the $\mathrm{cluster}(v)$ if the area of $\mathrm{cluster}(v) \cup \mathrm{group}(u,w)$ does not exceed the area constraint M. However, $u \in G_v - \mathrm{cluster}(v)$ implies that the area of $\mathrm{cluster}(v) \cup \mathrm{group}(u,w)$ exceeds M. When $u \in C_v$ and the area of $\mathrm{cluster}(v) \cup \mathrm{group}(u,w)$ is greater than M, there must exist an edge (f,g) such that $f \in \mathrm{group}(u,w)$, $g \in \mathrm{group}(u,w)$, $f \in (G_v - C_v)$ and $g \in C_v$. The situation is depicted in Fig. 4(b). Based on the induction hypothesis, $\mathrm{delay}(f) + \Delta(g,v) + D \geq l(f) + \Delta(g,v) + D$. Then, by *P3* of Lemma 1, $\mathrm{delay}(v) \geq \mathrm{delay}(f) + \Delta(g,v) + D \geq l(f) + \Delta(g,v) + D > l(u) + \Delta(w,v) + D = l(v) > \mathrm{delay}(v)$ which is impossible.

c) If $(\mathrm{cluster}(v) - C_v) \neq \phi$, we can divide $\mathrm{cluster}(v)$ into two disjoint subsets $\mathrm{cluster}(v) - C_v$ and $\mathrm{cluster}(v) \cap C_v$. By Corollary 1, there exists an edge (s,t) such that $s \in (\mathrm{cluster}(v) - C_v)$ and $t \in \mathrm{cluster}(v) \cap C_v$ while $l(s) + \Delta(s,t) + D = l''(s,t) \geq l_2(v) = l(v)$. This is depicted in Fig. 4(c). Since we know $\mathrm{delay}(v) \geq \mathrm{delay}(s) + \Delta(s,t) + D$, due to the induction hypothesis that $\mathrm{delay}(s) + \Delta(s,t) + D \geq l(s) + \Delta(s,t) + D$, we have

$$\begin{aligned}\mathrm{delay}(v) \geq & \mathrm{delay}(s) + \Delta(s,t) + D \\ \geq & l(s) + \Delta(s,t) + D \geq l(v)\end{aligned}$$

which contradicts the assumption $\mathrm{delay}(v) < l(v)$. As a result, the statement is also true for vertex v. □

Lemma 4: In our algorithm, for any vertex v in the clustering S generated by the clustering phase (lines 13–19), the path delay at v is less than or equal to $l(v)$.

Proof: Our delay model is different from that in [1], but the clustering phase in our algorithm is the same as that of [1], so the proof is the same. Details can be found in [1]. □

Based on Lemma 3 and Lemma 4, we can easily derive the following theorem.

Theorem 1: The clustering S generated in our algorithm is an optimal clustering for any instance of the problem described in Section II.

Proof: In Lemma 3, it is shown that for each vertex v, the label $l(v)$ in our algorithm is less than or equal to the path delay at vertex v in any optimal clustering; Lemma 4 states that our algorithm is able to generate a clustering with the path delay at v less than or equal to $l(v)$ which is the lower bound of the path delay at vertex v in any optimal clustering. Together with Lemma 1 in [1], the clustering S generated by our algorithm is an optimal clustering. □

We analyze the complexity of our algorithm. In $\mathrm{Grouping}()$, $\mathrm{Group\_vertex}()$ would run at most $|V|$ times, so the time complexity of the WHILE loop is $O(|V||E|)$. In $\mathrm{Circuit\_clustering}()$, finding the maximum delay matrix Δ takes $O(|V|(|V| + |E|))$, finding a topological order in line 4 takes $O(|V| + |E|)$ time, the sorting in line 11 takes time $O(|E|lg(|E|))$, and $\mathrm{Labeling}()$ takes only $O(|E|)$ time. So, the first WHILE loop of $\mathrm{Circuit\_clustering}()$ takes $O(|V|(|E|lg(|E|) + |V||E|))$ time. Clustering phase (lines 13–19) takes time $O(|V| + |E|)$. So the overall time complexity is $O(|V|(|E|lg(|E|) + |V||E|)) = O(|V|^2|E|)$.

Remarks: In fact, our algorithm can also handle the case where the intercluster delay D is a variable value (say $D(x,y), \forall (x,y) \in E$). It is because the calculation of $l''(x,y) = l(x) + D + \Delta(y,r)$ includes the value of D such that if D becomes a variable $D(x,y)$, the calculation becomes $l''(x,y) = l(x) + D(x,y) + \Delta(y,r)$, and still correctly represents the situation when (x,y) becomes an intercluster edge. Besides, the optimality of the algorithm still holds because all the theoretical results remain true and can be proved similarly.

VII. Conclusion

In this paper, we have introduced a new delay model which is more general and practical than the general delay model [3]. Under our new delay model, a circuit clustering algorithm based on a novel vertex grouping technique is proposed and is proved to optimally solve the area-constrained combinational circuit clustering problem for delay minimization in polynomial time.

References

[1] R. Rajaraman and D. F. Wong, "Optimum clustering for delay minimization," *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 1490–1495, Dec. 1995.

[2] H. Yang and D. F. Wong, "Circuit clustering for delay minimization under area and pin constraints," *IEEE Trans. Computer-Aided Design*, vol. 16, pp. 976–986, Sept. 1997.

[3] R. Murgai, R. K. Brayton, and A. Sangiovanni-Vincentelli, "On clustering for minimum delay/area," in *Proc. IEEE Int. Conf. Computer-Aided Design*, 1991, pp. 6–9.

[4] E. Lawler, K. Levitt, and J. Turner, "Module clustering to minimize delay in digital networks," *IEEE Trans. Comput.*, vol. C-18, pp. 47–57, Jan. 1966.

Slicing Floorplan With Clustering Constraint

W. S. Yuen and Evangeline F. Y. Young

***Abstract*—In floorplan design, it is useful to allow users to specify some placement constraints in the final packing. Clustering constraint is a popular type of placement constraint in which a given set of modules are restricted to be placed adjacent to one another. The wiring cost can be reduced by placing modules with a lot of interconnections closely together. Designers may also need this type of constraint to restrict the positions of some modules according to their functionalities. In this paper, a method addressing clustering constraint in slicing floorplan will be presented. We devised a linear time algorithm to locate neighboring modules in a normalized Polish expression and to rearrange them to satisfy the given constraints. Experiments were performed on some benchmarks and the results are very promising.**

***Index Terms*—Clustering constraint, design floorplanning, floorplanning, physical design, very large scale integrated computer-aided design (VLSI CAD).**

I. Introduction

Floorplan design is the problem of planning the positions and shapes of a set of modules on a chip in order to optimize the circuit performance at a very early designing stage. During this floorplanning phase, circuit performance like layout area, interconnect cost, heat dissipation and power consumption, etc., should be taken into consideration.

Manuscript received December 3, 2001; revised May 17, 2002. This paper was recommended by Associate Editor T. Yoshimura.

The authors are with the Department of Computer Science and Engineering, Chinese University of Hong Kong, Shatin, New Territories, Hong Kong (e-mail: fyyoung@cse.cuhk.edu.hk).

Digital Object Identifier 10.1109/TCAD.2003.810738

There are three kinds of floorplans: slicing, nonslicing, and mosaic floorplan. A slicing floorplan is one that can be obtained by recursively partitioning a rectangle in two by either a vertical or a horizontal line. The advantages of using slicing floorplan are that it has simple representations such as slicing tree and Polish expression [7], [8], and optimal module shaping can be done efficiently. However, slicing floorplan is only a small subset of all feasible packings and is not general enough. A nonslicing floorplan is not necessarily slicing, and can represent any kind of packing. Several methods, sequence-pair [5], bound-sliceline-grid (BSG) [6], O-tree [2], B*-tree [1], and transitive closure graph [4], have been proposed for representation of nonslicing floorplan. The paper [3] proposes a new type of floorplan called mosaic floorplan that is similar to general nonslicing floorplan except that it does not have any unoccupied rooms. A representation called corner block list can be used to represent mosaic floorplan.

A floorplanning algorithm has to deal with several essential issues, including module shaping, routability, area and delay, in order to optimize the circuit performance. With the scaling down of the IC technology, the number of transistors that can be built into a standard size chip has increased rapidly, and it has become increasingly important to consider the circuit performance as early as possible in the floorplanning stage. Placement constraints in floorplan design are useful for restricting the relative positions between the modules according to their functionalities in order to improve the circuit performance like interconnect cost and delay, etc. Some previous works on preplace constraint and range constraint in slicing floorplan [12], [11] have been done. Clustering constraint is another popular type of constraint in which a given set of modules are restricted to be placed adjacent to one another. The wiring cost can be reduced by placing modules with a lot of interconnections closely together. Designers may also need this type of constraint to restrict the positions of some modules according to their functionalities. The paper [9] proposed a hybrid floorplanning method using partial clustering and module restructuring. Their method restricts clusters to be placed in rectangular regions (subtrees in a slicing tree). Since constrained modules are restricted to be placed in rectangular regions, the packing topology is limited and the deadspace of the final floorplan is usually large. The paper [10] proposed a unified method to handle different kinds of placement constraints in general floorplan. However, their method does not handle the clustering constraint as defined here.

In this paper, clustering constraint is considered in which some modules are required to be placed next to each other. The cost of routing can be reduced by imposing clustering constraints to those modules which are heavily connected. The method we used will determine the surrounding positions of a target module in a Polish expression and swap the constrained modules into those positions in order to satisfy the constraints. Our method can also be extended to handle more than one cluster in a floorplan. This paper is organized as follows. We will define the problem in Section II. In Section III, a slicing floorplanner on which our method is based will be described. In Section IV, detailed descriptions of the clustering method will be given. Results will be shown in Section V.

II. Problem Definition

A floorplan with n modules $(1, 2, \ldots, n)$ is an enveloping rectangle R subdivided by horizontal and vertical line segments into n nonoverlapping rectilinear regions such that each region R_i must be large enough to accommodate the corresponding module i.

In most iterative methods, a floorplan is evaluated by a function $A + \lambda W$, where A is the area of the floorplan and W is the total wire length. The aspect ratio of each module will be limited so that the delay inside each module will not be too long. For each rectangular module i, there

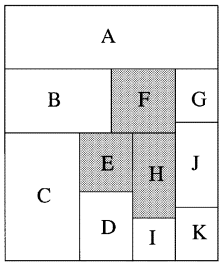

Fig. 1. Example of clustering constraint.

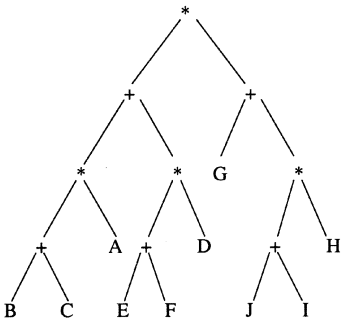

Fig. 2. Slicing tree.

are three input values A_i, r_i and s_i. A_i is the area of the module, and r_i and s_i are the minimum and maximum aspect ratio of the module respectively. Let w_i and h_i be the width and height of the module, then $A_i = w_i h_i$ and $r_i \leq h_i/w_i \leq s_i$. The overall aspect ratio of the floorplan is also required to be within a given range.

In this paper, clustering constraint is considered in floorplan design. Given a set of modules Φ and a subset of modules $\Delta \subseteq \Phi$, we want to pack the modules in Φ such that the modules in Δ will be adjacent to each other. Fig. 1 shows an example of clustering constraint. Modules E, F, and H are the subset of modules to be clustered and they have to be placed adjacent to each other in the final packing. The floorplanning problem with clustering constraints is defined as follows.

A. Problem FP/CC

Given a set of n modules $\Phi = \{m_1, m_2, \ldots, m_n\}$ and $m_i = (A_i, r_i, s_i)$ for $i = 1, \ldots, n$ where A_i is the area of module i, and r_i and s_i are the minimum and maximum aspect ratio of module i respectively. Let Δ be a subset of the modules in Φ, pack the modules in Φ to minimize the total area and interconnect cost such that the following three conditions are satisfied.

1) *The modules in Δ will form a cluster (lying adjacent to each other) in the final packing.*
2) *Each module satisfies its area and aspect ratio constraint.*
3) *The aspect ratio of the whole packing is within a given range $[r, s]$.*

III. Basic Slicing Floorplanner

Our work is based on a well-known slicing floorplanner [8]. A slicing floorplan can be represented by a binary tree. The leaf nodes of the tree are the basic modules and the internal nodes are labeled either with a + or a * operator to represent a horizontal or a vertical cut, respectively. An example is shown in Fig. 2. Reading the tree in postorder, a Polish expression will be obtained that is used to represent the floorplan structure in the algorithm. A normalized Polish expression is a Polish expression with no consecutive identical operators.

Simulated annealing is used to optimize the total area and interconnect cost of the floorplan. The cost is computed as $A + \lambda W$ where A is

the total area and W is the wire length. Normalized Polish expression is used to represent a solution packing during the annealing process. The neighbors of each solution have to be defined such that the optimal solution is reachable. Three types of moves M1, M2, and M3 are used. M1 swaps two adjacent operands in the expression, M2 interchanges the operators in a chain (a chain is a substring of operators in the Polish expression) and M3 swaps two adjacent operand and operator.

IV. Floorplanning With Clustering Constraint

In this paper, we consider clustering constraint in slicing floorplan. A simple method to solve this problem is by adding a term in the cost function of the annealing process as a penalty for violating the constraint. This method was tested but the results are poor and the constraints will usually be violated in the final packing. In our approach, clusters are maintained throughout the whole annealing process.

We devised a method to locate the surrounding positions of a target module. A target module M_t is picked randomly from the subset Δ given by the user. A set of M_t's surrounding modules Π_t in the packing is then obtained. For each $M_i \in \Pi_t$, if $M_i \notin \Delta$, we will swap M_i with some M_j where $M_j \in \Delta \setminus \Pi_t$. This algorithm will maintain the cluster in the floorplan throughout the whole annealing process. An overview of the algorithm is given below:

```
Main Program
begin
 Initialize temperature T
 While T ≥ Threshold do
 begin
  Move by either M1, M2, or M3
  Call procedure Clustering
  Compute Cost
  If Cost is reduced
   Accept the move
  Else
   Prob = min(1, e^{-kΔ_c/T})
    where Δ_c = change in cost, k=constant
   If Random(0,1) ≤ Prob then
    Accept the move
   Else
    Reject the move
 end
end
```

A. Locating Surrounding Modules

An algorithm is devised to locate the surrounding positions of a target module in a normalized Polish expression. We define the *surrounding set* of a module M_i as

Definition: Given a module M_i in a slicing floorplan F, the surrounding set Π_i of M_i is a set of modules in F such that a module M_j is in Π_i if and only if (1) M_j is above (below) M_i and there is no module M_k in F where M_k is above (below) M_i and M_j is above (below) M_k, or (2) M_j is on the right (left) of M_i and there is no module M_k in F where M_k is on the right (left) of M_i and M_j is on the right (left) of M_k.

Note that we can locate the modules in the surrounding set of a given module in linear time by just looking at the Polish expression and no real packing is needed. A target module $M_t \in \Delta$ is selected randomly. A set Π_t is found such that M_t is surrounded by the modules in Π_t in the packing. An example is shown in Fig. 3. In this example $M_t = F$ and $\Pi_t = \{A, B, C, E, G, H\}$.

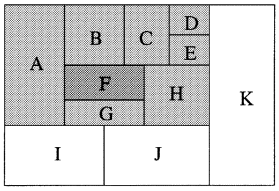

Fig. 3. Example of the surrounding modules of F.

For each module M_t in the slicing floorplan, M_t is surrounded by at most four cuts which correspond to four operators in the normalized Polish expression. If those four operators are located in the Polish expression, the set of modules Π_t of M_t can be found. For a Polish expression $\alpha = \alpha_1\,\alpha_2,\,\ldots,\,\alpha_n$, we define a *valid subexpression* $\beta = \alpha_k\,\alpha_{k+1},\,\ldots,\,\alpha_{k+m}$ where $k \geq 1$ and $n \geq k+m$ as a subexpression in α such that α_k must be an operand and the number of operands is equal to the number of operators plus one in β. A valid subexpression indeed represents a subtree in the whole slicing tree.

The two operators, + and *, correspond to cuts of different orientations. Let ζ, δ, and γ be valid subexpressions in a Polish expression. Some terms are defined as follows:

i) $\text{below}(\delta,\zeta) \iff \gamma = \zeta\delta+$
ii) $\text{above}(\delta,\zeta) \iff \gamma = \delta\zeta+$
iii) $\text{left}(\delta,\zeta) \iff \gamma = \zeta\delta*$
iv) $\text{right}(\delta,\zeta) \iff \gamma = \delta\zeta*$.

Given a target module M_t, the algorithm *Find_Surrounding* will find four valid subexpressions a, b, c, and d such that $\text{below}(\delta_1, a)$, $\text{above}(\delta_2, b)$, $\text{left}(\delta_3, c)$, and $\text{right}(\delta_4, d)$, and δ_i for $i = 1,\ldots,4$ is the smallest valid subexpression containing M_t and satisfying the corresponding relationship.

```
Algorithm : Find_Surrounding(M_t, α)
Input: α = α_1 α_2, ..., α_{2n-1} is a Polish ex-
 pression of the original packing.
 t is the index of the target module.
Output: Valid subexpressions a, b, c, and
 d such that below(δ_1,a), above(δ_2,b), left(δ_3,c),
 and right(δ_4,d), and δ_i for i = 1,...,4 is the
 shortest valid subexpression containing
 M_t and satisfying the above relation-
 ship.
1 first = end = t
2 While (a, b, c, d are not all found) and
 (first ≥ 1) and (end ≤ 2n − 1)
3 begin
4  If α_{end+1} is an operator
5  begin
6   Find k such that
7    e = α_{first−k} α_{first−k+1},..., α_{first−1}
8   is the shortest valid subexpression
9   If (α_{end+1}=+) and (a is not found yet)
10     a = e
11   Else if (α_{end+1}=*) and (c is not found
 yet)
12     c = e
13   first = first − k; end = end + 1
14  end
15  Else
16  begin
17   Find k such that
18    e = α_{end+1} α_{end+2},..., α_{end+k}
19   is the shortest valid subexpression
```

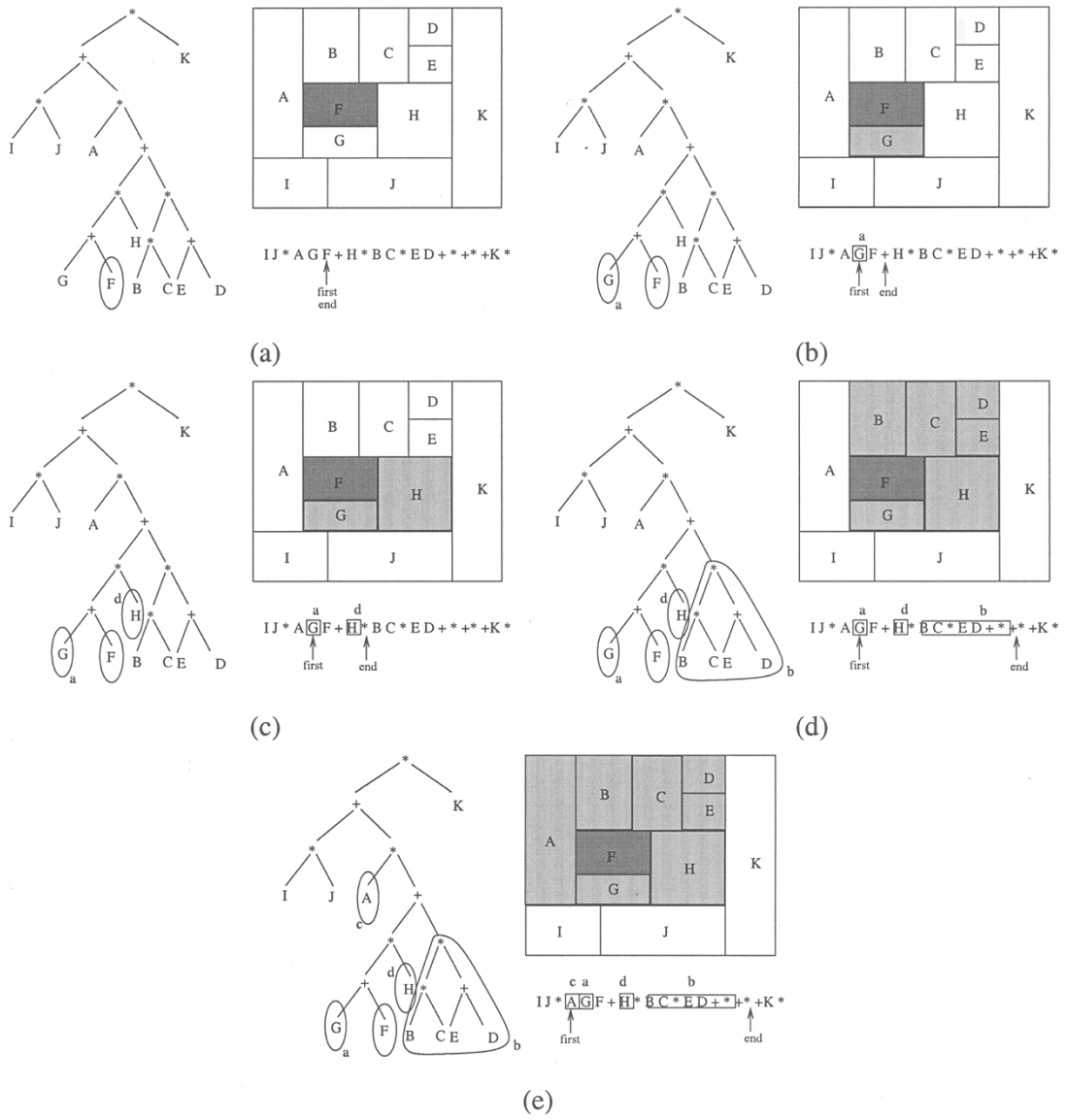

Fig. 4. Example to illustrate the algorithm Find_Surrounding.

```
20   If (α_{end+k+1}=+) and (b is not
 found yet)
21     b = e
22   Else if (α_{end+k+1}=*) and (d is not
 found yet)
23     d = e
24   end = end + k + 1
25  end
26 end
```

The complexity of this algorithm is $O(n)$. Fig. 4 illustrates the steps of the algorithm. The expressions a, b, c, and d are valid subexpressions representing subtrees in the slicing tree. For the example in Fig. 3, $M_t = F$, $a = G$, $b = BC*ED+*$, $c = A$, and $d = H$. The shortest valid subexpression can be obtained by counting the number of operators and operands. Note that not all the basic modules in a, b, c, and d belong to the surrounding set Π_t of M_t. For example, the subexpression b is lying above M_t and only the modules lying at the bottom of the supermodule corresponding to b belong to Π_t. Fig. 5 shows an example in which $b = BC*ED+*$ but D does not belong to Π_t. A recursive procedure can be used to find Π_t efficiently given a, b, c, and d. The following procedure is for finding the modules in Π_t from the subexpression lying below the target module, i.e., from the subexpression a. Procedures for the other three subexpressions can be constructed similarly.

```
Procedure : Mark_Neighbor_Below(first, end)
Input: first is the first index of the valid
 subexpression a.
 end is the last index of a and first ≤ end.
Output: Π is the set of modules lying at
 the top of the supermodule represented by
 a.
1 If first = end
2   Π = Π ∪ α_{first}
3 Else
4 begin
5  Find k such that e = α_{end-k} α_{end-2} ... α_{end-1}
 is the shortest valid subexpression
6  If (α_{end} = *)
```

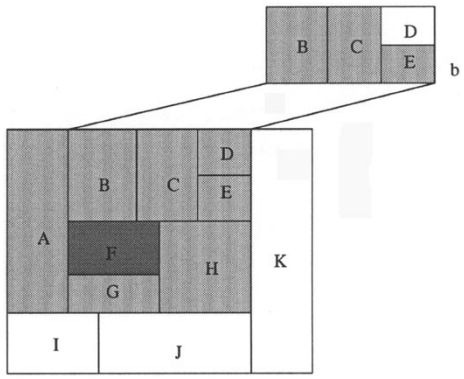

Fig. 5. D does not belong to Π_t where M_t is F.

```
7   Mark_Neighbor_Below(first, end − k − 1)
8   Mark_Neighbor_Below(end − k, end − 1)
9  Else if (α_end = +)
10   Mark_Neighbor_Below(end − k, end − 1)
11 end
```

B. Constraint Satisfaction

In the annealing process, all the given constraints have to be satisfied in order to make the floorplan feasible. Modules in the constraint set Δ will be swapped with the surrounding set Π_t until the condition $\Delta \subseteq \{M_t\} \cup \Pi_t$ is satisfied.

In the first iteration, M_t is randomly selected from Δ. An intersection set Υ is defined to be $\Delta \cap \Pi_t$. If $|\Delta| > |\Upsilon| + 1$, swapping is needed to satisfy the clustering constraints. If $|\Delta| > |\Pi_t| + 1$, there is not enough room for swapping, and the whole process will be repeated recursively by selecting another module that is already in the cluster as the new target module until all the constraints are satisfied. All three moves in the simulated annealing process can affect the neighboring structure and may lead to infeasible packings. However we will swap operands in the Polish expression to maintain a feasible solution throughout the whole annealing process. There is only one case in move M1 that does not affect the feasibility of the packing, i.e., if two adjacent operands to be swapped are both in Δ or both in $\Phi - \Delta$. Modification is not required in this case and the clustering constraints will not be violated after the move.

The following algorithm describes the strategy to ensure that all the clustering constraints are satisfied throughout the annealing process.

```
Algorithm : Clustering(α, Δ)
Input: α = α1 α2, ..., α2n−1 is a Polish ex-
 pression of the packing.
 Δ is the set of modules having clus-
 tering constraint.
Output: Modified α that represents a fea-
 sible floorplan satisfying the given
 clustering constraint
1 count = 0
2 Candidate=Δ
3 While count < |Δ| − 1
4 begin
5  For each M_i ∈ Candidate
6  begin
7   Find Π_i by:
8    Initialize Π_i as empty
9    Call Find_Surrounding(M_i, α)
10   Call Mark_Neighbor_Below(first(a), end(a)).
11   Call Mark_Neighbor_Above(first(b), end(b)).
12   Call Mark_Neighbor_Left(first(c), end(c)).
13   Call Mark_Neighbor_Right(first(d), end(d))
14   Υ_i = Δ ∩ Π_i
15   If |Υ_i| + 1 ≥ |Δ|
16    All clustering constraints are sat-
 isfied.
17    Return
18  end
19  Take i where |Υ_i| is maximum and M_i ∈
 Candidate is not marked
20  If count = 0, put Clustering = {·}
21  Add the modules in Υ_i to Candidate
22  If |Π_i| ≥ |Δ| − 1
23  begin
24   For each M_k ∈ Δ ∩ Π̄_i
25    Find M_j ∈ Π_i ∩ Δ̄
26    Swap(M_j, M_k)
27   count = |Δ| − 1
28  end
29  Else
30  begin
31   For each M_k ∈ Π_i ∩ Δ̄
32    Find M_j ∈ Δ ∩ Π̄_i and M_j ∉ Clustering
33    Swap(M_j, M_k)
34    Add M_j to Clustering
35    count = count + 1
36  end
37  Mark M_i
38 end
```

If $|\Pi_i| < |\Delta| - 1$ (lines 30–36), the number of positions in Π_i is not enough for accommodating all the constrained modules. Other target modules will be selected and the process will be repeated until all the constraints are satisfied. The algorithm can handle even large cluster size.

C. Multiclustering Extension

Multiclustering constraint allows users to have more than one cluster in the final packing. The algorithm described above handles only one cluster. Multiclustering constraint can be handled by invoking the above algorithm several times. However, the major problem is that the surrounding sets of the target modules in different clusters can overlap. Infeasible packing will be resulted if modules are swapped randomly. For example, given two clustering sets Δ_1 and Δ_2, a target module M_{t_1} and M_{t_2} is picked from each clustering set. Let Π_{t_1} and Π_{t_2} be the surrounding sets of M_{t_1} and M_{t_2}, respectively. If a module M_k exists such that $M_k \in \Pi_{t_1}$ and $M_k \in \Pi_{t_2}$, this module M_k should be removed from either Π_{t_1} or Π_{t_2}. Otherwise, the swapping space provided by M_k may be used twice.

D. Cost Function

The cost function is computed as $A + \lambda W + \beta C$ where A is the total area of the packing, W is the half perimeter estimation of the wire length, and C is a penalty term for the clustering constraint. The penalty term C is the sum of squares of the center to center distances between every pair of modules in the same cluster. The penalty term helps in generating a packing in which the constrained modules will be placed as close to each other as possible. The parameters λ and β are constants that control the relative importance of the three terms.

TABLE I
RESULTS OF THE CONTROL EXPERIMENTS

Data Set	n	% Deadspace	Time (sec)
ami33	33	2.45	12.7
ami49	49	3.00	37.5
playout	62	4.35	124.4

TABLE II
RESULTS OF TESTING WITH ONE CLUSTER FOR THE MCNC EXAMPLES

Data Set	n	Cluster Size	% Deadspace	Time (sec)
ami33-cc1	33	7	1.62	19.1
ami33-cc2	33	7	2.80	18.4
ami33-cc3	33	7	2.74	22.6
ami49-cc1	49	10	4.65	53.4
ami49-cc2	49	10	3.53	53.2
ami49-cc3	49	10	4.04	51.9
playout-cc1	62	12	8.44	146.5
playout-cc2	62	12	7.43	147.8
playout-cc3	62	12	6.57	146.4

V. EXPERIMENTAL RESULTS

We tested our method with three Microelectronics Center of North Carolina (MCNC) building blocks examples (ami33, ami49 and playout). Ami33 has 33 modules and 123 nets. Ami49 has 49 modules and 408 nets. Playout has 62 modules and 1161 nets. A control experiment without any clustering constraint was performed for each data set and the results are shown in Table I. The temperature was decreased at a constant rate (0.9), and the number of iterations at each temperature step was one hundred times the number of modules. All experiments were performed on an UltraSPARC-II 400-MHz processor.

In the first set of experiments, 20% of the modules in each benchmark were selected randomly to have clustering constraint. (Ami33, ami49, and playout have 7, 10, and 12 constrained modules, respectively.) For each benchmark, we repeated the experiment three times by selecting different modules into the constraint set. The results are shown in Table II. We can see that the clustering constraint can be satisfied in all the experiments with only a small increase in total area and execution time. This has demonstrated the effectiveness of our method in handling a single cluster in slicing floorplan.

In the second set of experiments, we tested our method with multiclustering constraints. In each benchmark problem, we picked three, four, and five clusters, each having 2–7 modules. The results are shown in Table III. If we compare Table III with Table II, we can see that our method can be extended to handle more than one cluster successfully without imposing much penalty in the floorplan quality and the execution time. Figs. 6 and 7 show a result packing of ami33 with three clusters and a result packing of ami49 with four clusters, respectively.

TABLE III
RESULTS OF TESTING WITH MULTICLUSTERS FOR THE MCNC EXAMPLES

Data Set	n	# of Clusters (Cluster Sizes)	% Deadspace	Time (sec)
ami33-mc1	33	3(4,4,3)	2.35	21.0
ami33-mc2	33	4(3,3,3,2)	3.16	21.4
ami33-mc3	33	5(3,2,2,2,2)	2.17	21.9
ami49-mc1	49	3(6,5,5)	3.83	57.8
ami49-mc2	49	4(4,4,4,4)	2.77	56.7
ami49-mc3	49	5(4,3,3,3,3)	3.99	57.1
playout-mc1	62	3(7,7,6)	8.28	156.9
playout-mc2	62	4(5,5,5,5)	7.18	154.6
playout-mc3	62	5(4,4,4,4,4)	5.87	151.8

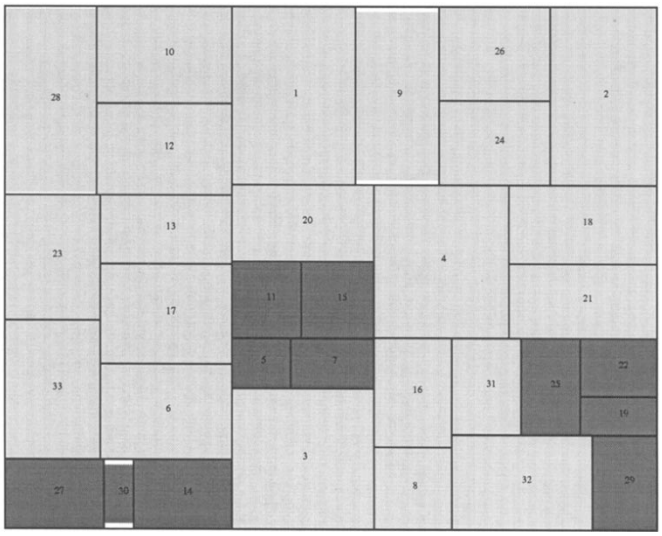

Fig. 6. Result packing of ami33 with three clusters (C_1:5,7,11,13; C_2:14,27,30; C_3:19,22,25,29).

Figs. 8 and 9 show the improvement in interconnection by imposing clustering constraints. We observed from the data set ami33 that modules 15, 18, 19, 20, 21, 24, and 25 are heavily connected with each other, so we imposed a clustering constraint between them. Figs. 8 and 9 show the result packings with and without the clustering constraint. One can see that the interconnect cost in Fig. 8 is much smaller than that in Fig. 9. This example demonstrated that with careful selection of the clustering constraints, our method can be used to reduce the interconnect cost of a floorplan by constraining those strongly connected modules in a cluster.

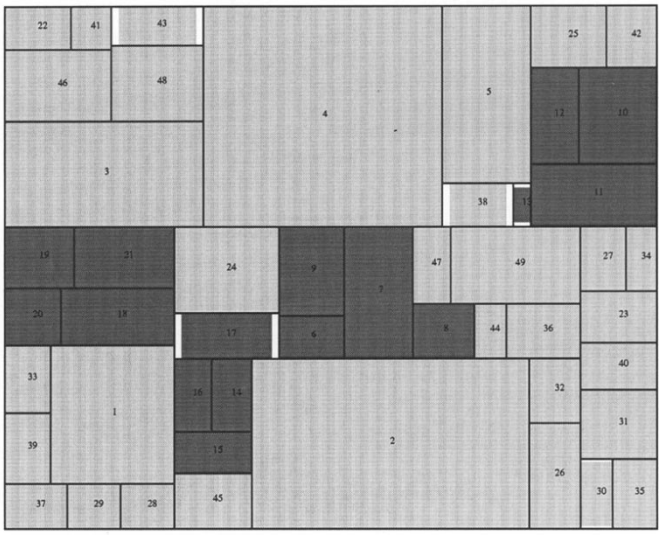

Fig. 7. Result packing of ami49 with four clusters (C_1:6,7,8,9; C_2:10,11,12,13; C_3:15,16,17,18; C_4:18,19,20,21).

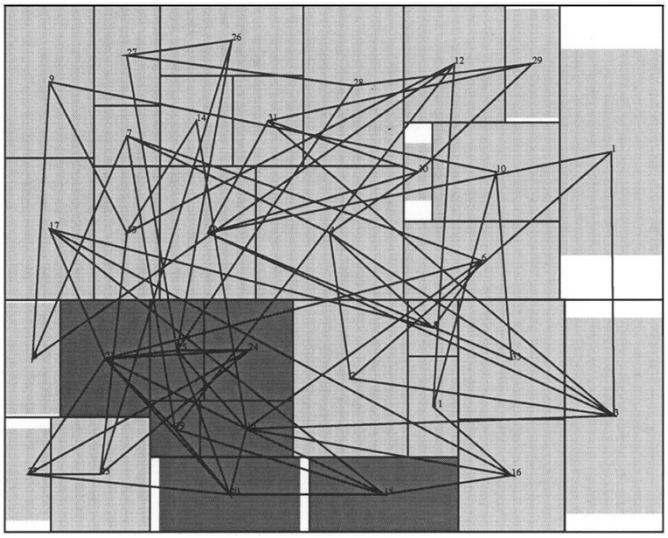

Fig. 8. Result packing showing the improvement in interconnection by imposing clustering constraints (wire length $= 0.1472 \times 10^6$ units).

VI. Conclusion

We have devised and implemented an efficient method to handle clustering constraint in slicing floorplan. Experimental results showed that our method can handle clustering constraint effectively without imposing much penalty in the quality of the floorplan and the execution time. In addition, this method can be extended to handle multiclusters. We have also demonstrated how the method can be used to reduce interconnect cost by imposing clustering constraint between those modules that are strongly connected with one another.

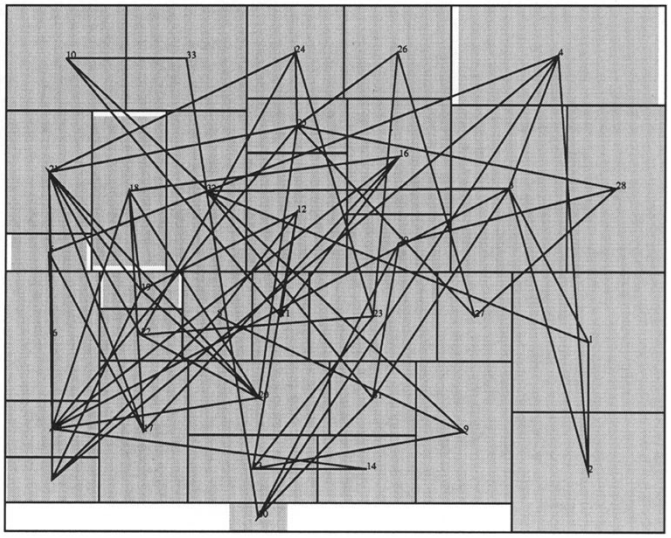

Fig. 9. Result packing of the same problem in Fig. 8 without imposing any clustering constraint (wire length $= 0.1596 \times 10^6$ units).

References

[1] Y. C. Chang, Y. W. Chang, G. M. Wu, and S. W. Wu, "B*-trees: A new representation for nonslicing floorplans," in *Proc. 37th ACM/IEEE Design Automation Conf.*, 2000, pp. 458–463.

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

[3] X. Hong, G. Huang, Y. Cai, J. Gu, S. Dong, C.-K. Cheng, and J. Gu, "Corner block list: An effective and efficient topological representation of nonslicing floorplan," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 2000, pp. 8–12.

[4] J.-M. Lin and Y.-W. Chang, "TCG: A transitive closure graph-based representation for nonslicing floorplans," in *Proc. 19th ACM/IEEE Design Automation Conf.*, 2001, pp. 764–769.

[5] H. Murata, K. Fujiyoshi, and M. Kaneko, "VLSI/PCB placement with obstacles based on sequence-pair," *IEEE Trans. Computer-Aided Design*, vol. 17, pp. 60–68, Jan. 1998.

[6] S. Nakatake, K. Fujiyoshi, H. Murata, and Y. Kajitani, "Module placement on BSG-structure and IC layout applications," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, Nov. 1996, pp. 484–493.

[7] R. H. J. M. Otten, "Automatic floorplan design," in *Proc. 19th IEEE/ACM Int. Conf. Computer-Aided Design*, 1982, pp. 261–267.

[8] D. F. Wong and C. L. Liu, "A new algorithm for floorplan design," in *Proc. 23rd ACM/IEEE Design Automation Conf.*, June 1986, pp. 101–107.

[9] T. Yamanouchi, K. Tamakashi, and T. Kambe, "Hybrid floorplanning based on partial clustering and module restructuring," in *Proc. Int. Conf. Computer-Aided Design*, 1996, pp. 478–483.

[10] E. F. Y. Young, C. C. N. Chu, and M. L. Ho, "A unified method to handle different kinds of placement constraints in floorplan design," in *Proc. 7th Asia South Pacific Design Automation Conf./15th Int. Conf. VLSI Design*, 2002, pp. 661–667.

[11] F. Y. Young and D. F. Wong, "Slicing floorplans with boundary constraints," in *Proc. IEEE Asia South Pacific Design Automation Conf.*, 1999, pp. 17–20.

[12] ——, "Slicing floorplans with range constraints," in *Proc. Int. Symp. Physical Design*, 1999, pp. 97–102.