blocks. Among these sub-placements, we pick the one with the minimum $S_{perf} = u \times v - \sum_{i=1}^{k} A_i$ and treat the sub-placement as a rectilinear block. Then we fix the rectilinear block (and thus fix the delay) for further processing with other blocks. By clustering performance blocks into an appropriate rectilinear block and fixing its shape, we can guarantee that the performance constraint will be satisfied throughout the remaining processing. For example, we repartition the rectilinear super block of Figure 6(a) as the new sub-blocks shown in Figure 6(c) and fix the relation between the new sub-blocks. By maintaining such a rectilinear block, we can guarantee a feasible placement with the performance constraints after the whole processing.

Theorem 2 *By pre-processing the performance blocks into the rectilinear blocks and keeping their shapes, we can guarantee to generate feasible placements with performance constraints.*

6 Algorithm

The flow of our algorithm is summarized in Figure 7. We use simulated annealing to search for an optimal solution. We perturb a B*-tree to another by the following operations:

- **Op1:** Rotate a block.
- **Op2:** Flip a block.
- **Op3:** Move a block to another place.
- **Op4:** Swap two blocks.
- **Op5:** Move a set of alignment blocks to another place.

The first four operations are used in [1] and the last one is designed for alignment constraints. In $Op1$, we rotate a block. This action can be applied to any node without changing the relationship between any two nodes except performance blocks. For performance blocks, we need to rotate its corresponding rectilinear blocks together. In $Op2$, we flip a block. Same as $Op1$, we need to maintain the correct relation for rectilinear blocks. $Op3$ and $Op4$ change the relations of blocks to get a different placement. We do not apply these two operations to alignment nodes. For performance blocks, we still need to move its corresponding rectilinear block to another place. $Op5$ moves a set of alignment blocks to another place. We first change the positions of the first pair of a dummy and an alignment nodes in the alignment shape. Then, we attach other pairs of dummy and alignment nodes to the correct positions to maintain their shape.

7 Experimental Results

We implemented our algorithm in the C++ programming language on a 450 MHz SUN Sparc Ultra 60. The benchmark circuits used for the comparative studies were adopted from [11]. Columns 1, 2, 3, and 4 of Table 1 give the name of the circuit, the number of blocks, the number of blocks with the alignment constraint, and the number of block with the performance constraint, respectively. Note that the constrained blocks are also the same as those defined in [11].

circuit	block	constrained blocks		Tang and Wong [11]			Ours		
		align	perf	time (s)	area (mm^2)	dead space	time (s)	area (mm^2)	dead space
apte	9	4	0	8	47.08	1.1%	3.6	46.92	0.8%
xerox-1	10	4	0	9	20.16	4.0%	5.8	20.08	3.7%
xerox-2	10	4	2	9	20.93	7.5%	6.4	20.08	3.7%
hp-1	11	4	0	10	9.342	5.5%	5.9	9.20	4.0%
hp-2	11	4	2	17	9.342	5.5%	6.1	9.349	5.6%
ami33-1	33	4	0	31	1.221	5.3%	35.4	1.180	2.0%
ami33-2	33	4	3	45	1.226	5.7%	52.6	1.181	2.2%
ami49-1	49	5	0	41	38.20	7.2%	132.7	36.60	3.2%
ami49-2	49	4	3	241	38.51	7.8%	97.9	36.56	3.1%
ami49-3	49	4	6	278	38.72	8.4%	109.2	36.64	3.3%
average				-		5.8%	-		3.2%

Table 1: Area and runtime comparison based on the benchmark circuits used in [11]. Note that the results reported in [11] ran on a 440 MHz SUN Sparc Ultra 10 machine while ours ran on a 450 MHz SUN Sparc Ultra 60 machine.

```
Algorithm: Placement with Alignment and Performance
           Constraints($blocks$, $constraints$)
Input: A set of blocks and alignment and performance constraints.
Output: A placement without violating the given constraints.
1.  Generate the rectilinear blocks for performance blocks
2.  Initialize a B*-tree for the input blocks and constraints;
3.  Simulated annealing process;
4.  do
5.      perturb();
6.      first-packing();
7.      adjust $y$-coordinates of the sub-blocks for rectilinear blocks.
8.      if alignment blocks fall out of the required area
9.          then adjust heights of dummy blocks
                to fix alignment violations
10.     final-packing();
11.     evaluate the B*-tree cost;
12. until converged or cooling down;
13. return the best solution;
```

Figure 7: The alignment and performance driven design flow.

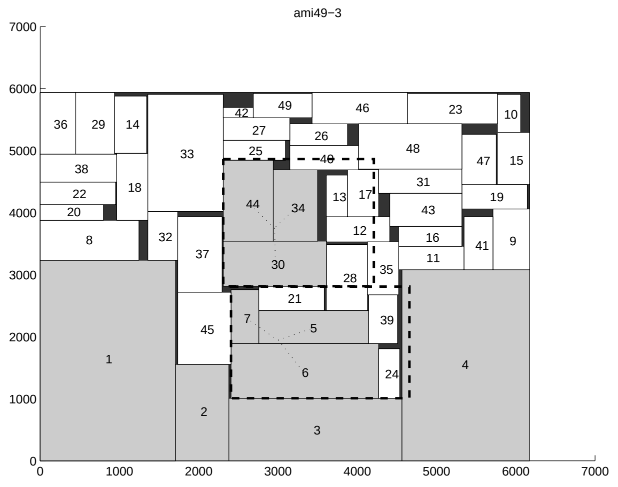

Figure 8: The placement of ami49-3. Blocks 1, 2, 3, and 4 are a group of blocks with an alignment constraint, blocks 5, 6, and 7 are with the same performance constraint, and blocks 30, 34, and 44 are another group of blocks with the same performance constraint.

Table 1 shows the experimental results. The results show that our algorithm obtained an average deadspace of only 3.2% for the set of ten benchmark circuits with the alignment constraint (and the performance constraint), compared to 5.8% reported in the previous work [11]. Figure 8 shows the resulting layout for ami49-3 with ten constrained blocks, four with alignment constraints and six with performance constraints. Further, as shown in Table 1, our B*-tree based algorithm is also very efficient. (Note that the results reported in [11] ran on a 440 MHz SUN Sparc Ultra 10 machine while ours ran on a 450 MHz SUN Sparc Ultra 60 machine.)

8 Conclusions

We have presented an efficient and effective algorithm to deal with the placement with the alignment and performance constraints. The algorithm is based on the B*-tree representation and the simulated annealing scheme. We have derived the feasibility conditions with the alignment and performance constraints. We have also proposed an algorithm that can guarantee a feasible placement with alignment constraints and generate a good placement with performance constraints during each operation. To evaluate a B*-tree with the constraints, it takes only amortized linear time (based upon the same assumption as the previous work that the number of groups of constrained blocks is constant), which achieves the best published time complexity for the evaluation operation. The experimental results have shown the effectiveness and efficiency of our algorithm.

Acknowledgments

We would like to thank Professor Martin D. F. Wong and Dr. Xiaoping Tang for providing us with the benchmark circuits. Special thanks go to Dr. Xiaoping Tang for his kind assistance.

References

[1] Y.C. Chang, Y.W. Chang, G.M. Wu, and S.W. Wu, ”B*-trees: A new representation for non-slicing floorplans,” *Proc. DAC*, pp. 458-463, 2000.

[2] P.N. Guo, C.K. Cheng, and T. Yoshimura, ”An O-tree representation of non-slicing floorplans and its applications,” *Proc. DAC*, pp. 268-273, 1999.

[3] J.M. Lin and Y.W. Chang, ”TCG: A transitive closure graph-based representation for non-slicing floorplans,” *Proc. DAC*, pp. 764-769, 2001.

[4] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani, “Rectangle-packing based module placement,” *Proc. ICCAD*, pp. 472–479, 1995.

[5] S. Nakatake, H.Murata, K. Fujiyoshi, and Y. Kajitani, ”VLSI module placement on BSG-structure and IC layour applications,” *Proc. ICCAD*, pp. 484-491, 1996.

[6] R.H.J.M. Otten, ”Automatic floorplan design,” *Proc. DAC*, pp. 261-267, 1982.

[7] Y. Pang, C.-K. Cheng, K. Lampaert, and W. Xie, ”Rectilinear block packing using O-tree representation,” *Proc. ISPD*, pp. 156-161, 2001.

[8] K. Sakanushi and Y. Kajitani, “The quarter-state sequence (Q-sequence) to represent the floorplan and applications to layout optimization,” *Proc. APCAS*, pp. 829–832, 2000.

[9] X. Tang, R. Tian, and D.F. Wong, ”Fast evalution of sequence pair in block placement by longest common subsequence computation,” *Proc. DATE*, pp. 106-111, 2000.

[10] X. Tang and D.F. Wong, ”FAST-SP: A fast algorithm for block placement based sequence pair,” *Proc. APS-DAC*, pp. 521-526, 2001.

[11] X. Tang and D.F. Wong, ”Floorplanning with alignment and performance constraints,” *Proc. DAC*, pp. 848-853, 2002.

[12] D.F. Wong and C.L. Liu, ”A new Algorithm for floorplan design,” *Proc. DAC*, pp. 101-107, 1986.

[13] J. Xu, P.-N. Guo, and C.-K. Cheng, ”Rectilinear block placement using sequence-pair,” *Proc. ISPD*, pp. 173-178, 1998.

[14] B. Yao, H. Chen, C.-K. Cheng, and R. Graham, ”Revisiting floorplan representations,” *Proc. ISPD*, pp. 138–143, 2001.

[15] F.Y. Young and D.F. Wong, ”Slicing floorplans with range constraint,” *Proc. ISPD*, pp. 97-102, 1999.