Slicing Floorplan Design with Boundary-Constrained Modules

En-Cheng Liu[1], Ming-Shiun Lin[1], Jianbang Lai[2], and Ting-Chi Wang[3]

[1]Department of Information and Computer Engineering, Chung Yuan Christian University, Chungli, Taiwan

[2]Tatung Corp., 22, Chungshan N. Rd., 3rd Sec., Taipei, Taiwan

[3]Department of Electrical Engineering, Texas A&M University, College Station, TX 77843-3259, USA

{s8777003, s8877030}@ice.cycu.edu.tw, ban@tatung.com, wangtc@ee.tamu.edu

ABSTRACT

We consider in this paper the problem of slicing floorplan design with boundary-constrained modules. We develop a quadratic-time method that correctly transforms a slicing floorplan into one that satisfies all given boundary constraints. The transformation method is incorporated into our floorplanning algorithm which employs the simulated annealing technique to seek for a possibly best solution. Unlike any other existing algorithm such as the one in [10], our floorplanning algorithm is always able to generate solutions satisfying all given boundary constraints. Furthermore, the experimental results indicate that our floorplanning algorithm can also generate solutions with smaller area and interconnect wirelength than the algorithm in [10].

1. INTRODUCTION

Floorplan design is an early stage in VLSI physical design, and is to determine the shape and location of each module on the chip such that the total area and/or interconnect cost of the chip is as small as possible. In this stage, all (or most) of the modules are not designed yet and thus are called *soft* modules due to their flexibility in shape. There are two kinds of floorplans: slicing [3,6] and non-slicing [1,2,7]. A slicing floorplan is a floorplan that can be obtained by recursively cutting an enclosing rectangle by either a vertical line or a horizontal line. On the other hand, a floorplan that is not slicing is called a non-slicing floorplan. Clearly slicing floorplans constitute a smaller solution space than non-slicing floorplans. In addition, the shape flexibility of each module in a slicing floorplan can be efficiently utilized to pack modules as tightly as possible using proper shape curve computation techniques [4,5].

For slicing floorplan design several important breakthroughs have been made recently. The well-known simulated annealing based Wong-Liu algorithm [6] has been successfully extended to handle the case where some pre-placed modules are present [8], and the case where some modules are constrained to be placed within certain ranges [9]. The key to the success of each extension is based on a new shape curve computation technique.

ISPD'01, April 1-4, 2001, Sonoma, California, USA.

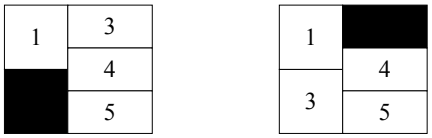
Besides, another new slicing floorplan design problem that considers *boundary constraints* was recently proposed [10]. The boundary constraints may come from the need where designers would like to place some modules along certain boundaries of the chip such that those modules are easier to be connected to certain I/O pads. The main idea behind the Young-Wong algorithm [10] for handling boundary constraints is to extend the Wong-Liu algorithm [6] by scanning each normalized Polish expression (which represents a slicing floorplan) from right to left once for determining the boundary information of each module in the floorplan. The boundary information of a module indicates whether there are modules above, below, on the left of, and on the right of the module. Hence with the boundary information, all the modules violating their boundary constraints in the floorplan can be found immediately. For example, if the boundary information of a module indicates that there is no module to the left of the module, then the module can be placed along the left boundary. For those modules violating their boundary constraints they will be then swapped with other modules in order to eliminate violations as many as possible. However, the swapping method may not always resolve all the violations, and hence a *penalty term* is added into the cost function to account for those violations. This unfortunately implies that the Young-Wong algorithm cannot guarantee that a floorplan satisfying all given boundary constraints is always obtainable unless the annealing process is long enough.

In this paper, we consider the same slicing floorplan design problem as that in [10]; that is, some modules must be placed along the boundaries. We develop a quadratic-time method that correctly transforms a normalized Polish expression into a slicing floorplan that satisfies all given boundary constraints. The transformation method is incorporated into our floorplanning algorithm which employs the simulated annealing technique to seek for a possibly best solution. Unlike the Young-Wong algorithm [10], our floorplanning algorithm is always able to generate solutions satisfying all given boundary constraints. Furthermore, the experimental results indicate that our floorplanning algorithm can also generate solutions with smaller area and interconnect wirelength than the Young-Wong algorithm.

2. PROBLEM FORMULATION

Each module is assumed to be a rectangle with a fixed area; its aspect ratio is defined to be its height divided by its width. A soft module is a module whose shape can be changed as long as the aspect ratio is within a given range and the area is as given. A slicing floorplan R of n modules consists of n non-overlapping

rectangles such that each rectangle is large enough to accommodate the module assigned to it. The problem of slicing floorplan design with boundary constraints is described as follows. The input consists of five *disjoint* module sets M_F, M_L, M_R, M_T, M_B, and the area and the aspect ratio range of each module. Each module in M_F is a *free* module that can be placed anywhere in a floorplan. Each module in M_L (M_R, M_T, M_B, respectively) is a module with the *left* (*right*, *top*, *bottom*, respectively) boundary constraint and has to be placed along the left (right, top, bottom, respectively) boundary. Note that M_L (M_R, M_T, M_B, respectively) is an empty set if no module must be placed along the left (right, top, bottom, respectively) boundary. A slicing floorplan is said to be *feasible* if each module in $M_L \cup M_R \cup M_T \cup M_B$ (referred to as a *boundary-constrained* module) is placed along the boundary as specified, and each module satisfies its area and aspect ratio constraints; otherwise the floorplan is said to be *infeasible*. (See Figure 1 for an example, where we assume each module satisfies its area and aspect ratio constraints.) The objective of the problem is to find a feasible slicing floorplan such that the cost function $A+\lambda W$, same as that used in [6], is as small as possible, where A is the total area of the floorplan, and W is the estimated interconnect wirelength, and λ is a user-specified constant to control the relative importance between area and wirelength. Given a floorplan, its W is defined to be $\sum_{i \neq j} c_{ij} d_{ij}$, where each c_{ij} is the number of common nets between modules i and j, and each d_{ij} is the Manhattan distance between the centers of i and j.



Suppose module 2 is constrainted to be placed along the right boundary. (a) is an infeasible floorplan but (b) is a feasible one.

Figure 1. A feasible and an infeasible floorplans.

3. A BRIEF REVIEW OF THE WONG-LIU ALGORITHM

A slicing floorplan can be described by an oriented rooted binary tree called *slicing* tree in which each internal node is labeled with + or * denoting a horizontal or a vertical cut, respectively, and each leaf denotes a module. Each leaf or internal node corresponds to a *sub-floorplan*. (Note that each sub-floorplan is also a rectangle consisting of all modules in the sub-tree rooted at the corresponding node.) For each internal node labeled with * (+), its left child is placed to the left of (below) its right child. Figure 2 gives a slicing floorplan of 5 modules, and its corresponding slicing tree. If a slicing tree is traversed in postorder, then a Polish expression is obtained. For example the Polish expression corresponding to the slicing tree in Figure 2 is 21+45*3+*. A Polish expression is said to be *normalized* if there is no consecutive +'s nor consecutive *'s in the expression.

The Wong-Liu algorithm in [6] uses the set of all normalized Polish expression as the solution space, and applies the simulated annealing technique to search for the possibly best solution. Whenever a new normalized Polish expression is generated, an efficient shape curve computation technique [4,5] is first used to find a floorplan of "best" area among all equivalent floorplans represented by that Polish expression. The interconnect wirelength is then computed from that floorplan, and finally the weighted cost of the area and the interconnect wirelength is used for evaluating whether the Polish expression should be accepted or not.

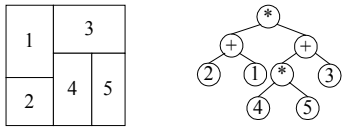


Figure 2. A slicing floorplan and its corresponding slicing tree.

4. DEFINITIONS

Recall that the five *disjoint* sets M_F, M_L, M_R, M_T, and M_B denote the set of free modules and the sets of modules having the left, right, top and bottom boundary constraints, respectively. Based on what types of boundary-constrained modules are contained in a sub-floorplan, we define 16 disjoint types of sub-floorplans below.

Definition 1: A sub-floorplan is said to have the *F* constraint if each module contained in it is in M_F.

Definition 2: A sub-floorplan is said to have the *L* constraint if it contains at least one module in M_L, but contains no module in M_R, M_T or M_B. A sub-floorplan having the *R*, *T*, or *B* constraint is defined similarly. A sub-floorplan having the *L*, *R*, *T*, or *B* constraint is said to have *one* type of boundary constraint.

Definition 3: A sub-floorplan is said to have the *LR* constraint if it contains at least one module in M_L and at least one module in M_R, but contains no module in M_T or M_B. A sub-floorplan having the *LT*, *LB*, *RT*, *RB*, or *TB* constraint is defined similarly. A sub-floorplan having the *LR*, *LT*, *LB*, *RT*, *RB*, or *TB* constraint is said to have *two* types of boundary constraints.

Definition 4: A sub-floorplan is said to have the *LRT* constraint if it contains at least one module in M_L, at least one module in M_R, and at least one module in M_T, but contains no module in M_B. A sub-floorplan having the *LRB*, *LTB*, or *RTB* constraint is defined similarly. A sub-floorplan having the *LRT*, *LRB*, *LTB*, or *RTB* constraint is said to have *three* types of boundary constraints.

Definition 5: A sub-floorplan is said to have the *LRTB* constraint if at least one module in M_L, at least one module in M_R, at least one module in M_T, and at least one module in M_B are contained in it. A sub-floorplan having the *LRTB* constraint is said to have *four* types of boundary constraints.

Clearly, each sub-floorplan is in exactly one of the above 16 different types of sub-floorplans. Besides, the constraint of a sub-floorplan can be also determined from the constraints of its two child sub-floorplans. For example, if the two child sub-floorplans have the *R* and the *LB* constraints, respectively, then the sub-floorplan has the *LRB* constraint

Definition 6: A sub-floorplan is said to have a *feasible topology* if each boundary-constrained module contained in the sub-floorplan is placed along the required boundary in the sub-floorplan; otherwise the sub-floorplan has an *infeasible topology*. Since each node in a slicing tree corresponds to a sub-floorplan, the definitions of feasible and infeasible topologies can be also applied to a node.

For a sub-floorplan to have a feasible topology, each of its two child sub-floorplans must also have a feasible topology. However, a sub-floorplan obtained from horizontally (or vertically) combining two sub-floorplans each of which has a feasible topology may or may not have a feasible topology. A combining is said to be *feasible* if the resulting sub-floorplan has a feasible topology. By considering all possible constraints that a sub-floorplan may have, and all possible ways of combining two sub-floorplans each of which has a feasible topology, all possible feasible combinings are obtained and shown in Table 1. In Table 1, *A* and *B* denote the left and right child sub-floorplans, respectively, and each is assumed to have a feasible topology.

Table 1. All feasible combinings.

Resulting constraint	* (vertical cut) A	* (vertical cut) B	+ (horizontal cut) A	+ (horizontal cut) B
F	1. F	1. F	1. F	1. F
L	1. L	1. F	1. L 2. F	1. F 2. L
R	1. F	1. R	1. R 2. F	1. F 2. R
T	1. T 2. F	1. F 2. T	1. F	1. T
B	1. B 2. F	1. F 2. B	1. B	1. F

(a)

Resulting constraint	* (vertical cut) A	* (vertical cut) B	+(horizontal cut) A	+(horizontal cut) B
LRTB	1. L 2. LT 3. LT 4. LB 5. LB 6. LTB 7. LTB 8. LTB 9. LTB	1. RTB 2. RB 3. RTB 4. RT 5. RTB 6. R 7. RT 8. RB 9. RTB	1. B 2. LB 3. LB 4. RB 5. RB 6. LRB 7. LRB 8. LRB 9. LRB	1. LRT 2. RT 3. LRT 4. LT 5. LRT 6. T 7. LT 8. RT 9. LRT

(b)

Resulting constraint	* (vertical cut) A	* (vertical cut) B	+ (horizontal cut) A	+ (horizontal cut) B
LR	1. L	1. R	1. L 2. R 3. L 4. R 5. LR 6. LR 7. LR 8. F	1. R 2. L 3. LR 4. LR 5. L 6. R 7. F 8. LR
LT	1. L 2. LT 3. LT	1. T 2. T 3. F	1. L 2. L 3. F	1. T 2. LT 3. LT
LB	1. L 2. LB 3. LB	1. B 2. B 3. F	1. B 2. LB 3. LB	1. L 2. L 3. F
RT	1. T 2. T 3. F	1. R 2. RT 3. RT	1. R 2. R 3. F	1. T 2. RT 3. RT
RB	1. B 2. B 3. F	1. R 2. RB 3. RB	1. B 2. RB 3. RB	1. R 2. R 3. F
TB	1. T 2. B 3. T 4. B 5. TB 6. TB 7. TB 8. F	1. B 2. T 3. TB 4. TB 5. T 6. B 7. F 8. TB	1. B	1. T

(c)

Resulting constraint	* (vertical cut) A	* (vertical cut) B	+ (horizontal cut) A	+ (horizontal cut) B	Middle constraint
LRT	1. L 2. LT 3. LT	1. RT 2. R 3. RT	1. L 2. R 3. L 4. R 5. LR 6. LR 7. LR 8. LR 9. F	1. RT 2. LT 3. LRT 4. LRT 5. T 6. RT 7. LT 8. LRT 9. LRT	T
LRB	1. L 2. LB 3. LB	1. RB 2. R 3. RB	1. B 2. LB 3. LB 4. RB 5. RB 6. LRB 7. LRB 8. LRB 9. LRB	1. LR 2. R 3. LR 4. L 5. LR 6. L 7. R 8. LR 9. F	B
LTB	1. L 2. LT 3. LT 4. LB 5. LB 6. LTB 7. LTB 8. LTB 9. LTB	1. TB 2. B 3. TB 4. T 5. TB 6. T 7. B 8. TB 9. F	1. LB 2. B 3. LB	1. T 2. LT 3. LT	L
RTB	1. TB 2. B 3. TB 4. T 5. TB 6. T 7. B 8. TB 9. F	1. R 2. RT 3. RT 4. RB 5. RB 6. RTB 7. RTB 8. RTB 9. RTB	1. RB 2. B 3. RB	1. T 2. RT 3. RT	R

(d)

5. A TRANSFORMATION METHOD

In this section we describe a method that correctly transforms a normalized Polish expression into a slicing floorplan that has a feasible topology. (Note that by Definition 6, a floorplan having a feasible topology satisfies all given boundary constraints.) The main idea is to first construct the slicing tree from the given Polish expression, and then examine each internal node of the tree in a bottom-up fashion (i.e., postorder traversal of the tree) and use Table 1 to determine if the node has a feasible topology or not. (It should be noted that our transformation method is designed in such a way that it is more efficient on a slicing tree than a Polish expression, and that is why the slicing tree needs to be constructed from a Polish expression in our method.) If the node has an infeasible topology, the tree will be modified to make the node have a feasible topology. There are three *basic operations*, called O_1, O_2, and O_3, which can transform a node having an infeasible topology into one having a feasible topology in most cases (but not all cases). Each O_1 operation changes the cut direction of a node, each O_2 operation swaps the left and the right sub-trees of a node, and each O_3 operation is to perform an O_1 operation followed by an O_2 operation. For example, suppose *A* has the *R* constraint, *B* has the *L* constraint, and both *A* and *B* have feasible topologies. Figure 3(a) shows a floorplan having an infeasible topology, but the floorplans in Figure 3(b), 3(c) and 3(d) all have feasible topologies. Figures 3(b), 3(c) and 3(d) can be obtained from Figure 3(a) by performing an O_1, an O_2, and an O_3 operations on the root, respectively. Although in most cases the three basic operations can help to generate nodes having feasible topologies, there are also some cases (as shall be seen later) where the basic operations cannot help.

Even if an internal node has a feasible topology, the tree also needs to be modified for some cases in order to ensure that the root has a feasible topology later on. Throughout the rest of this section, let *C* denote the internal node being examined, and *A* and *B* denote the left and right child nodes of *C*. Both *A* and *B* are ensured to have feasible topologies by our transformation method. Depending on whether *C* has a feasible topology or not, different approaches to modifying the tree are described in the following two sub-sections.

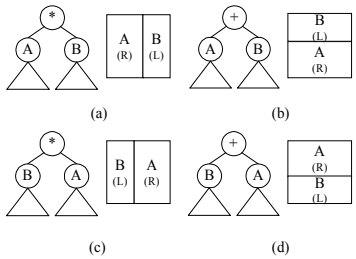


Figure 3. Illustration of the three basic operations.

5.1 A Feasible Combining

Suppose *C* corresponds to a sub-floorplan having a feasible topology. If *C* is the root of the tree, the whole transformation work is done and the current tree is the output. Otherwise, the following three cases (i.e., Cases 1, 2, and 3) need to be considered.

5.1.1 Case 1: C has the LRT, LRB, LTB, or RTB constraint

When this case happens, we must add into the sub-tree rooted at *C* all the modules that have the "middle" boundary constraint but are not currently in the sub-tree rooted at *C*. The *middle* boundary constraint is determined from the constraint that *C* has. For example, if *C* has the *LRT* constraint, the middle boundary constraint is defined to be the *T* constraint (i.e., top boundary constraint). The middle boundary constraint with respect to the *LRB*, *LTB*, or *RTB* constraint can be found in the rightmost column of Table 1(d). The reason why we need to pay attention to

this case can be explained using the following example. In Figure 4, assume *A* has the *LR* constraint, and *B* and *D* both have the *T* constraint. Then *C* has the *LRT* constraint. However, no matter which basic operation (i.e., O_1, O_2, or O_3) is applied to *E* (i.e., the parent of *C*) later on, *E* cannot have a feasible topology. Similar scenarios can be created for *C* having the *LRB*, *LTB*, or *RTB* constraint.

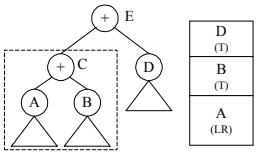


Figure 4. A scenario of Case 1.

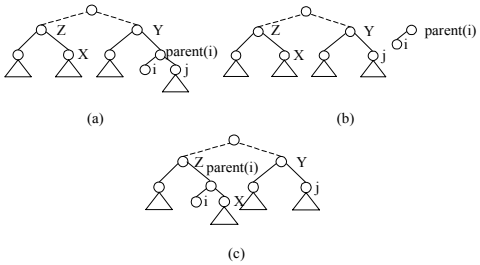


Figure 5. Illustration of the delete and insert operations.

Method: For each module that has the middle boundary constraint but is not in the sub-tree rooted at *C*, we combine it with *X,* where *X*=*A* if the left child sub-floorplan corresponding to *A* contains at least one module having the middle boundary constraint, and *X*=*B* otherwise. To combine a module *i* with *X*, we first perform the "*delete* (*i*)" operation which deletes *i* and its parent, say *parent*(*i*), from the tree and moves the other child of *parent*(*i*) to the original location of *parent*(*i*) in the tree. (See Figures 5(a) and 5(b).) We then perform the "*insert*(*parent*(*i*),*X*)" operation which makes *X* become the other child of *parent*(*i*) and moves *parent*(*i*) to the original location of *X* in the tree. (See Figures 5(c).) Now if *parent*(*i*) has an infeasible topology, then we continue to apply a proper basic operation to make it become feasible. By using proper data structures, each delete, insert or basic operation can be implemented in constant time. Hence combining *i* and *X* can be done in constant time. As a result, the method for handling Case 1 can be done in linear time because the number of modules that have the middle boundary constraint is not more than the total number of modules.

5.1.2 Case 2: C has the LRTB constraint

If this case happens, we need to modify the tree such that the two sub-trees of the root have exactly one and three types of boundary constraints, respectively. The reason why we need to consider this case can be explained using the following example. In Figure 6, assume *A* has the *LRB* constraint and *B* has the *T* constraint. Since *C* is not the root, we know it is not possible to obtain a floorplan having a feasible topology even after applying any of the three basic operations to the root later on.

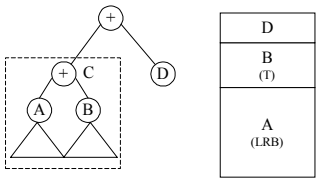


Figure 6. A scenario of Case 2.

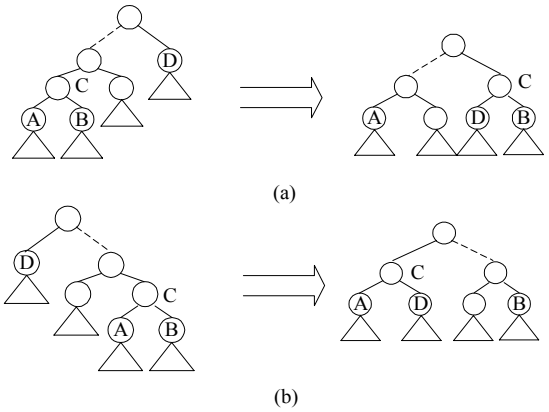


Figure 7. Illustration of the method for handling Case 2.

Method: First, we make one child node of *C*, say *X,* become having three types of boundary constraints (i.e., having the *LRT*, *LRB*, *LTB*, or *RTB* constraint) such that each module having any of the three types of boundary constraints is in the sub-tree rooted at *X*, and make the other child node, say *Y*, become having the remaining type of boundary constraint such that each module having that type of boundary constraint is in the sub-tree rooted at *Y*. This can be done by performing one delete operation, one insert operation, and possibly one basic operation for each boundary-constrained module. We then consider two cases. For the case where *C* is in the left sub-tree of the root, we continue to perform the "*subtree_delete*(*B*)" operation which deletes *parent*(*B*) (i.e., *C*) and the subtree rooted at *B*, and then perform the "*insert*(*parent*(*B*),*right_child*(*root*))" operation, where *root* denotes the root of the tree and *right_child*(*root*) denotes the right child of *root.* (See Figure 7(a).) For the other case where *C* is in the right sub-tree of the root, we continue to perform the "*subtree_delete*(*A*)" operation, and then perform the "*insert*(*parent*(*A*),*left_child*(*root*))" operation, where *left_child*(*root*) denotes the left child of *root.* (See Figure 7(b).) Now if *C* has an infeasible topology, we need to apply a proper basic operation to make it feasible. Finally if the root of the tree has an infeasible topology, we apply one more proper basic operation to make the root also feasible. Since the whole floorplan now has a feasible topology, the transformation work is done and the resulting tree is the output. It is not hard to verify that the method for handling Case 2 can be also implemented in linear time.

5.1.3 Case 3: C is obtained from one of the remaining feasible combinings

For this case, we do nothing.

5.2 An Infeasible Combining

Suppose *C* corresponds to a sub-floorplan having an infeasible topology. We have the following three cases (i.e., Cases 4, 5, and 6) to consider.

5.2.1 Case 4: The two child nodes of C have the LR and TB constraints, respectively

When this case happens, no matter which basic operation is applied to *C*, we still cannot make *C* feasible. Figure 8 gives two possible scenarios for this case, where *A* has the *LR* constraint and *B* has the *TB* constraint. To make *C* have a feasible topology, we will modify one of its child nodes to have exactly one type of boundary constraint (i.e., having the *T*, *B*, *L*, or *R* constraint) and the other child node to have the remaining three types of boundary constraints.

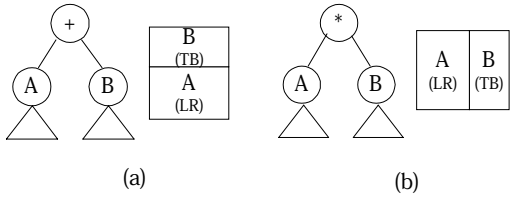


Figure 8. Two possible scenarios of Case 4.

Method: We first combine *A* with a boundary-constrained module that is currently in the sub-tree rooted at *B*. This can be done by performing one delete operation, one insert operation, and possibly one basic operation. Now *A* still has a feasible topology but becomes having three types of boundary constraints. Since *A* satisfies the condition given in Case 1, we then apply the Case 1 method to *A*. After that, if *C* still has an infeasible topology, we continue to apply a proper basic operation to make *C* satisfy the condition given in Case 2, and finally apply the Case 2 method to *C* if *C* is not a root. It is not hard to verify that the method for handling Case 4 can be also implemented in linear time.

5.2.2 Case 5: Both A and B have the same LT (RT, LB, or RB) constraint

When this case happens, no matter which basic operation is applied to *C*, we still cannot make *C* feasible. Figure 9 shows two possible scenarios for this case, where both *A* and *B* have the same *LT* constraint. To make *C* have a feasible topology, we will modify one of its child nodes to have exactly one type of boundary constraint.

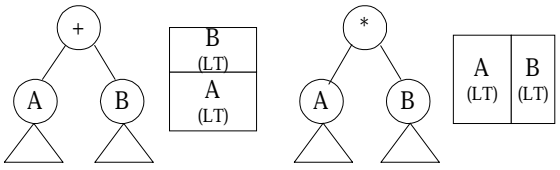


Figure 9. Two scenarios of Case 5.

Method: We arbitrarily choose one of the two types of boundary constraints as the "target" boundary constraint, and then remove each module in *B* that has the target boundary constraint, and add it into the sub-tree rooted at *A*. This can be done by repeatedly performing one delete operation, one insert operation, and possibly one basic operation. Now if *C* still has an infeasible topology, one proper basic operation is applied to it. The method for handling Case 5 can be also implemented in linear time.

5.2.3 Case 6: C is obtained from one of the remaining infeasible combinings

For this case, we apply a proper basic operation to *C* such that *C* becomes having a feasible topology. Now if *C* is not the root but meets the condition given in Case 1 (Case 2), the Case 1 method (Case 2 method) will be then applied to *C*. The method for handling Case 6 can be also implemented in linear time.

5.3 Overall Algorithm and Time Complexity

As stated at the beginning of this section, the transformation algorithm checks internal nodes in a bottom-up fashion. The algorithm will be terminated either by the Case 2 method or after the root has been checked. When examining an internal node *u*, the algorithm first applies the corresponding method to modify the tree (when necessary). If Case 2 never happens and the tree gets changed, the algorithm needs to update the constraint information of each internal node which is visited prior to *u* in the new tree (with respect to the postorder traversal) before the algorithm continues to examine the next internal node (which is the one right after *u* in the postorder traversal of the new tree). Each update can be done in constant time because each node will meet the condition of Case 3 and hence no modification to the tree needs to be made. Therefore, checking and fixing each internal node takes linear time. There are *n*-1 internal nodes, and hence the overall time complexity of the algorithm is $O(n^2)$, where *n* is the number of modules. (Note that constructing the slicing floorplan from a normalized Polish expression only takes linear time.) We have the following theorem.

Theorem 1: Our transformation method correctly transforms a normalized Polish expression into a slicing floorplan that satisfies all given boundary constraints. Moreover, it can be implemented in quadratic time.

6. OUR FLOORPLANNING ALGORITHM

In this section, we present our simulated annealing based algorithm for solving the problem of slicing floorplan design with boundary-constrained modules. Our algorithm is basically an extension of the Wong-Liu algorithm [6]. Each floorplan is represented by a normalized Polish expression. To generate the next expression from the current one, three types of operations, M1, M2 and M3, are used. Each M1 operation swaps two adjacent operands in an expression. (Each module is referred to as an operand.) Each M2 operation complements a chain of operators. (Each + or * is referred to as an operator.) Each M3 operation swaps two adjacent operand and operator. Since none of the three operations can always generate a slicing floorplan that satisfies all given boundary constraints, when a new Polish expression is generated, it will be transformed into a slicing floorplan satisfying all boundary constraints using our transformation method described in Section 5. Let *e* denote a normalized Polish expression, *T*' denote the slicing tree obtained from applying the transformation method to *e*, and *e*' denote the Polish expression obtained from the postorder traversal of *T*'. Note that *e* is the same as *e*' if *e* already represents a slicing floorplan that satisfies all boundary constraints. Our algorithm replaces *e* by *e*', and uses the same method as that given in the Wong-Liu algorithm [6] to

compute the cost of *e*'. If *e*' get accepted, then in order to have the property that each normalized Polish expression can be reached from any other through a finite set of M1, M2, or M3 operations, our algorithm uses *e* (instead of *e*') to generate the next Polish expression. The floorplan with the best cost during the entire annealing process is reported as the final solution.

7. EXPERIMENTAL RESULTS

Our algorithm has been implemented in C language. We compared our floorplanning algorithm with the Young-Wong algorithm [10]. Both algorithms used the same parameter values in the annealing process. That is, for each test case, both have the same initial temperature, termination condition, number of neighboring solutions generated at each temperature, initial normalized Polish expression, and the probabilities of the three move operations, M1, M2 and M3. All the experiments were conducted on a Pentium-III 600 processor with 128MB RAM. Two MCNC examples: ami33, ami49 were used as the test data. For each test data, we randomly generated three sets of boundary constraints. Each set of boundary constraints requires 16 (20) modules to be evenly placed along the boundaries for ami33 (ami49). The aspect ratio of each floorplan was set to be within 0.5~2. For each set of boundary constraints, we ran both algorithms 5 times. To optimize area alone, we set λ=0, and the experimental results are shown in Table 2. To optimize both area and interconnect wirelength, we set λ=0.0158, and the experimental results are shown in Table 3. Tables 2-3 list the minimum and average results of the area, interconnect wirelength (i.e., *W*) and run time generated by both algorithms. In both tables, the number of times that the Young-Wong algorithm failed to find a solution satisfying all given boundary constraints is also shown in the column "# Failure" for each test case. (Note that for the Young-Wong algorithm, the average and minimum results are calculated only from those successful ones.) The last four columns of each table give the improvement ratios of our algorithm over the Young-Wong algorithm in terms of the minimum and average results of area, and interconnect wirelength.

As can be seen from Table 2, when optimizing area alone, our algorithm improved the average area up to 9.38% for ami33, and up to 2.97% for ami49, as compared to the Young-Wong algorithm. Meanwhile the average interconnect wirelength was also improved up to 6.90% for ami33, and up to 15.64% for ami49. Our algorithm was also able to beat the Young-Wong algorithm in minimum area for each test case, and in minimum interconnect wirelength almost for all test cases.

When optimizing both area and interconnect wirelength, our algorithm was able to improve the average area up to 4.88% for ami33, and up to 4.48% for ami49. The average wirelength was also improved up to 8.53% for ami33, and up to 10.87% for ami49, as shown in Table 3. Again, our algorithm was able to beat the Young-Wong algorithm in minimum area and in minimum interconnect wirelength almost for all test cases.

It is also clear from Tables 2 and 3 that for some test cases, the Young-Wong algorithm failed to find feasible solutions 1~2 times, which never happened to our algorithm. As for the average run time, our algorithm was about 2~3X slower than the Young-Wong algorithm, but it is acceptable because our algorithm ran very fast, less than 1 minute almost for all test cases.

8. ACKNOWLEDGMENTS

We thank Prof. F. Y. Young at Chinese University of Hong Kong and Prof. D. F. Wong at University of Texas at Austin for kindly providing us with the source code of their algorithm [10] for the comparative study.

9. REFERENCES

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

[2] H. Murata and E. S. Kuh, "Sequence-Pair Based Placement Method for Hard/Soft/Pre-Placed Modules," *Proc. International symposium on Physical Design*, 1998, 167-172.

[3] R. H. J. M. Otten, "Automatic Floorplan Design," *Proc. Design Automation Conference*, 1982, pp. 61-267.

[4] R. H. J. M. Otten, "Efficient Floorplan Optimization," *Proc.* International *Conference on Computer Design*, 1983, pp. 499-502.

[5] L. Stockmeyer, "Optimal Orientations of Cells in Slicing Floorplan Designs," *Information and Control*, 1983, pp. 91-101.

[6] D. F. Wong and C. L. Liu, "A New Algorithm for Floorplan Design," *Proc. Design Automation Conference*, 1986, pp. 101-107.

[7] Y.-C. Chang, Y.-W. Chang, G.-M. Wu and S.-W. Wu, "B*-Tree: A New Representation for Non-Slicing Floorplans", *Proc. Design Automation Conf.*, 2000, pp. 458-463.

[8] F. Y. Young and D. F. Wong, "Slicing Floorplans with Pre-Placed Modules," *Proc. International Conference on Computer-Aided Design*, 1998, pp. 252-258.

[9] F. Y. Young and D. F. Wong, "Slicing Floorplans with Range Constraint", *Proc. International Symposium on Physical Design*, 1999, pp. 97-102.

[10] F. Y. Young and D. F. Wong, "Slicing Floorplans with Boundary Constraint," *Proc. ASP-DAC*, 999, pp. 17-20.

Table 2. Experimental results when optimizing area only.

	Young-Wong Algorithm							Our Algorithm						Improv. over Young-Wong Alg.			
	Area (mm^2)		Wirelength (mm)		# Failure	Time (sec)		Area (mm^2)		Wirelength (mm)		Time (sec)		Area		Wirelength	
λ = 0	Min	Average	Min	Average		Min	Average	Min	Average	Min	Average	Min	Average	Min	Average	Min	Average
ami33-1	1.17	1.22	89.09	95.40	0	3.09	5.706	1.16	1.17	84.19	90.58	14.25	15.51	0.85%	4.10%	5.50%	5.05%
ami33-2	1.17	1.28	94.82	111.89	1	7.02	7.523	1.16	1.16	95.08	104.17	14.35	16.07	0.85%	9.38%	-0.27%	6.90%
ami33-3	1.17	1.19	103.25	113.70	0	5.13	7.146	1.16	1.17	102.19	106.03	12.83	15.36	0.85%	1.68%	1.03%	6.75%
ami49-1	36.47	37.52	1818.60	1906.39	2	10.28	32.92	36.41	37.04	1763.30	1826.19	80.04	84.05	0.16%	1.28%	3.04%	4.21%
ami49-2	37.11	38.17	1864.80	1874.27	2	13.82	30.2	36.25	37.25	1497.54	1581.15	52.01	63.25	2.32%	2.41%	19.69%	15.64%
ami49-3	36.22	38.09	1702.43	1802.80	1	14.03	27.74	36.01	36.96	1357.10	1522.17	50.98	56.65	0.58%	2.97%	20.28%	15.57%

Table 3. Experimental results when optimizing both area and interconnect wirelength.

	Young-Wong Algorithm							Our Algorithm						Improv. over Young-Wong Alg.			
	Area (mm^2)		Wirelength (mm)		# Failure	Time (sec)		Area (mm^2)		Wirelength (mm)		Time (sec)		Area		Wirelength	
λ = 0.0158	Min	Average	Min	Average		Min	Average	Min	Average	Min	Average	Min	Average	Min	Average	Min	Average
ami33-1	1.17	1.21	75.13	78.56	0	4.04	6.97	1.17	1.18	67.99	73.58	12.64	15.39	0.00%	2.48%	9.50%	6.34%
ami33-2	1.21	1.23	76.20	83.12	0	3.68	6.87	1.17	1.17	76.08	82.16	15.04	16.44	3.31%	4.88%	0.16%	1.15%
ami33-3	1.19	1.24	90.67	99.02	0	3.48	6.29	1.17	1.18	88.16	90.57	13.56	15.19	1.68%	4.84%	2.77%	8.53%
ami49-1	37.00	38.87	1238.29	1436.56	0	5.83	22.64	36.44	37.13	1260.99	1323.91	50.19	56.67	1.51%	4.48%	-1.83%	7.84%
ami49-2	37.10	38.10	1316.60	1416.82	1	10.82	23.84	35.66	37.02	1306.20	1341.09	52.42	55.95	3.88%	2.83%	0.79%	5.34%
ami49-3	37.27	38.46	1287.51	1492.07	1	9.58	24.49	36.01	36.95	1204.94	1329.87	42.68	56.51	3.38%	3.93%	6.41%	10.87%