SLICING FLOORPLANS WITH BOUNDARY CONSTRAINT*

F.Y. Young and D.F. Wong
Department of Computer Sciences
The University of Texas at Austin
fyyoung@cs.utexas.edu wong@cs.utexas.edu

ABSTRACT

In floorplanning of VLSI design, it is useful if users are allowed to specify some placement constraints in the packing. One particular kind of placement constraints is to pack some modules on one of the four sides: on the left, on the right, at the bottom or at the top of the final floorplan. These are called boundary constraints. In this paper, we enhanced a well-known slicing floorplanner [10] to handle these boundary constraints. Our main contribution is a necessary and sufficient characterization of the Polish expression, a representation of the intermediate solution in a simulated annealing process, so that we can check these constraints efficiently and can fix the expression in case the constraints are violated. We tested our algorithm on some benchmark data and the performance is good.

1. INTRODUCTION

Floorplan design is an important step in physical design of VLSI circuits. It is the problem of placing a set of circuit modules on a chip to minimize total area and interconnection cost. In this early stage of physical design, most of the modules are not yet designed and thus are flexible in shape (soft modules) and are free to move (free modules).

Many existing floorplanners are based on slicing floorplans [1, 10, 3, 5, 9] and it is shown theoretically that slicing floorplans can pack modules tightly [11]. There are several advantages in using slicing floorplans. Firstly, focusing only on slicing floorplans significantly reduces the search space and this leads to fast runtime. Secondly, the shape flexibility of the modules can be fully exploited to pack modules tightly using an efficient shape curve computation technique [8, 6]. As a result, existing floorplanners that use slicing floorplans are very efficient in runtime and yet can pack modules tightly.

Recently, there are some interesting research activities in the direction of non-slicing floorplans. Two methods, bound-sliceline-grid (BSG) [7] and sequence-pair (SP) [2], are proposed. These methods are originally designed for placement of modules which have no flexibility in shape (hard modules). The sequence-pair method is recently extended to handle soft modules [4]. In order to handle soft modules, it needs to solve an expensive convex programming problem to determine the exact shape of each module numerous times, and thus results in long runtime. Note that for the same set of benchmark data (apte, xerox, hp, ami33, ami49) in [4], we run the slicing floorplanner in [10] and can obtain comparable results using only a fraction of the runtime. In fact, we have less than 1% dead space using no more than 7 seconds for all the test problems.

In floorplanning, it is useful if users are allowed to specify some placement constraints in the final packing. We did some previous work on floorplanning with pre-placed modules [12]. A pre-placed module is fixed in position, height and width. We solved this problem by a novel shape curve computation procedure which takes the positions of the pre-placed modules into consideration.

The placement constraint we consider here is called boundary constraint: some modules are constrainted to be packed on one of the four sides: on the left, on the right, at the bottom or at the top of the final floorplan. This is needed because users may want to place some modules along the boundary for I/O connections. In particular, if floorplanning is done independently for different units of a chip, it helps if some modules are constrained to be packed along the boundary so that they can abut with some other modules in the neighboring units. We extend a well-known slicing floorplanner by Wong and Liu [10]. Our main contribution is a necessary and sufficient characterization of the Polish expression, a representation of the intermediate solution in the simulated annealing process, so that we can check these boundary constraints efficiently and can fix the expression in case the constraints are violated. We tested our algorithm with some benchmark data and the performance is good.

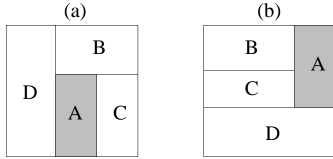
The rest of the paper is organized as follows. We first define the problem formally in Section 2. Section 3 provides a brief review of the Wong-Liu floorplanner. The new work is presented in Section 4 and the experimental results are shown in Section 5.

2. PROBLEM DEFINITION

A module A is a rectangle of height $h(A)$, width $w(A)$ and area $area(A)$. The aspect ratio of A is defined as $h(A)/w(A)$. 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 floorplan for n modules consists of an enveloping rectangle R subdivided by horizontal lines and vertical lines into n non-overlapping rectangles such that each rectangle must be large enough to accommodate the module assigned to it. There are two kinds of floorplans: slicing and non-slicing. A slicing floorplan is a floorplan which can be obtained by recursively cutting a rectangle into two parts by either a vertical line or a horizontal line. A non-slicing floorplan is a floorplan which is not slicing.

In our problem, we are given two kinds of soft modules $M = F \cup B$. The modules in F are free to move while

*This work was partially supported by the Texas Advanced Research Program under Grant No. 003658288 and by a grant from the Intel Corporation.



Suppose module A is constrainted to be packed along the right boundary. Then the packing in (a) is infeasible but the packing in (b) is feasible.

Figure 1. An example of a feasible floorplan

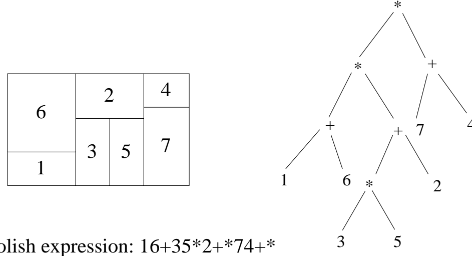


Figure 2. Slicing tree representation and Polish expression representation of a slicing floorplan

the modules in B are constrainted to be packed on one of the four sides of the final floorplan. A feasible packing is a packing in the first quadrant such that the width and the height of all the modules are consistent with their aspect ratio constraints and their area constraints, and all the modules in B are placed on the boundaries as required (Figure 1). Our objective is to construct a feasible floorplan R to minimize $A+\lambda W$ where A is the total area of the floorplan R , W is an estimation of the interconnect cost and λ is a constant that controls the relative importance of A and W. We require that the aspect ratio of the final packing is between two given numbers r_{min} and r_{max}.

3. WONG-LIU FLOORPLANNER

A slicing floorplan can be represented by an oriented rooted binary tree, called a slicing tree (Figure 2). Each internal node of the tree is labeled by a * or a +, corresponding to a vertical or a horizontal cut respectively. Each leaf corresponds to a basic module and is labeled by a number from 1 to n. No dimensional information on the position of each cut is specified in the slicing tree. If we traverse a slicing tree in postorder, we obtain a Polish expression and a Polish expression is said to be *normalized* if there is no consecutive *'s nor consecutive +'s in the sequence. It is proved in [10] that there is a 1-1 correspondence between the set of normalized Polish expressions of length $2n-1$ and the set of slicing floorplans with n modules.

In [10], Wong and Liu used the set of all normalized Polish expressions as the solution space for a simulated annealing method. In order to search the solution space efficiently, they defined three types of moves (M1, M2 and M3) to transform a Polish expression into another. They can make use of the flexibility of the soft modules to select the "best" floorplan among all the equivalent ones represented by the same Polish expression. This is done by carrying out an efficient shape curve computation [6, 10] whenever a Polish expression is examined. The cost function is $A+\lambda W$ where A is the total packing area and W is the interconnect cost. This algorithm is very efficient

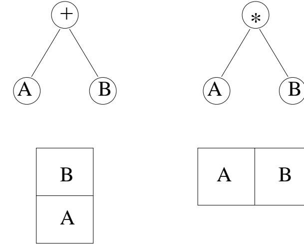


Figure 3. Relative positions of modules denoted by slicing trees

and the performance is very well.

However their method does not consider any placement constraint and there is actually a simple and natural way to handle boundary constraint in the Polish expression representation. We will describe it in the following section.

4. OUR METHOD

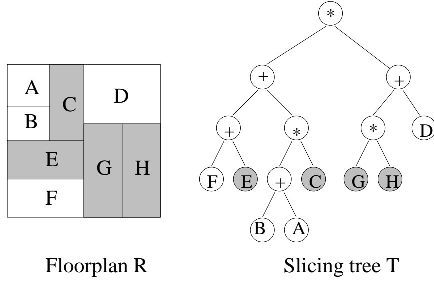
In the simulated annealing process, we check the normalized Polish expression in each iteration to see whether the boundary constraints are satisfied. This can be done efficiently in linear time by scanning the expression once. Then we fix the violated constraints as much as possible, and include in the cost a boundary constraint term to penalize the remaining violations.

4.1. Checking the Boundary Constraint

The slicing trees and Polish expressions have orientation. In Figure 3, the slicing tree on the left corresponds to a Polish expression $AB+$, which means that module A is below module B. The slicing tree on the right corresponds to the expression $AB*$, which means that module A is on the left of module B. Therefore if we want to pack a module A on the right (left) boundary of the final floorplan, the slicing tree T should be such that A is always in the right (left) subtree of any internal node of T labeled $*$. Similarly if we want to put a module A at the top (bottom) of the floorplan, the slicing tree T should be such that A is always in the right (left) subtree of any internal node of T labeled $+$. An example is shown in Figure 4. Lemma 1 summarizes the above observations:

Lemma 1 *Given a slicing tree T, a module in T is on the right boundary of the floorplan R corresponding to T if and only if it is in the right subtree of any internal node in T labeled $*$. A module is on the left boundary of R if and only if it is in the left subtree of any internal node in T labeled $*$. A module is on the upper boundary of R if and only if it is in the right subtree of any internal node in T labeled $+$. A module is on the lower boundary of R if and only if it is in the left subtree of any internal node in T labeled $+$.*

In the annealing process, we use Polish expressions to represent the slicing trees. It will be inefficient if we build a slicing tree in each iteration to check the conditions in Lemma 1. Actually we can check the necessary and sufficient conditions in Lemma 1 efficiently by scanning the Polish expression once. This is done by keeping a stack when scanning the expression from right to left. Each stack element x has four bits: $x.left$, $x.right$, $x.top$ and $x.bottom$. We push an element onto the stack whenever we see an operator α in the expression. This stack element represents the sub-floorplan X denoted by the subtree rooted at α in the slicing tree T. The four bits indicate whether there are modules above X, below X, on the right of X and on the



Module E is on the left boundary of R, so it must be in the left subtree of any internal node in T labeled "*"

Module H is on the right boundary of R, so it must be in the right subtree of any internal node in T labeled "*"

Module C is on the upper boundary of R, so it must be in the right subtree of any internal node in T labeled "+"

Module G is on the lower boundary of R, so it must be in the left subtree of any internal node in T labeled "+"

Figure 4. Characterization of slicing trees for different boundary constraints

left of X, e.g. $x.left = 1$ if and only if there is at least one module on the left of X in the floorplan.

We scan the Polish expression from right to left. When we scan a $+$, we push a new element x onto the stack. The four bits of x are copied from the previous stack top element, except that $x.bottom$ is assigned to 1. Similarly we push a new element onto the stack whenever we scan a $*$ but now we assign $x.left$ to 1 and copy the other three bits from the previous stack top element. The complete algorithm is given below. The invariant is that whenever we scan a module A in the expression, the four bits at the top of the stack will indicate whether there are modules above A, below A, on the right of A and on the left of A, and we can copy these information to $A.above$, $A.below$, $A.right$ and $A.left$. These four bits, when attached to a module name, indicate whether there are modules lying above, below, on the right and on the left of that module in the final floorplan. Finally we can check the boundary constraints with these information, e.g. a module A at the top of the floorplan should have $A.top = 0$.

4.2. Fixing a Polish Expression

If a Polish expression does not satisfy the boundary constraints, we can fix it as much as possible by shuffling some modules. An example is shown in Figure 5. In Figure 5, the boundary constraint is violated in (a) since module E is not packed at the bottom, as required. To fix this, we exchange E with F where F is the module closest to E in the Polish expression and that F is packed on the lower boundary. In general, if a module A is not packed along the boundary as required, we will shuffle it with another module B which is closest to A in the Polish expression and that B's position satisfies the boundary constraint of A.

It is possible that some constraints are still violated after all the possible shufflings, since a Polish expression may correspond to a floorplan which does not have enough positions along the boundaries to satisfy all the required constraints. We include a boundary constraint term in the cost function to penalize the remaining violated constraints. All violations will be eliminated as the annealing process proceeds because of this boundary constraint penalty term.

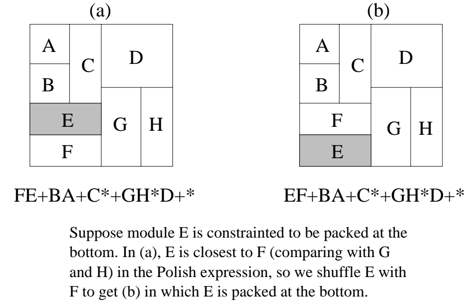


Figure 5. An example of fixing a Polish expression

4.3. Cost Function

The cost function is defined as $A + \lambda W + \gamma D$ where A is the total area of the packing obtained from the shape curve at the root of the slicing tree, W is the half-perimeter estimation of the interconnect cost, and D is a penalty term for the boundary constraint. The penalty term D is the distance of the module from the boundary of the floorplan along which it should be packed. For instance, if module A is constrainted to be packed on the right, the penalty term for A will be the distance between the right side of A and the right boundary of the final floorplan. The penalty terms are similarly defined for modules constrainted to be packed on the left, at the top and at the bottom. λ and γ are constants which control the relative importance of the three terms. λ is usually set such that the area term and the interconnect term are approximately balanced. The boundary constraint terms D will drop to zero as the process proceeds.

Algorithm Check-Boundary-Constraints

Input: A Polish expression
Output: For each module A, decide whether there are modules lying above A, below A, on the right of A and on the left of A in the final floorplan.

1. *Assign 0 to all four bits of stack[top]*
2. *For* $i = 2n - 1$ *downto* 1*:*
3.   *Let* α *be the* i^{th} *character in the Polish expression*
4.   *If* α *is a* $*$ *operator:*
5.   *Push a new element* x *onto the stack*
6. $x.left = 1$
7.   *Copy* $x.right$, $x.above$ *and* $x.below$ *from* $stack[top - 1]$
8. $x.flag = 0$; $x.op = *$
9.   *If* α *is a* $+$ *operator:*
10.   *Push a new element* x *onto the stack*
11. $x.below = 1$
12.   *Copy* $x.left$, $x.right$ *and* $x.above$ *from* $stack[top - 1]$
13. $x.flag = 0$; $x.op = +$
14.   *If* α *is a module name:*
15.   *Copy the four bits from stack[top] to* α
16.   *While* $stack[top].flag == 1$ *and* $top > 0$
17.   *Pop stack*
18.   *If* $top > 0$*:*
19. $stack[top].flag = 1$
20.   *If* $stack[top].op == *$
21. $stack[top].right = 1$
22. $stack[top].left = stack[top - 1].left$
23.   *If* $stack[top].op == +$
24. $stack[top].above = 1$
25. $stack[top].below = stack[top - 1].below$

5. EXPERIMENTAL RESULTS

We tested the above method on two MCNC building blocks examples: ami33 and ami49. ami33 has 33 modules and

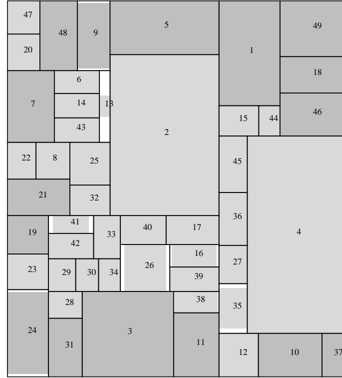


Figure 6. Result packing of ami49-rc1. Module 1, 5, 9 and 48 are constrainted to the upper boundary, 3, 10, 11 and 31 to the lower, 7, 19, 21 and 24 to the left, and 18, 37, 46 and 49 to the right.

123 nets. ami49 has 49 modules and 408 nets. We pick twelve modules from ami33 and sixteen modules from ami49, and require them to be packed along the boundaries evenly. We tested the floorplanner with ten data sets which are derived from the MCNC examples by imposing different boundary constraints on the selected module. The starting temperature is decided such that an accepting ratio is 100% at the beginning. The temperature is lowered at a constant rate (0.9), and the number of iterations at one temperature step is twenty times the number of modules. All the experiments were carried out on a 300 MHz Pentium II Intel processor.

Table 1 shows the experimental results. All the boundary constraints are satisfied in each data set. Both the packing quality and the efficiency are satisfactory. Figure 6 is the result packing of ami49-bc1 in which modules 7, 19, 21, 24 are constrainted to the left, module 18, 37, 46, 49 to the right, module 1, 5, 9, 48 to the top and module 3, 10, 11 and 31 to the bottom. Figure 7 is another result packing of ami49 in which we require modules 1, 3, 5, 7, 10, 24, 46, 49 to be packed at the top and modules 9, 11, 18, 19, 21, 31, 37, 48 at the bottom. Both packings are very tight and all the boundary constraints are satisfied.

Data	Dead space (%)	Time (sec)
ami33-bc1	1.81	4.58
ami33-bc2	1.33	4.53
ami33-bc3	1.62	4.41
ami33-bc4	1.86	4.28
ami33-bc5	1.51	4.01
ami49-bc1	1.51	37.77
ami49-bc2	3.17	36.98
ami49-bc3	4.65	39.76
ami49-bc4	3.48	36.03
ami49-bc5	4.25	37.60

Table 1. Results of testings with MCNC examples

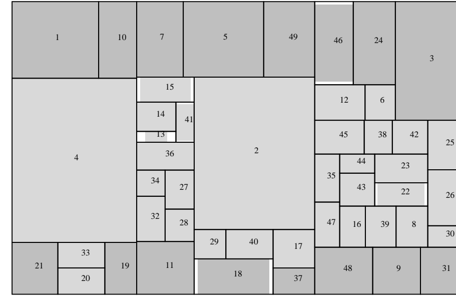


Figure 7. Another result packing of ami49. Module 1, 3, 5, 7, 10, 24, 46 and 49 are constrainted to the upper boundary. Module 9, 11, 18, 19, 21, 31, 37 and 48 are constrainted to the lower boundary.

REFERENCES

[1] K. Bazargan, S. Kim, and M. Sarrafzadeh. Nostradamus: A floorplanner of uncertain design. *International Symposium on Physical Design*, pages 18–23, 1998.

[2] S. Nakatake H. Murata, K. Fujiyoushi and Y. Kajitani. Rectangle-packing-based module placement. *Proceedings IEEE International Conference on Computer-Aided Design*, pages 472–479, 1995.

[3] D.P. Lapotin and S.W. Director. Mason: A global floorplanning tool. *Proceedings IEEE International Conference on Computer-Aided Design*, pages 143–145, 1985.

[4] H. Murata and Ernest S. Kuh. Sequence-pair based placement method for hard/soft/pre-placed modules. *International Symposium on Physical Design*, pages 167–172, 1998.

[5] R.H.J.M. Otten. Automatic floorplan design. *Proceedings of the 19th ACM/IEEE Design Automation Conference*, pages 261–267, 1982.

[6] R.H.J.M. Otten. Efficient floorplan optimization. *IEEE International Conference on Computer Design*, pages 499–502, 1983.

[7] H. Murata S. Nakatake, K. Fujiyoushi and Y. Kajitani. Module placement on BSG-structure and IC layout applications. *Proceedings IEEE International Conference on Computer-Aided Design*, pages 484–491, 1996.

[8] L. Stockmeyer. Optimal orientations of cells in slicing floorplan designs. *Information and Control*, 59:91–101, 1983.

[9] T. Tamanouchi, K. Tamakashi, and T. Kambe. Hybrid floorplanning based on partial clustering and module restructuring. *Proceedings IEEE International Conference on Computer-Aided Design*, pages 478–483, 1996.

[10] D.F. Wong and C.L. Liu. A new algorithm for floorplan design. *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pages 101–107, 1986.

[11] F.Y. Young and D.F. Wong. How good are slicing floorplans. *Integration, the VLSI journal*, 23:61–73, 1997. Also appeared in ISPD97.

[12] F.Y. Young and D.F. Wong. Slicing floorplans with preplace modules. *Proceedings IEEE International Conference on Computer-Aided Design*, 1998.