Block Placement Using the Segment Tree Data Structure from Computational Geometry*

Sarat C. Maruvada Karthik Krishnamoorthy Florin Balasa

Dept. of Computer Science, University of Illinois at Chicago

Abstract – Since Murata *et al.* introduced the elegant topological representation of a block placement configuration called sequence-pair [8], there was a significant research effort in the field of encoding systems for non-slicing floorplans. This paper presents a block placement technique operating on the set of binary tree representations of the layout – called B*-trees [3]. The novelty of this approach is due to an efficient B*-tree evaluation based on a data structure called *segment tree* [2], mainly used in computational geometry. Experimental results using device-level analog placement problems as benchmarks confirm the efficiency of the novel exploration approach.

1 Introduction

Block placement is a classical problem, but nevertheless, a difficult one: the block placement problem subject to nonoverlapping constraints was proven to be NP-hard even for rectangular blocks of fixed orientation [8]. The nature of the problem entailed the use of combinatorial optimization methods (as simulated annealing, genetic algorithms) to explore the solution space of placement configurations. These algorithms use stochastically controlled hill-climbing to avoid local minima during the optimization process. In addition, they do not impose severe constraints on the size of the problems or on the mathematical properties of the cost function. While efficiently trading-off between a variety of layout factors as area, total net length, aspect ratio, maximum chip width and/or height, cell orientation, "soft" cell shape, etc., they are very flexible – supporting incremental addition of new functionality, and they are relatively easy to implement (although good tunning requires more time).

The traditional way of approaching the block placement problem is to explore a huge search space using, for instance, simulated annealing as the optimization engine, where the cells are represented by means of absolute coordinates allowing illegal overlaps during their moves (translations, changes of orientation) in the chip plane [6].

Alternatively, the topological representations are based on a different idea – to define an encoding system as a solution space, each code representing a feasible placement configuration. While the classic absolute representation approach trades off a larger number of (annealing) moves for easier and quicker-to-build (often infeasible) layout configurations, the idea of using *topological* representations is to trade off more complex (but always feasible) layout constructions for a smaller number of moves. This is why fast algorithms are needed for any encoding system to evaluate each generated code representation, that is, to translate the code to its corresponding block placement.

The first remarkable encoding system not restricted to slicing floorplan topologies was proposed by Murata *et al.*, who suggested to encode the "left-right" and "above-below" positioning relations between blocks using two sequences of block permutations, named *sequence-pair* [8]. An $O(n^2)$ algorithm based on constructing a pair of horizontal and vertical constraint graphs was used for sequence-pair evaluation. More recently, a different approach of $O(n \log \log n)$ complexity – based on computing longest common subsequence in a pair of weighted sequences – was proposed by Tang *et al.* [11]. Nakatake *et al.* devised a meta-grid structure without physical dimensions (called *bounded-sliceline grid*) to define orthogonal relations between blocks, where the construction of the placement configuration is of quadratic complexity [9].

Guo *et al.* proposed the *ordered tree* (*O-tree*) data structure to reduce the drawback effect of redundancies in the two previous representations [4]. Chang *et al.* suggested a layout representation based on binary trees – called B*-trees [3]. This encoding is based on the so-called *natural correspondence* between forests of (rooted) trees and binary trees [7], and therefore it could be regarded as a representation equivalent to the ordered trees [4]. The corner block list, proposed more recently [5], is a representation that can be used to encode floorplans with zero dead-space ("mosaic" floorplans). Different from the tree-based representations, the sequence-pair, the bounded-sliceline grid, and the corner block list define the topological relations between blocks independent of their dimensions.

This paper thoroughly analyzes a novel algorithm building a placement configuration from a tree representation. Our algorithm achieves a $O(n \log n)$ complexity employing a more sophisticated data structure used in computational geometry [10] – called *segment tree*. Note that in [4] the authors claim that an ordered tree evaluation can be performed in $O(n)$ time, but they do not offer any information about the data structures employed. Chang *et al.* do not investigate the complexity of the running time, just quoting the result claimed by [4]. Insufficient clarity concerning implementation details prohibit us to independently assess the time complexity of the algorithm in [4]. The reason of our wariness is the following: a known lower bound for the problem of finding the p-edge contour of a union of n isothetic rectangles is $\Omega(n \log n + p)$ [10]. Since Guo *et al.* use a contour modification technique (idea also used in this paper), we just wonder whether a better complexity than $O(n \log n)$ can be obtained for the tree evaluation problem (as p can be $O(n)$ in the worst case). Nevertheless, our algorithm has an advantage which makes it extremely useful: it can be extended to address also device-level analog placement problems with symmetry constraints [1] (see

*This research was partly sponsored by the Design Automation Conference Graduate Scholarship Program.

0-7803-7523-8/02/$17.00 ©2002 IEEE

Section 4) without affecting its complexity. However, due to lack of space, this extension will be published elsewhere.

Section 2 will present the novel placement algorithm using the segment tree data structure, while Section 3 will discuss an illustrative example. Finally, Section 4 will give an overview of the experimental results and Section 5 will present the basic conclusions of this research.

2 Placement using a segment tree

The evaluation algorithm described below – which is executed in each inner-loop iteration of the simulated annealing – assumes that the given placement encoding is a binary tree; but due to the *natural correspondence* [7], it can operate with minor modifications if the encoding is an ordered tree [4]. A binary tree which nodes represent rectangular blocks imposes the following vertical and horizontal positioning constraints: (a) each block in the left binary subtree is above its root block; (b) if two blocks are overlapping along their y- axis projection, the block visited first in a preorder traversal (visit the root, traverse the left subtree, traverse the right subtree) is to the left of the block visited the second (see Fig. 2).

Let $B_1, \ldots, B_n$ be rectangular blocks to be placed on a chip area, each block B_i having the width w_i and the height h_i, and having (x_i, y_i) as coordinates of its left-bottom corner. First, the y- coordinates are computed during a preorder traversal of the binary tree:

```
initialize  y_i = 0;
for each node B_i (visited in a preorder traversal)
  if node B_k is the closest ancestor of B_i (if any)
   such that B_i is in the left subtree of B_k
  then y_i = max{y_i, y_k + h_k};
end_for
```

The algorithm above computes the block coordinates y_i since in the preorder traversal the nodes corresponding to the blocks *below* are visited before the nodes corresponding to the blocks *above* (see the vertical positioning constraints (a)). As the traversal of the tree is performed in linear time, and finding the closest ancestor of block B_i takes constant time (providing a stack is used in order to push the nodes reached when the traversal advances to the left), the computation of y_i's is done in $O(n)$ time.

The subtle part of the algorithm computing the block abscissae is the use of a *segment tree*, a data structure mainly employed in computational geometry, originally introduced by Bentley [2], designed to handle operations with intervals whose extremes belong to a fixed set of coordinates.

The (complete) segment tree is, basically, a rooted binary tree, where each node v has attached an interval $v.I = [c, d]$ with integer bounds. If $d - c > 1$, then node v has a left and a right descendant – denoted below as $v.left$ and $v.right$ – having associated the intervals $[c, \lfloor \frac{c+d}{2} \rfloor]$ and, respectively, $[\lfloor \frac{c+d}{2} \rfloor, d]$. The intervals attached to the nodes are called *standard*, while those pertaining to the leaves and having the length equal to 1 are named *elementary*. The complete segment tree is balanced; the depth of the complete segment tree is $\lceil \log_2(d-c) \rceil$ [10]. A complete segment tree is shown in Fig. 1(e-g).

First, the y- coordinates of the blocks are normalized by replacing each of them by its rank in their bottom-up order: the algorithm operates with intervals $[a_i, b_i]$ rather than $[y_i, y_i + h_i]$, where a_i, b_i are the indices of y_i and $y_i + h_i$ in the set S of interval extremes. In this way, the size of the segment tree will be kept minimal. Without loss of generality, the y- coordinates can hence be considered integers in the range $[0, 2n - 1]$. The scheme of the algorithm computing the block abscissae is given below:

```
initialize  x_i = 0;
sort the set of numbers S = {y_i} ∪ {y_i + h_i} ,
   eliminating the duplicates;
let m be the number of elements of set S;
SegmTreeNode v_0 = CreateNode ([0, m-1], 0);
for each block B_i (visited in a preorder traversal)
  let a_i be the index of y_i, and
  let b_i be the index of y_i + h_i in set S;
  InsertInterval (v_0, [a_i, b_i]);
  UpdateRightContour (v_0, [a_i, b_i]);
end_for
```

After the coordinate normalization, the segment tree is recursively built by the procedure *InsertInterval*. *CreateNode* is the constructor of a new node v in the segment tree, the two parameters being the interval $v.I$ and a value $v.x$ used for the computation of the abscissae x_i. The roots of the left and right subtrees of v are denoted $v.left$ and, respectively, $v.right$.

The procedure *InsertInterval* is inserting the normalized interval of $[y_i, y_i + h_i]$ – the spanning of block B_i along the y axis – into the segment tree, decomposing it into standard intervals. At the same time, the abscissa x_i of the left-bottom corner of block B_i is computed by taking the maximum over all the abscissae $v.x$ of the standard intervals v in the decomposition.

```
procedure InsertInterval (v, [a, b])
  if v.I ⊆ [a, b] then x_i = max{x_i, v.x};
  else let v.I = [c, d] and mid = ⌊(c+d)/2⌋;
    if v.left == Null then
        v.left  = CreateNode ([c, mid], 0);
        v.right = CreateNode ([mid, d], 0);
    if v.x > max{v.left.x, v.right.x}
       then v.left.x = v.right.x = v.x;
    if a < mid then InsertInterval(v.left, [a, b]);
    if mid < b then InsertInterval(v.right, [a, b])
end_procedure
```

Subsequently, the procedure *UpdateRightContour* sets the values of all the nodes corresponding to these standard intervals to $x_i + w_i$ – the abscissa of B_i's right border.

```
procedure UpdateRightContour (v, [a, b])
 if v.I ⊆ [a, b] then v.x = x_i + w_i;
 else let v.I = [c, d] and mid = ⌊(c+d)/2⌋;
   if a < mid then UpdateRightContour(v.left, [a, b]);
   if mid < b then UpdateRightContour(v.right, [a, b]);
   v.x = max{v.left.x, v.right.x};
end_procedure
```

The decomposition of the root segment into standard intervals, as well as the procedures *InsertInterval* and *UpdateRightContour*, are done in $O(\log n)$ time each. Indeed, since the interval associated to the root of the segment tree is $[0, m-1] \subseteq [0, n]$, the height of the segment tree is upper-bounded by $\lceil \log_2(n) \rceil$. Consequently, the overall complexity of the algorithm is $O(n \log n)$.

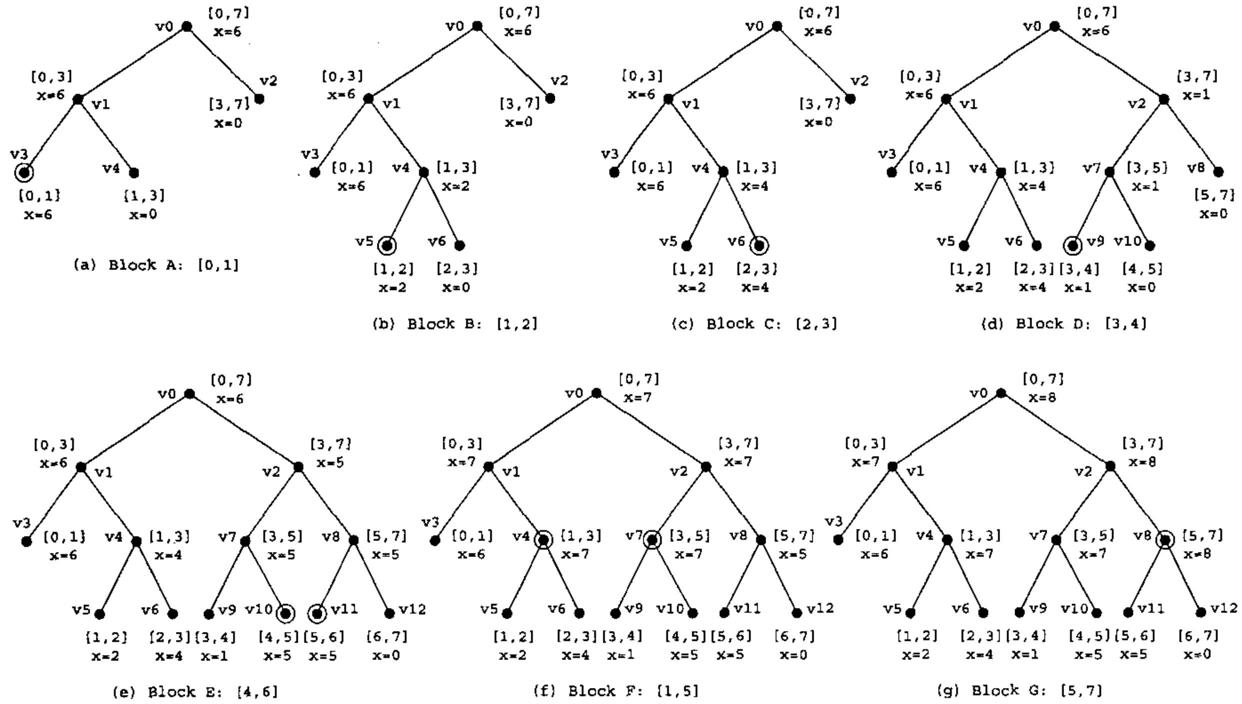

Figure 1: The segment tree after each iteration

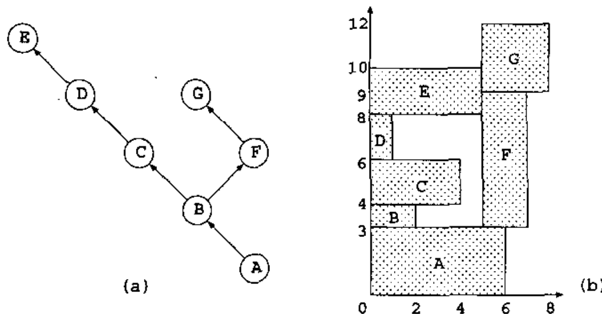

Figure 2: Illustrative example: (a) binary tree representation, and (b) corresponding block placement

3 Illustrative example

Consider the binary tree representation in Fig. 2(a) for seven rectangular blocks having the widths and heights indicated: A(6x3), B(2x1), C(4x2), D(1x2), E(5x2), F(2x6), G(3x3). In the preorder traversal the nodes of the tree are visited in the order A,...,G.

The computation of the y-coordinates yields successively $y_A = 0$, $y_B = y_A + h_A = 3$, $y_C = y_B + h_B = 4$, $y_D = y_C + h_C = 6$, $y_E = y_D + h_D = 8$, $y_F = y_A + h_A = 3$, $y_G = y_F + h_F = 9$. Note that the closest ancestor of node F such that F lies in its left subtree is node A.

All the block abscissae are initially zero. After sorting and eliminating the duplicates, the set $S = \{y_i\} \cup \{y_i + h_i\} = \{0, 3, 4, 6, 8, 9, 10, 12\}$ has $m = 8$ elements. Due to the normalization, the root node v_0 of the segment tree has attached the interval [0,7]. The interval $[y_A, y_A + h_A] = [0, 3]$ of the first node visited in the preorder traversal is normalized to $[a, b] = [0, 1]$ (since the indices of the elements 0 and 3 in set S are 0 and, respectively, 1). Since $v_0.I = [0, 7]$ and $mid = 3$, two new nodes v_1 and v_2 are created having attached the intervals [0,3] and [3,7]. *InsertInterval*$(v_1, [0, 1])$ is then recursively called and two new nodes v_3 and v_4 – having the intervals [0,1] and [1,3] – are created (Fig. 1(a)). The execution of *InsertInterval*$(v_3, [0, 1])$ computes $x_A = \max\{v_3.x, x_A\} = 0$ since $v_3.I = [0, 1]$. Afterwards, *UpdateRightContour* will visit once again the same nodes in the segment tree in order to update the x-values of the nodes with standard intervals. In this case, the only node is v_3: therefore, $v_3.x = x_A + w_A = 6$. Note that the segment tree in Fig. 1(a) describes exactly the contour of the right border after the placement of block A.

Figures 1(a-g) display the segment tree after each iteration, the blocks A,...,G being successively placed in the preorder traversal after the insertion of their normalized y-spanning intervals $[a_i, b_i]$ in the segment tree. The nodes corresponding to the standard segments are represented as double circles. The abscissae of the blocks are $x_A = x_B = x_C = x_D = x_E = 0$, $x_F = x_G = 5$, and the final value of $v_0.x$ (i.e., 8) is the width of the chip. The final placement is shown in Fig. 2(b).

Design	Nr. cells	Absolute		Seq.-pairs		B*-trees	
		Time	Area	Time	Area	Time	Area
n50	50	5.2	30.8	2.3	28.3	1.5	27.2
lpf2_b25b	52	7.5	42.8	2.4	39.9	1.9	38.4
dcservo_cmfb	66	17.8	62.0	4.0	60.8	2.9	60.4
modbias_2p4g	87	25.2	60.9	10.0	56.9	6.7	57.9
div_by_2or4	116	37.7	55.1	14.7	51.4	11.0	48.9

Table 1: Placement results (Time [min] , Area $[10^3 \times \mu m^2]$)

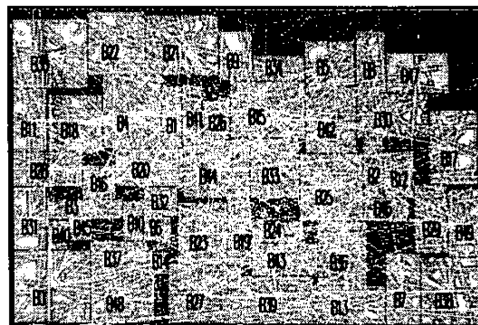

Figure 3: Placement of the *n50* test block

4 Experimental results

A placement tool using alternative exploration algorithms has been implemented. The tool can operate both with different topological representations (sequence-pairs and trees) and different code evaluation algorithms. In addition, for the purpose of comparative evaluation, a complementary placement algorithm based on the traditional absolute representation [6] has been embedded in the tool as well. The tool uses the simulated annealing algorithm as the combinatorial optimization engine. In order to ensure a correct comparative evaluation, the annealing schedule is identical for all the algorithms. The evaluation of a placement configuration is based on a cost function defined as a weighted sum, typical terms being the chip area and the total (estimated) wire length. For convenience, the result table displays only the area rather than an abstract cost.

Table 1 displays the results of our experiments carried out on a SUN Blade 100 workstation. The test benchmarks are several analog blocks, components of a digital spread spectrum transceiver. Since most of these analog macroblocks have symmetry requirements, the algorithm using sequence-pairs – which is basically similar to [11] – as well as our novel algorithm using segment trees were actually modified to handle symmetry constraints [1]. Figure 3 displays the placement solution obtained by our algorithm for one of our test blocks.

The experiments show that using topological representations is clearly better both in terms of computational effort and layout area than using the more traditional absolute representation. The running times obtained when using binary tree representations, with the evaluation algorithm employing the segment tree data structure are slightly better than the times obtained when using sequence-pairs (although in terms of placement quality, both algorithms prove to be effective). This may seem weird since the complexity of the sequence-pair algorithm is better $(O(n \log \log n))$. The explanation results from difference in size of the solution spaces, the number of trees being significantly smaller than the number of sequence-pairs [7].

5 Conclusions

This paper has presented a novel block placement technique operating on the set of binary tree representations of the layout. The efficiency of the new exploration approach is due to a data structure called *segment tree*, which is mainly used in computational geometry. A placement tool for analog layout, based on this novel technique, proved very effective while processing several industrial designs.

References

[1] F. Balasa, K. Lampaert, "Symmetry within the sequence-pair representation in the context of placement for analog design," *IEEE Trans. CAD of IC's and Systems*, Vol. 19, No. 7, pp. 721-731, July 2000.

[2] J.L. Bentley, "Algorithms for Klee's rectangle problem," Res. Report, Carnegie-Mellon Univ., Pittsburgh PA, 1977.

[3] Y.-C. Chang, Y.-W. Chang, G.-M. Wu, S.-W. Wu, "B*-Trees: a new representation for non-slicing floorplans," *Proc. 37th ACM/IEEE Design Automation Conf.*, pp. 458-463, Los Angeles CA, June 2000.

[4] P.-N. Guo, C.-K. Cheng, T. Yoshimura, "An O-tree representation of non-slicing floorplan and its applications," *Proc. 36th ACM/IEEE Design Automation Conf.*, pp. 268-273, June 1999.

[5] X. Hong, G. Huang, Y. Cai, J. Gu, S. Dong, C.-K. Cheng, J. Gu, "Corner block list: an effective and efficient topological representation of non-slicing floorplan," *Proc. IEEE Int. Conf. on Comp.-Aided Design*, pp. 8-12, San Jose CA, Nov. 2000.

[6] D.W. Jepsen, C.D. Gellat Jr., "Macro placement by Monte Carlo annealing", *Proc. IEEE Int. Conf. on Comp. Design*, pp. 495-498, Nov. 1983.

[7] D.E. Knuth, *The Art of Computer Programming*, Vol. 1 (3rd edition), Addison-Wesley Publ., 1997.

[8] H. Murata, K. Fujiyoshi, S. Nakatake, Y. Kajitani, "VLSI module placement based on rectangle-packing by the sequence-pair," *IEEE Trans. CAD of IC's and Systems*, Vol. 15, No. 12, pp. 1518-1524, Dec. 1996.

[9] S. Nakatake *et al.*, "Module packing based on the BSG-structure and IC layout applications," *IEEE Trans. on CAD of IC's and Syst.*, Vol. 17, pp. 519-530, June 1998.

[10] F.P. Preparata, M.I. Shamos, *Computational Geometry*, Springer-Verlag, 1985.

[11] X. Tang, D.F. Wong, "FAST-SP: A fast algorithm for block placement based on sequence pair," *Proc. Asia-S. Pacific Design Aut. Conf.*, pp. 521-526, Yokohama, Japan, 2001.