On extending slicing floorplans to handle L/T-shaped modules and abutment constraints

F.Y. Young[1], Hannah H. Yang[2] and D.F. Wong[3]

[1]Department of Computer Science and Engineering
The Chinese University of Hong Kong
fyyoung@cse.cuhk.hk

[2]Intel Corporation
Hillsboro, OR 97124-5961
hyang@ichips.intel.com

[3]Department of Computer Sciences
The University of Texas at Austin
wong@cs.utexas.edu

ABSTRACT

In floorplanning, it is common that a designer wants to have certain modules abutting with one another in the final packing. Unfortunately, few floorplanning algorithm can handle abutment constraints although this feature is useful in practice. The problem of controlling the relative positions of an arbitrary number of modules is non-trivial. Slicing floorplans have an advantageous feature that the topological structure of the packing can be found without knowing the module dimensions. This feature is good for handling placement constraints in general. In this paper, we make use of it to solve the abutment problem in the presence of L-shaped and T-shaped modules. This is done by a procedure which explores the topological structure of the packing and find the neighborhood relationship between every pair of modules in linear time. This enables us to check and fix the abutment constraints and to handle the L-shaped and T-shaped modules. There are many previous works on rectilinear block packing but none of them can handle rectilinear blocks with soft modules efficiently. Our main contribution is a method which can handle abutment constraints in the presence of L-shaped or T-shaped modules in such a way that the shape flexibility of the soft modules can still be fully exploited to obtain a tight packing. We tested our floorplanner with some benchmark data and the results are promising. We can pack 62 modules, 10% of which are L-shaped or T-shaped, with twelve abutment constraints in about 15 minutes giving less than 6% deadspace using a 143 MHz UltraSPARC workstation.

1. INTRODUCTION

Floorplanning 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 optimize the circuit performance. It is not just a simple packing problem. Besides optimizing the packing area and interconnect cost, there are some constraints that the designers may want to impose on the final packing for different reasons. For example, a designer may want to have the logic modules in a pipeline of a circuit to abut one after another to favor the transmission of data between them. This abutment problem is very common in practice but few floorplanning algorithm can handle these constraints. The problem of controlling the relative positions of an arbitrary number of modules is non-trivial. In most stochastic floorplanning algorithms, the abutment information is not known until the exact dimensions of the modules are taken into account and there is no systematic method to fix the violated constraints.

In the floorplanning stage, most of the modules are not yet designed and thus are flexible in shape (*soft modules*), while some of them are re-used and their shapes are fixed (*hard modules*). A good floorplanning algorithm should be able to handle both soft and hard modules effectively. 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 one not restricted to be slicing. There are several advantages of using slicing floorplans although non- slicing floorplans are more general. Firstly, focusing only on slicing floorplans significantly reduces the search space which in turn leads to a faster runtime. Secondly, the shape flexibility of the soft modules can be fully exploited to give a tight packing based on an efficient shape curve computational technique [3; 4]. It has been shown mathematically that a tight packing is achievable [7] for slicing floorplans.

Slicing floorplans have another advantageous feature that we can find out the topological structure of the packing without knowing the module dimensions. This feature is good for handling placement constraints in general We can check and fix the constraints given those topological information. In the case of abutment constraints, we devised a procedure called *Neighbor* which can find out the neighborhood relationship between all pairs of modules in linear time and the results of which enable us to check whether two modules abut as required and to fix a violated constraint by shuf-

(a) Both C and D are right neighbors of A.

(b) A and B now abut horizontally.

Figure 1: *Shuffling modules to obtain a feasible packing.*

fling the modules. An example is shown in Figure 1. In this example, module A is constrained to abut with module B horizontally. Figure 1(a) is the original packing in which the constraint is violated. By examining the topological structure of the packing, we find the neighbors of A, i.e. C and D in this example. Shuffling B with a right neighbor of A gives us a similar packing with the constraint satisfied (Figure 1(b)). This approach is actually an extension and generalization of the method used in [8] to handle boundary constraints. This work is however more general as we are finding out the neighborhood relationship between every pair of modules, not just between the modules and the chip boundary.

We have also made used of this feature to handle rectilinear blocks in slicing floorplans. Because of the recent advance in semiconductor manufacturing technology, new packaging schemes such as Multi-Chip Modules (MCMs) and integrated circuit components often have their shapes more complex than a simple rectangle. A lot of works have been reported on placement of rectilinear blocks but none of them can handle rectilinear blocks with soft modules efficiently. [2] proposes the bounded 2D contour searching algorithm to handle arbitrarily shaped rectilinear and soft modules but their method focuses mainly on area minimization [6] extends the idea of slicing floorplan to handle L-shaped modules, but only hard modules and L-shaped modules can be handled. Only [1] can handle soft modules with a greedy heuristic method but it takes about 3.54 hours to pack a set of 50 modules on a Sun SPARC 20 Workstation.

Slicing floorplans are well known to be effective in handling soft modules. It is not obvious how it can handle L-shaped or T-shaped modules because of the nature of slicing floorplans that the regions inside must be rectangular in shape. A common practise is to partition the L-shaped or T-shaped modules into rectangular sub-modules. These sub-modules are packed independently initially. A post-processing step will later move them back together to form the original shapes. This method is unnatural as we will repeatedly move the sub-modules to different places and move them back together in every iteration. The post-processing step is complicated and time-consuming. In our work, we treat the L-shaped or T-shaped modules as single modules but they will be *expanded* to their original shapes when being packed. An example is shown in Figure 2. In this example, module D is L-shaped. The initial packing is shown in Figure 2(a). We will expand D to its original shape before computing the total area and interconnect cost. The packing after expansion is shown in Figure 2(b). The expansions are dependent on the relative positions of the L-shaped or

Figure 2: *An Example of module expansion. D is L-shaped.*

T-shaped modules in the original packing. Again, we explore the topological structure of the packing and expand the modules accordingly. After calculating the total area and interconnect cost, they are treated as single modules again before the floorplan is transformed in the next iteration of the stochastic process.

Our main contribution is a method which can handle abutment constraints in the presence of L-shaped or T-shaped modules in such a way that the shape flexibility of the soft modules can still be fully exploited to obtain a tight packing. We tested our floorplanner using some benchmark data. The experiments give very promising results. We can pack 62 modules, 10% of which are L-shaped or T-shaped modules, with twelve abutment constraints in about 15 minutes giving less than 6% deadspace using a 143 MHz UltraSPARC workstation. The rest of the paper is organized as follows. We first define the problem formally in Section 2. Section 3 presents our method to handle abutment constraints with L-shaped and T-shaped modules. Experimental results are shown in Section 4.

2. PROBLEM DEFINITION

We consider three kinds of modules $M = M_R \cup M_L \cup M_T$ where M_R is a set of rectangular modules, M_L is a set of L-shaped modules and M_T is a set of T-shaped modules. A rectangular module A is a rectangle of height $h(A)$ and width $w(A)$. The *aspect ratio* of A is defined as $h(A)/w(A)$. A rectangular module can either be *hard* or *soft*. The height and width of a hard module are fixed but the module is free to rotate. The shape of a soft module can be changed as long as the area remains a constant and the aspect ratio is within a given range. An L-shaped module B (Figure 3(a)) consists of two rectangular sub-modules B_1 and B_2, where $w(B_1)$ and $w(B_2)$ are aligned and $h(B_1) > h(B_2)$. A T-shaped module C (Figure 3(b)) consists of three rectangular sub-modules C_1, C_2 and C_3 where $w(C_1)$, $w(C_2)$ and $w(C_3)$ are aligned and $h(C_1) > \max\{h(C_2), h(C_3)\}$. We assume that all T-shaped and L-shaped modules are hard modules.

In general, two modules A and B are said to be abut horizontally (Figure 4), denoted by *Habut(A,B)* if a vertical boundary l_A of module A and a vertical boundary l_B of module B abut such that l_A lying immediately on the left of l_B and the length of the abutment is at least $\min\{len(l_A), len(l_B)\}$ where $len(l_A)$ is the length of l_A and $len(l_B)$ is the length of l_B. The abutment in the vertical direction is defined similarly.

A floorplan for n modules is a dissection of a rectangle by horizontal and vertical lines into n non-overlapping regions such that each region must be large enough to accommodate the module assigned to it. A *packing* is a non-overlap

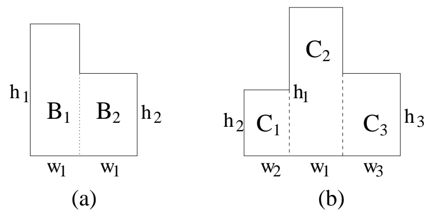


Figure 3: L-shaped Modules and T-shaped Modules

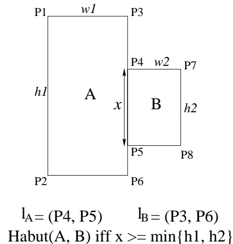


Figure 4: An Abutment Example

placement of all the modules in M. A *feasible packing* is a packing such that all the abutment constraints are satisfied and the widths and heights of all the soft modules are consistent with their aspect ratio constraints and area constraints. Our objective is to construct a feasible packing F to minimize $A + \lambda W$ where A is the total area of the packing, W is an estimation of the interconnect cost and λ is a user-specified constant which controls the relative importance of A and W in the cost function. We require that the aspect ratio of the final packing is between two given numbers r_{min} and r_{max}.

3. SLICING FLOORPLANS

A slicing floorplan can be represented by an oriented rooted binary tree, called a slicing tree (Figure 5). Each internal node of the tree is labeled by a $*$ or a $+$ operator, 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*. A Polish expression is said to be *normalized* if there is no consecutive $*$'s or $+$'s in the sequence. It is proved in [5] 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. Our method is developed based on the simulated annealing algorithm in [5].

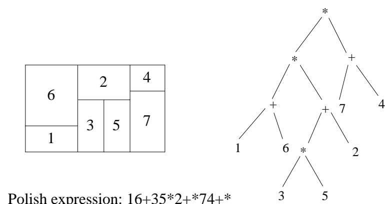


Figure 5: Slicing tree representation and Polish expression representation of a slicing floorplan

4. OUR APPROACH

4.1 An Overview

The algorithm *Main* below outlines the flow of our method. In each step of the annealing process, we consider a particular Polish expression. We will scan the expression once to find out the topological structure of the packing and, in particular, the neighborhood relationship between every pair of modules. This is possible because the operators $+$ and $*$ in a Polish expression have orientations, e.g. $AB+$ means that A is right below B and $AB*$ means that A is on the left of B immediately. We will scan the expression once to mark the left, right, top and bottom neighbors of every module. Figure 6 shows a simple example in which the neighbors of every module are marked in a table after this step. Then we will shuffle the modules to satisfy as many abutment constraints as possible. Please refer back to Figure 1 as an example. In this example, module A is constrained to abut with module B horizontally, i.e. *Habut(A,B)*, but this constraint is violated in the original packing (Figure 1(a)). After finding the neighborhood information between all pairs of modules, we will shuffle B with a closest right neighbor of A, i.e. module D in this example, to obtain a similar packing (Figure 1(b)) which satisfys the constraint. After this shuffling step, the abutting modules will stay together unless some later moves break them apart.

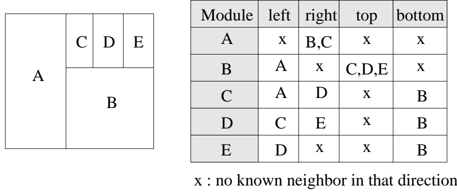
After fixing the abutment constraints, we will *expand* the L-shaped or T-shaped modules into their original shapes. This is done by modifying the Polish expression to embed the sub-modules of the rectilinear blocks in such a way that the relative positions between all the modules in the original Polish expression are preserved. Please refer back to Figure 2 as an example. In this example, module D is L-shaped. The initial packing is shown in Figure 2(a). We will expand D to its original shape before computing the total area and interconnect cost. The packing after expansion is shown in Figure 2(b). After expansion, we can do the shape curve computation as usual to obtain the total area of the final floorplan. The implementation is simple and the flexibility of the soft modules can still be fully exploited. We will describe the steps in details in the following sections.

Algorithm Main

Input: *The size, shape and interconnection of a set of modules $M = M_R \cup M_L \cup M_T$, where M_R is a set of rectangular modules, M_L is a set of L-shaped modules and M_T is a set of T-shaped modules, a set of horizontal abutment constraints and a set of vertical abutment constraints.*

Output: *A feasible packing of the modules in M*

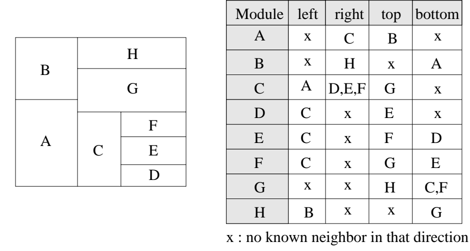
1. *Initialization.*
2. *Repeat:*
3. *Transform the Polish expression α_{old} to α.*
4. *Scan α to find the neighbors of every module.*
5. *Modify α to α_{new} by shuffling modules to fix the violated abutment constraints.*
4. *Expand the L-shaped or T-shaped modules in α_{new} to obtain a new Polish expression β.*
5. *Calculate the total area and interconnect cost of the floorplan represented by β.*
6. *Decide whether to accept α_{new}. If yes, $\alpha_{old} = \alpha_{new}$.*
7. *Until Cost $< k$.*



Module	left	right	top	bottom
A	x	B,C	x	x
B	A	x	C,D,E	x
C	A	D	x	B
D	C	E	x	B
E	D	x	x	B

x : no known neighbor in that direction

Figure 6: Neighborhood information can be obtained from the Polish expression



Module	left	right	top	bottom
A	x	C	B	x
B	x	H	x	A
C	A	D,E,F	G	x
D	C	x	E	x
E	C	x	F	D
F	C	x	G	E
G	x	x	H	C,F
H	B	x	x	G

x : no known neighbor in that direction

Figure 7: Abutment between modules

4.2 Handling Abutment Constraints

4.2.1 *Finding the Neighbors of a Module*

We can find the neighborhood of a module from the Polish expression because the operators in the expression have orientations, e.g. $AB+$ means that A is right below B and $AB*$ means that A is on the left of B immediately. These topological relationship is independent of the dimensions of the modules. For example, Figure 7 is a packing corresponding to the expression $AB+CDE+F+*G+H+*$. We can tell from the Polish Expression the neighborhood relationship as shown in the table. This information can be obtained by scanning the expression once and update the table whenever an operator is seen, i.e. when two sub-floorplans are combined by either a $+$ operator (vertical cut) or a $*$ operator (horizontal cut). The algorithm *Neighbor* below outlines the step to find this neighborhood information. Notice that the variables $Lside[X]$, $Rside[X]$, $Tside[X]$ and $Bside[X]$ denote the set of modules lying along the left boundary, right boundary, top boundary and bottom boundary of a sub-floorplan X. Consider combining two sub-floorplans X and Y horizontally as in $XY*$. If both $Rside[X]$ and $Lside[Y]$ have more than one modules, the top module in $Rside[X]$ will abut horizontally with the top module in $Lside[Y]$ and the bottom module in $Rside[X]$ will abut horizontally with the bottom module in $Lside[Y]$. Lets explain with the example in Figure 7. When we combine the sub-floorplan containing A and B and the sub-floorplan containing C, D, E, F, G and H by the $*$ operator, we know that B will abut with H horizontally and A will abut with C horizontally. Notice that we do not know whether G will abut with A or B or both because this is dependent on the dimensions of the modules, so we will not say anything about the abutment of G. However, if any one of $Rside[X]$ or $Lside[Y]$ has only one module, every module in $Rside[X]$ will abut with every module in $Lside[Y]$ horizontally. For example, in Figure 7, when we combine the sub-floorplan containing C and the sub-floorplan containing D, E and F by the $*$ operator, we know that C will abut with D, E and F horizontally. Similarly, we can derive the vertical neighborhood relationship from the $+$ operator.

Algorithm Neighbor

Input: *A Polish expression* $\alpha = \alpha_1\alpha_2\ldots\alpha_{2n-1}$

Output: *For each module A, find the modules abutting with A in all four directions.*

1. *For* $i = 1$ *to* $2n-1$*:*
2. *If* α_i *is a module name:*
   $Lside[\alpha_i] = Rside[\alpha_i] = Tside[\alpha_i] = Bside[\alpha_i] = \alpha_i$
3. *Push* α_i.
4. *If* α_i *is a* $*$ *operator:*
5. *Pop* Y. *Pop* X.
6. *If* $Rside[X]$ *or* $Lside[Y]$ *has only one module:*
7. $Habut[A, B]$ *is true for all* $A \in Rside[X]$ *and* $B \in Lside[Y]$.
8. *Else:*
9. $Habut[A_1, B_1]$ *and* $Habut[A_2, B_2]$ *are true where* A_1, A_2 *are the top and bottom modules in* $Rside[X]$ *resp., and* B_1, B_2 *are the top and bottom modules in* $Lside[Y]$ *resp..*
10. $Rside[\alpha_i] = Rside[Y]$, $Lside[\alpha_i] = Lside[X]$, $Tside[\alpha_i] = Tside[X] + Tside[Y]$, $Bside[\alpha_i] = Bside[X] + Bside[Y]$.
11. *Push* α_i.
12. *If* α_i *is a* $+$ *operator:*
13. *Pop* Y. *Pop* X.
14. *If* $Tside[X]$ *or* $Bside[Y]$ *has only one module:*
15. $Vabut[A, B]$ *is true for all* $A \in Tside[X]$ *and* $B \in Bside[Y]$.
16. *Else:*
17. $Vabut[A_1, B_1]$ *and* $Vabut[A_2, B_2]$ *are true where* A_1, A_2 *are the left and right modules in* $Tside[X]$ *resp., and* B_1, B_2 *are the left and right modules in* $Bside[Y]$ *resp..*
18. $Tside[\alpha_i] = Tside[Y]$, $Bside[\alpha_i] = Bside[X]$, $Rside[\alpha_i] = Rside[X] + Rside[Y]$, $Lside[\alpha_i] = Lside[X] + Lside[Y]$.
19. *Push* α_i.

4.2.2 *Shuffling Modules to Fix Violated Abutment Constraints*

If a Polish expression does not satisfy all the abutment constraint, we can fix it as much as possible by shuffling the modules. An example is shown in Figure 8. In this example, assume that module B is required to abut with F vertically, i.e. $Vabut(B, F)$, but it is violated initially as shown in Figure 8(a). We will then try to move F to the top of B or move B to the bottom of F. In the first case, B has two neighbors at the top: C and D. Since F is closer to D than to C in the Polish expression, we will shuffle F and D in order to fix this violated constraint. In general, if an abutment constraint $Vabut(X, Y)$ is violated, we will first try to move Y to the top of X by shuffling Y with the closest top neighbor of X in the Polish expression. If it is failed, e.g. all the top neighbors of X are fixed in position, we will try to move X to the bottom of Y by shuffling

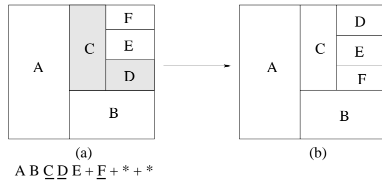


Figure 8: Shuffling modules to fix violated abutment constraints

X with the closest bottom neighbor of Y. The procedure for the horizontal direction is defined similarly. Notice that (Please refer to *Main*) we will not shuffle the modules back to their original positions if an expression is accepted, i.e. the constrained modules will stay together unless some later moves break them apart.

It is possible that some constraints are still violated after all the possible shufflings. We include an abutment constraint term in the total cost to penalize the remaining violated constraints. All violations will be eliminated as the annealing process proceeds in most of the cases.

4.3 Handling L-shaped and T-shaped Modules

Instead of partitioning into rectangular sub-modules, L-shaped and T-shaped modules are treated as single modules in the annealing process. They will be *expanded* to their original shapes when being packed and the expansions are dependent on their topological positions in the original Polish expression. After calculating the total area and interconnect cost, they are treated as single modules again in the floorplan transformation.

4.3.1 Expansion of L-shaped Modules

Consider an L-shaped module X in a Polish expression α, we will expand it into its sub-modules X_1 and X_2 by modifying the expression according to the relative position of X in α. There are four different cases as shown in Figure 9. The subtree labeled "1" can either be a basic module or a subtree of modules. We are trying to pack modules into the unoccupied area of the L-shaped modules. The L-shaped module is oriented differently in different cases so as to preserve as much as possible the relative position between all the other modules in the original Polish expression.

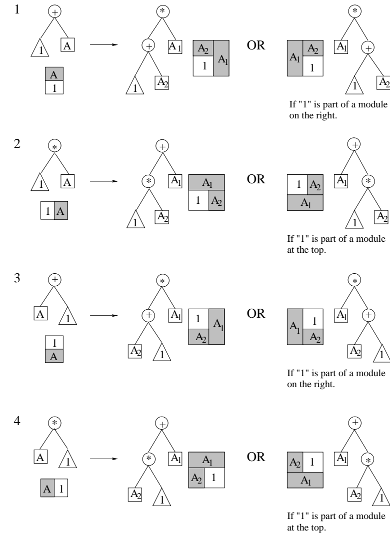


Figure 9: Expansion of an L-shaped Module

4.3.2 Expansion of T-shaped Modules

Similar to an L-shaped module, we will expand a T-shaped module X into its sub-modules X_1, X_2 and X_3 by modifying the Polish expression α according to the relative position of X in α. There are two different cases, depending on the sibling u of X in the slicing tree. If u is an internal node and the two children subtrees of u are not parts of the same module, we will pack the sub-modules of X with the children subtrees of u as shown in Figure 10 and 11. The subtree labeled "1" or "2" can either be a basic module or a subtree of modules. Again, we are trying to pack modules into the two unoccupied areas of the T-shaped module, and the T-shaped module is oriented differently in different cases to preserve as much as possible the relative positions between all the other modules in the original Polish expression. If u is a single basic module or that the two children subtrees of u belong to the same module (so we cannot pack them apart as shown in Figure 10 and 11), we will label C as a *degenerated* T-shaped module which will be expanded into its sub-modules as described in Figure 12.

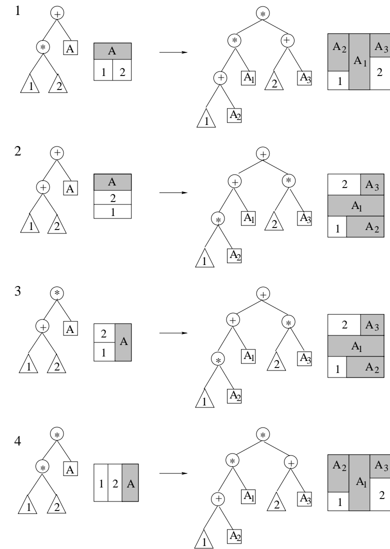


Figure 10: Expansion of a T-shaped Module which is a right child

4.3.3 Expansion Order

The result of the expansion will depend on the order in which the modules are expanded. An example is shown in Figure 13. Assume that both module A and B in the figure are L-shaped. Expanding B followed by A will give us the packing in (a), while expanding in the reverse order will give us the packing in (b). If the order is not defined well, we may need to scan the Polish expression once for each L-shaped or T-shaped module. In our implementation, we will first expand the T-shaped modules. This requires scanning the

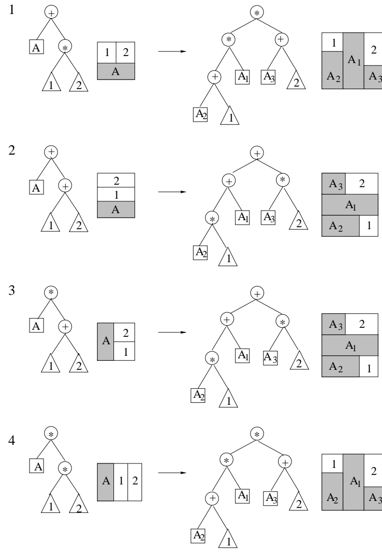


Figure 11: Expansion of a T-shaped Module which is a left child

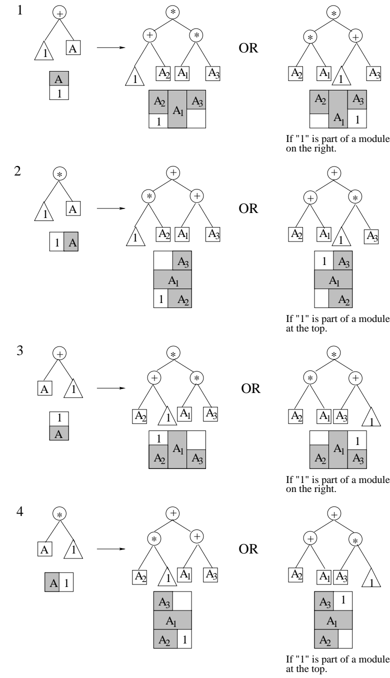


Figure 12: Expansion of a degenerated T-shaped Module

expression twice. The first scan expands all the T-shaped modules which are right children, and the second scan expands all the T-shaped modules which are left children. The degenerated T-shaped modules are labeled on the way. After these two scans, any T-shaped module will either be expanded or labeled as degenerated. We will then expand the remaining L-shaped modules and the degenerated T-shaped modules. This also requires scanning the expression twice. The first scan expands all the L-shaped modules or degenerated T-shaped modules which are right children, and the second scan expands those which are left children. The algorithm is described by the algorithm *Expansion* below. We need to scan the expression four times in total. An example of expansion is shown in Figure 14. In this example, module A is T-shaped and module B is L-shaped. A is expanded first because it is a T-shaped module and a right child. After that, we should expand the T-shaped modules which are left children followed by the L-shaped modules which are right children, but there is none of them. Finally, we will expand B which is an L-shaped module and a left child.

Algorithm Expansion

Input: *A Polish expression α with a set of modules $M = M_R \cup M_L \cup M_T$, where M_R is a set of rectangular, modules M_L is a set of L-shaped modules and M_T is a set of T-shaped modules.*

Output: *A Polish expression β with all the modules in M_L and M_T expanded to their corresponding sub-modules.*

1. Scan α from left to right and generate a new Polish expression α_1 by:
2. For any T-shaped module X which is a right child:
3. If the sibling u of X is an internal node and the children subtrees of u are not parts of one module:
4. Expand X as described in Figure 10.
5. Else:
6. Label X as a degenerated T-shaped module.
7. Scan α_1 from left to right and generate a new Polish expression α_2 by:
8. For any T-shaped module X which is a left child:
9. If the sibling u of X is an internal node and the children subtrees of u are not parts of one module:
10. Expand X as described in Figure 11.
11. Else:
12. Label X as a degenerated T-shaped module.
13. Scan α_2 from left to right and generate a new Polish expression α_3 by:
14. For any L-shaped module or degenerated T-shaped module X which is a right child:
15. If X is an L-shaped module:
16. Expand X as in case 1-2 of Figure 9.
17. Else:
18. Expand X as in case 1-2 of Figure 12.
19. Scan α_3 from left to right and generate a new Polish expression β by:
20. For any L-shaped module or degenerated T-shaped module X which is a left child:
21. If X is an L-shaped module:
22. Expand X as in case 3-4 of Figure 9.
23. Else:
24. Expand X as in case 3-4 of Figure 12.
25. Output β.

4.4 Time Complexity

We need to scan the Polish expression once to find the neighbors of every module. This takes $O(n)$ time where n is total number of modules. Then shuffling modules to fix violated abutment constraints takes another $O(nq)$ time where q is the total number of abutment constraints. Notice that this is only a worst case analysis. Usually, we do not need to scan all the modules once to find the closest module to shuffle with and the average time taken is just $O(q+n)$. To expand

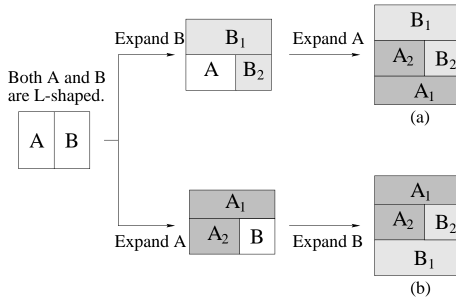


Figure 13: An Example Demonstrating the Effect of the Expansion Order

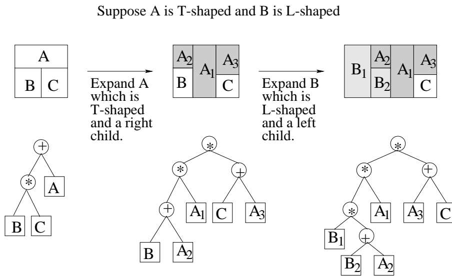


Figure 14: An Example of the Expansion Step

all the L-shaped and T-shaped modules, we need to scan the expression four times which takes $O(n)$ time. Therefore, the total time taken in each iteration of the annealing process to handle the abutment constraints and rectilinear blocks is $O(np)$ in the worst case, and $O(n+p)$ on the average.

4.5 Moves and Cost Function

We use the same set of moves (M1 M2 and M3) as in [5]. 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 and W is a half perimeter estimation of the interconnect cost. D is a penalty term for the violated abutment constraints. If an abutment constraint between two modules are violated, the corresponding penalty term is computed as the manhattan distance that one of the two module centers needs to move in order to make them abut. An example is shown in Figure 15. In this example, suppose A and B are constrained to abut horizontally, i.e. $Habut(A, B)$, but this constraint is violated and its corresponding penalty term D will be $x + y$ where x is the distance between the right boundary of A and the left boundary of B and y is the vertical distance between the centers of A and B. The penalty term is computed similarly in case of L-shaped or T-shaped modules by just considering the largest sub-modules in the rectilinear blocks. 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. We usually set γ large enough that D will drop rapidly right at the beginning.

5. EXPERIMENTAL RESULTS

We tested our floorplanner with some benchmark data: ami33, ami49 and playout. In all the data, the rectangular modules are soft modules with aspect ratios lying between 0.25 and 4.0 and the L-shaped or T-shaped modules are hard modules. For each experiment, the starting temperature is decided such that an acceptance ratio is 100% at the beginning. The temperature is lowered at a constant rate and the number of iterations in one temperature step is proportional to the number of modules. All the experiments were carried out on a 143 MHz UltraSPARC Workstation.

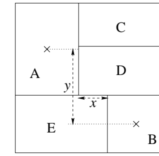


Figure 15: Penalty for violation of an abutment constraint

Scale: 1 unit = 23.8 pts

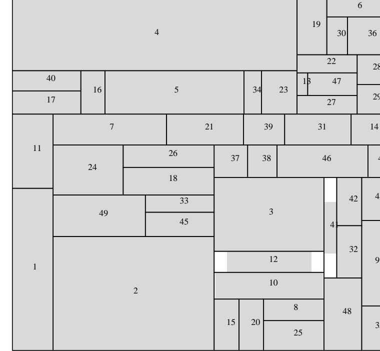


Figure 16: A result packing of ami49. Module 1, 2, 15, 20 and 25; 3, 41, 42 and 43 are required to abut horizontally. Module 25, 8, 10, 12 and 3; 43 and 44 are required to abut vertically. All constraints are satisfied.

We did two sets of experiments, one set with only rectangular modules and the other set with L-shaped or T-shaped modules. In the first set, we did five testings for each benchmark data, each testing with a different set of abutment constraints. The abutment constraints we imposed are usually that chains of four to five modules are required to abut horizontally or vertically. The averaged result for each benchmark data is shown in Table 1. We can see from the table that our method can handle abutment constraints efficiently. Figure 16 and 17 show two result packings.

In the second set of experiments, we modified the benchmark data by changing some modules to L-shaped or T-shaped. We called these data lt-ami33, lt-ami49 and lt-playout. Again, we did five testings for each data, imposing different abutment constraints on the modules for each testing. Table 2 summarizes the results. Figure 18 and Figure 19 show two result packings. The rectangular modules are white in color, the L-shaped modules are light greg and the T-shaped modules are dark grey.

6. REFERENCES

Scale: 1 unit = 4.5 pts

Figure 17: A result packing of playout. Module 1, 2, 50, 4, 5, 6 and 14 are required to abut horizontally, and module 8, 9, 10, 4, 11, 12 and 13 are required to abut vertically. Ten out of the twelve abutment constraints are satisfied.

Data Set	n	%Dead-space	Time (s)	#Viola-tion
ami33a	33	3.62	84.98	0
ami49a	49	2.02	164.51	0.2
playout	62	2.93	453.32	1.4

Table 1: Results of Testing Abutment Constraints with Rectangular Modules. Each data set has 12 modules having abutment constraint.

[1] M. Kang and W. W.M. Dai. General Floorplanning with L-shaped, T-shaped and Soft Blocks Based on Bounded Slicing Grid Structure. *IEEE Asia and South Pacific Design Automation Conference*, pages 265–270, 1997.

[2] T. Chang Lee. A Bounded 2D Contour Searching Algorithm for Floorplan Design with Arbitrarily Shaped Rectilinear and Soft Modules. *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 525–530, 1993.

[3] R.H.J.M. Otten. Efficient Floorplan Optimization. *IEEE International Conference on Computer Design*, pages 499–502, 1983.

[4] L. Stockmeyer. Optimal Orientations of Cells in Slicing Floorplan Designs. *Information and Control*, 59:91–101, 1983.

[5] 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.

[6] D.F. Wong and C.L. Liu. Floorplan Design for Rectangular and L-shaped Modules. *Proceedings IEEE International Conference on Computer-Aided Design*, pages 520–523, 1987.

[7] F.Y. Young and D.F. Wong. How Good are Slicing Floorplans. *Integration, the VLSI journal*, 23:61–73, 1997. Also appeared in ISPD-97.

[8] F.Y. Young and D.F. Wong. Slicing Floorplans with Boundary Constraints. *IEEE Asia and South Pacific Design Automation Conference*, pages 17–20, 1999.

Scale: 1 unit = 26.5 pts

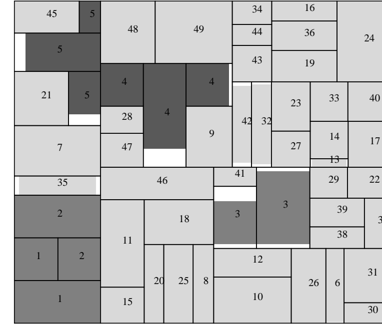


Figure 18: A result packing of lt-ami49. Module 1, 2, 15, 20, 25, 8 and 10 are required to abut horizontally and module 10, 12, 3, 41, 42, 43 and 44 are required to abut vertically. Eleven out of the twelve abutment constraints are satisfied.

Data Set	n	#L-Blocks	#T-Blocks	%Dead-space	Time (s)	#Viola-tion
lt-ami33a	33	2	1	5.20	78.49	1
lt-ami49a	49	3	2	5.38	294.34	1
lt-playout	62	3	3	3.91	807.89	1.4

Table 2: Results of Testing Abutment Constraints with L-shaped and T-shaped Modules. Each data set has 12 modules having abutment constraint.

Scale: 1 unit = 4.4 pts

Figure 19: A result packing of lt-playout. Module 1, 2 and 15; 8 and 9; 12, 13, 14 and 15 are required to abut horizontally. Module 9, 10, 11 and 12; 20, 17 and 48; 18 and 19 are required to abut vertically. Eleven out of twelve of the abutment constraints are satisfied.