Chapter 3

Algorithm for Cell Placement

Based on our motivation in Chapter 2, we propose two standard cell placement algorithms to minimize wirelength overhead and sleep transistor size. The first one is a functionality directed placement algorithm and the second one is a direct placement with iterative cell-moving algorithm.

In section 3.1, we first introduce the layout style of standard cells with sleep transistor insertion. Then, the detailed description of our placement algorithms will be presented in section 3.2 and section 3.3.

3.1 Standard-Cell Layout Style with Sleep Transistor

In this section, we introduce the standard-cell layout style in Section 3.1.1 and show how to perform cell characterization to compute the allowed maximum current flowing through the sleep transistor under a specified per-

formance degradation in 3.1.2.

3.1.1 Layout Style

To incorporate the MTCMOS layout style to the traditional standard cell style, the cell library is required to redesign. One way is to re-draw the cell from scratch and the other is to modify the existing library. The former has the advantage of compact layout and the latter takes less design effort. We will take the second way.

Figure 3.1 is the structure of standard cells used in MTCMOS layout style where the structure of a regular standard cell with low V_{th} is shown in Figure 3.1(a), and the structure of a sleep transistor with high V_{th} is shown in Figure 3.1(b) [6]. This second structure is proposed to modify a regular standard cell library so that the cell can be used for MTCMOS. MTCMOS layout require each cell connects to a virtual ground and in term the virtual ground connects to sleep transistors.

A cell in a regular cell library is modified so that it is a low V_{th} transistors and is connected to $VGND$ (virtual ground) through layer 2 where the GND remains on layer 1. A new cell for sleep transistors is designed where a high V_{th} transistor is used and connected to GND. The output of this sleep transistor cell is connected to the VGND through layer 2. Only one unit-size sleep transistor cell is designed. Figure 3.2 shows an example of a row

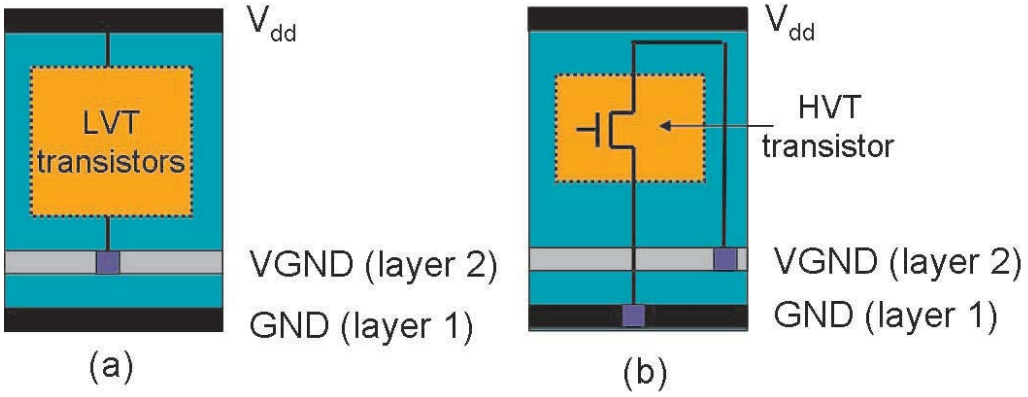


Figure 3.1: (a) Low V_{th} standard cell (b) High V_{th} sleep transistor cell

of connected standard cells where low V_{th} standard cells and high V_{th} sleep transistor cells are coneected. In this figure, low V_{th} standard cells *C1-C5* are clustered to be connected to high V_{th} sleep transistors, *S1-S3*, by $VGND$. We can see that a sleep transistor can be arbitrarily placed at two sides of a cluster from this structure.

Figure 3.3 shows an example of the standard cell placement. All cells are connected to sleep transistors. The cells in the same row are connected to the sleep transistors which are at the end of the same row in our placement. To maintain the circuit performance under a pre-specified performance degradation, multiple unit-size sleep transistors are abutted.

The *row size (width)* includes the the size (width) of low V_{th} standard cells and the size (width) of high V_{th} sleep transistor cells. The *chip width* is

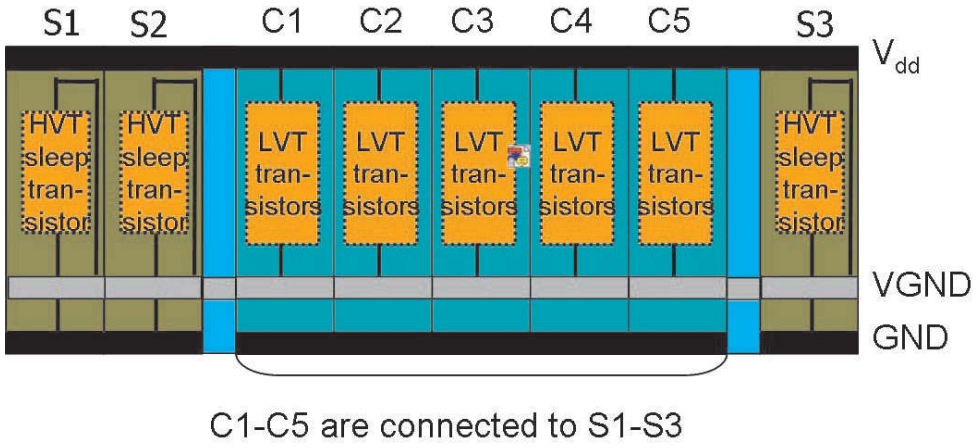


Figure 3.2: Example of the connection between low V_{th} standard cells and high V_{th} sleep transistor

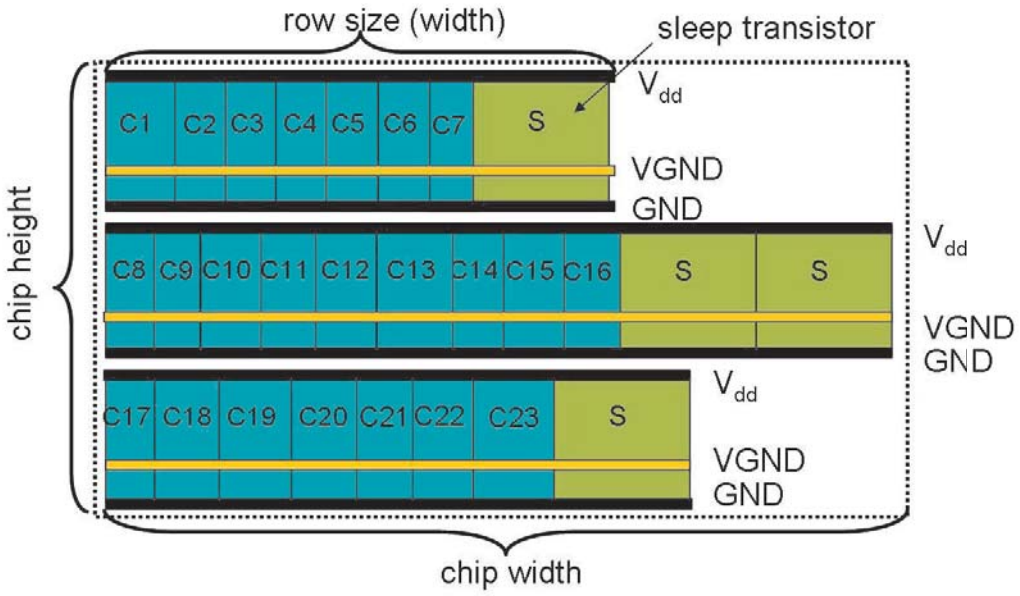


Figure 3.3: Example of placement style

equal to the maximum *row size*, the *chip height* is determined by the number of rows, and the *chip size* is the multiplication of the *chip width* and the *chip height*.

3.1.2 Sleep Transistor Sizing

The sleep transistor sizing problem is to determine the size of sleep transistor and compute the maximum current flowing through the sleep transistor under a specified timing constraint. We will calculate the size of sleep transistor by equations from [1].

When the sleep transistor is absent, the propagation delay (τ_d) for a CMOS gate can be approximated by Equation (3.1),

$$\tau_d = \frac{C_{load} V_{dd}}{(V_{dd} - V_{tL})^{\alpha}} \tag{3.1}$$

where C_{load} is the load capacitance, V_{dd} is the supply voltage, V_{tL} is the threshold voltage in the low V_{th} module, and α is the velocity saturation index for modeling short channel effects. When the sleep transistor is present, the propagation delay (τ_d^{sleep}) for a CMOS gate increases to Equation (3.2),

$$\tau_d^{sleep} = \frac{C_{load} V_{dd}}{(V_{dd} - V_x - V_{tL})^{\alpha}} \tag{3.2}$$

where V_x is the potential of the virtual ground. Assuming the circuit can tolerate a 5% performance degradation with the presence of the sleep transistor,

then

$$\frac{\tau_d}{\tau_d^{sleep}} = 95\% \tag{3.3}$$

According to Equation (3.1), Equation (3.2), and Equation (3.3), and assuming V_{dd}=*1.8V*, V_{tL}=*350mV*, and α=*1* for simplification, V_x can be formulated as in Equation (3.4).

$$V_x = 0.05(V_{dd} - V_{tL}) \tag{3.4}$$

Thus, V_x is computed as *0.0725V*. Then, we can calculate the maximum current flowing through one sleep transistor (denoted as $ST\_maxcurrent$) by SPICE simulation. We use TSMC spice model in *0.18*μm technology, and set high V_{th} of sleep transistor to be *500mV*. The $ST\_maxcurrent$ is computed as *432uA* under $(\frac{W}{L})_{sleep} \approx 35$. This means that if the maximum simultaneous switching (discharging) current ($MSSC$) of the low V_{th} module is less than *432uA* ($ST\_maxcurrent$), the performance of circuit will be maintained.

In the library, let the width of and height of a unit low V_{th} transistor be *0.25U* and *1U*, respectively. Then, from the above simulation, when V_x=*0.0725V*, to maintain the maximum current flowing through one sleep transistor being *432uA*, the width and height of the sleep transistor are set to *0.25U* and *8.7U* by SPICE simulation, respectively, which is nearly ninefold the width of the unit size low V_{th} transistor. Therefore, we assume that the size (width) of a sleep transistor cell is ninefold size (width) of the standard

cell which has only one transistor. Now, assuming the maximum simultaneous switching (discharging) current ($MSSC$) of the cells in row i is known, the number of the sleep transistor cells needed can be calculated as

$$Number_{st} = \lceil (\frac{MSSC(row_i)}{ST_{maxcurrent}}) \rceil \tag{3.5}$$

So, the size of the sleep transistor cells needed is $Number_{st}$ unit-size sleep transistors. In the next paragraph, we will show a method to estimate ($MSSC(row_i)$). Although this method overestimates ($MSSC(row_i)$), it is more simple and efficient compared to the method finding optimal solution of ($MSSC(row_i)$).

As to the computation of the maximum simultaneous switching (discharging) current ($MSSC$) of the cells in row i ($MSSC(row_i)$), it is computed based on the approach proposed in [3], which takes into consideration both topology and functionality. It proceeds as follows. First, we can construct a *relation graph* $G(V, E)$ which represent the discharge relation among cells in row i. A vertex $v_i \in V$ stands for one gate in row i and an edge $(v_i, v_j) \in E$ represents that v_i and v_j do not make transition at the same time. Note that all cells in a clique of the *relation graph* are mutually exclusive discharge. Therefore, only the maximum current among them need to be computed as the discharge current of the clique. Thus, we want to partition our *relation graph* to as fewer cliques as possible. The heuristic algorithm in [15] is uti-

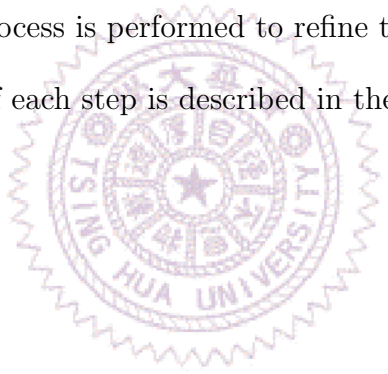
lized to achieve our objective. At each step, each pair of vertices connected by an edge is taken as a candidate, and a candidate with largest cost (the number of common neighbors) will be selected. After a candidate is selected, these two vertices are combined to form a new vertex and the *relation graph* is updated. This algorithm stops when there is no pair of vertices to combine and each vertex stands for one clique finally. Figure 3.4 shows this algorithm, and the more detailed description of this algorithm is shown in [15] . As a result, $MSSC(row_i)$ is the sum of the maximum current among each clique in row i

```
1 Algorithm : Clique Partitioning Algorithm()
2 Input : Relation Graph G = (V, E)
3 Output : Clustered Relation Graph
4
5   While(E ≠ ∅)
6   {
7       find (v_i, v_j) ∈ E with the largest cost ;
8       /* cost = N(CommonNeighbor) */
9       s ← v_i ∪ v_j;
10      V ← V ∪ s;
11      delete edges linked v_i or v_j;
12      add edges connecting between s and neighbors of s;
13  }
```

Figure 3.4: Clique partitioning

3.2 Functionality Directed Placement

In this section, we propose our functionality directed placement method. Our method follows the clustering technique proposed in [3], and incorporates the wire connectivity issue into placement process. The design flow is presented in Figure 3.5. First, we construct a *relation graph* among all cells taking topology and functionality into consideration. Then, we *find exclusive discharge clusters* based on the relation graph. After that, we perform cell placement according to the cluster information. Finally, the *cell moving among clusters* process is performed to refine the placement result. The detailed description of each step is described in the following sections.



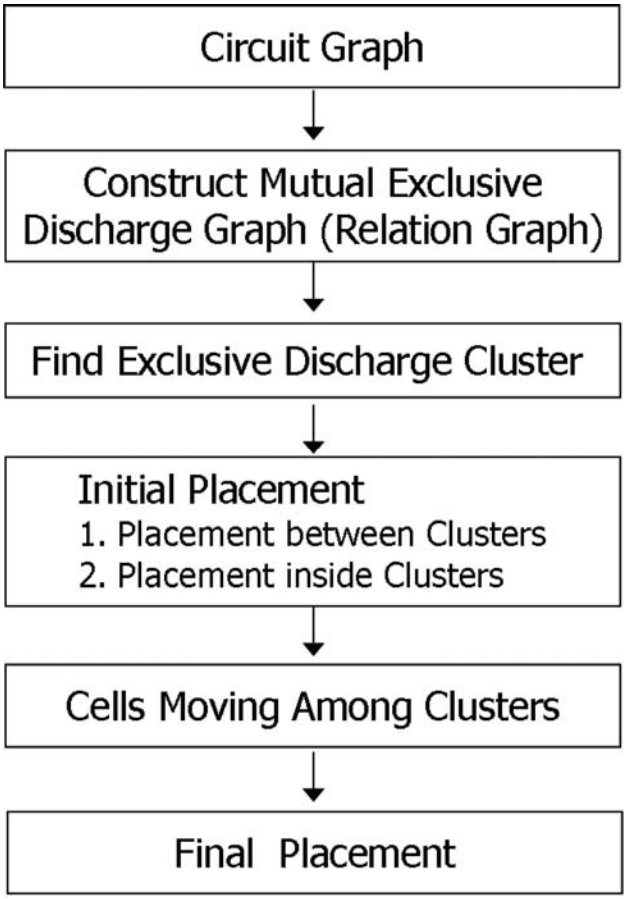


Figure 3.5: Design flow of the functionality directed placement

3.2.1 Find Exclusive Discharge Clusters

Following the clustering technique proposed in [3], we can construct a *relation graph* (*mutual exclusive discharge graph*) among all cells of the entire circuit. Then, we want to find exclusive discharge clusters. The algorithm used here is very similar to the clique partition algorithm in [15]. However, the algorithm of [15] has one major drawback. The algorithm only considers the reduction of cliques. Hence, two cells clustered together may be far away in the placement. This will cause the increase of total wirelength. As a result, we do some modifications to reduce the wirelength overhead. In the following, we show our modifications and explain why it can reduce the wirelength overhead.

The main modification is the *cost function* defined on an edge which is used to choose the clustering candidates. Let $cost(v_i, v_j)$ be the cost on an edge between v_i and v_j. It is defined as

$$cost(v_i, v_j) = \alpha * N(connection) + \beta * N(CommonNeighbor) \qquad (3.6)$$

where α and β are weighting factors, $N(CommonNeighbor)$ is the number of common neighbors of v_i and v_j, and $N(connection)$ is the number of connections between v_i and v_j. We take Figure 3.6 as an example to illustrate the cost function. The graph in Figure 3.6(a) is a relation graph. Each vertex v_i

represents a clique. Each edge (v_i, v_j) represents a mutual exclusive discharge edge. Two vertices (cliques) can be combined together to form a new vertex (clique). We see that both *v1* and *v2* have edges connecting *v3* and *v4*. So, *v3* and *v4* is the common neighbors of *(v1,v2)*, and $N(CommonNeighbor)$ of *(v1,v2)* is equal to 2. In Figure 3.6(b), we show the detailed connections between *v1* and *v2*. Each node c_i is a cell clustered into a vertex (clique). Each dotted line connecting two cells c_i and c_j means that there is a wire connecting c_i and c_j in the circuit topology. Then, $N(connection)$ of *(v1,v2)* is the number of dotted lines between $v1$ and $v2$. In this example, $N(connection)$ of *(v1,v2)* is equal to 3.

The reason that we add this factor, $(N(connection))$, in the cost function is to make the cells clustered in a clique more connective. This will reduce the wirelength overhead since the cells clustered in a clique will be placed together in our placement method proposed in Section 3.2.2.

Moreover, we add a procedure to remove edge when area constraint is not satisfied. $Area(v_i)$ denote the sum of area of the cells clustered in v_i . For an edge (v_i, v_j), if the sum of $Area(v_i)$ and $Area(v_j)$ is greater than area constraint, this edge will be deleted. This procedure limits the area of each clique (cluster) found finally under a computed area constraint. The reason behind this heuristic is because a large cluster causes large wirelength over-

head in our placement method proposed in Section 3.2.2. Figure 3.7 shows our modified clique partitioning algorithm, we call it *Placement Directed Clique Partitioning Algorithm*.

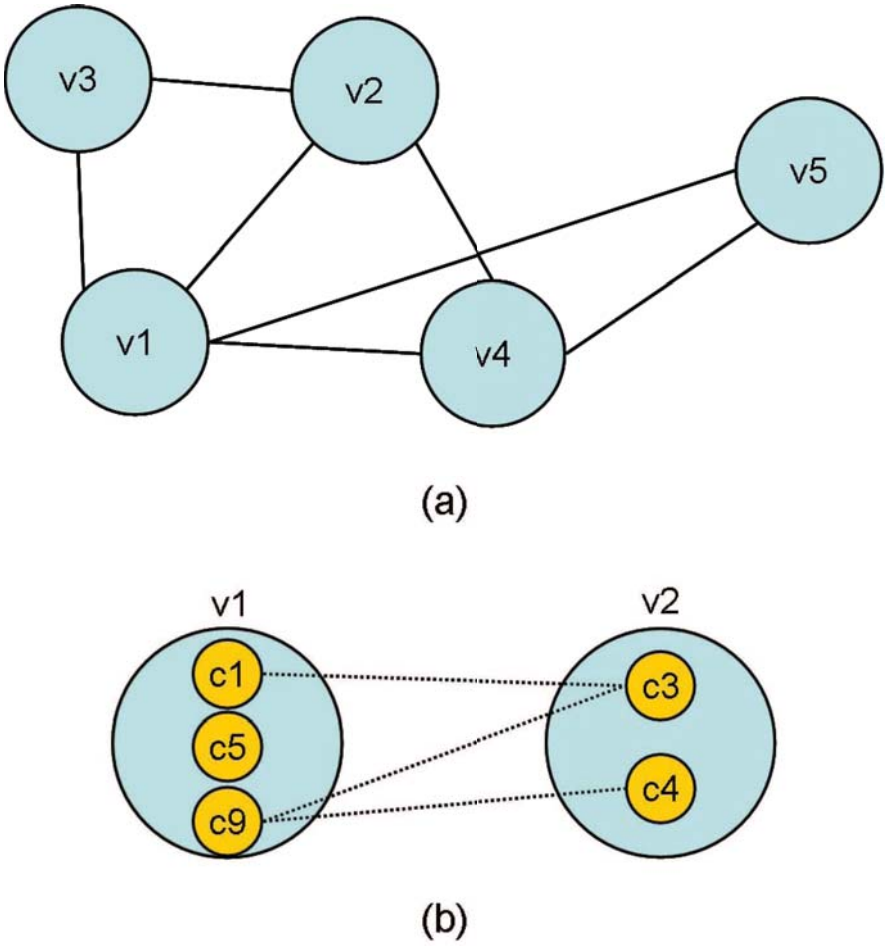


Figure 3.6: Illustration of the cost function in placement directed clique partitioning : (a) Relation graph (b) Detailed connections between *v1* and *v2*

```
1 Algorithm : Placement Directed Clique Partitioning Algorithm()
2 Input : Relation Graph G = (V, E)
3 Output : Clustered Relation Graph
4
5   construct connection edge
6
7   remove edges which are over area constraint
8   While(E ≠ ∅)
9   {
10      find (v_i, v_j) ∈ E with the largest cost;
11      /* cost = α * N(connection) + β * N(CommonNeighbor) */
12      s ← v_i ∪ v_j;
13      V ← V ∪ s;
14      delete edges linked v_i or v_j;
15      add edges connecting between s and neighbors of s;
16      remove edges which are over area constraint
17  }
```

Figure 3.7: Placement directed clique partitioning

3.2.2 Initial Placement

After performing our placement directed clique partitioning, we find many clusters, and mutual exclusive discharge cells in each cluster. Then, this cluster information is used to perform our initial placement. We use a fast, effective standard cell placement tool called "*Dragon*" [10] to assist us with placement of cells. First, we regard each cluster as a *supercell*, and construct the new netlist of supercells. Using the placement tool, we do the global placement to obtain the position of each cluster. Second, for each cluster, we do the local placement of the cells in the cluster. After these two steps, the position of each cell is known, and it is taken as our initial placement. Referring to Section 3.1.2, the size of sleep transistors needed for each row can be calculated.

3.2.3 Cell Moving Among Clusters

To reduce the total wirelength overhead, in this section we propose a *cell moving among clusters* method to refine the initial placement. Our algorithm is shown in Figure 3.8 . Step 1, step 2 and step 5 are in the *outer while loop of cell replacement*. Step 3 and step 4 are in the *inner while loop of iterative cell moving*. In the following, we give a detailed description for each step in the algorithm.

Step 1. We partition the initial placement according the *area constraint* given by user. Successive cells in the same row are view as in a *cluster* if the total area of these cells do not exceed the *area constraint*. Otherwise, a row is partitioned to more than one cluster. The *area constraint* here is used to constrain the size of each cluster. The reason for this *area constraint* is to has a more accurate estimation of wirelength. During the moving process, when a cell x moves to a cluster j, we assume that the center of the cluster j is the new position of cell x. A large cluster causes the inaccurate estimation of wirelength for the actual position of the cell.

Step 2. In this step, we compute the gains of all cells. Let the number of clusters be k. Each cell has *k-1* gains for moving to other *k-1* different clusters. Let $g(c_{x,i->j})$ be the gain of cell x moving to cluster j from cluster i. It is defined as :

$$g(c_{x,i->j}) = \alpha * W(c_{x,i->j}) + \beta * RD(c_{x,i->j}) \tag{3.7}$$

The first term, $W(c_{x,i->j})$, is the gain of wirelength while cell x moves to cluster j from cluster i. It is computed as :

$$W(c_{x,i->j}) = W(c_{x,i}) - W(c_{x,j}) \tag{3.8}$$

where $W(c_{x,i})$ is the total wirelength connecting to cell x before moving, and $W(c_{x,j})$ is the total wirelength connecting to cell x after moving to cluster j. Note that when a cell x moves to a cluster j, we assume that the center of the cluster j as the new position of cell x, and *Manhattan Distance* is used for our estimation of wirelength. The second term, $RD(c_{x,i->j})$, is the the difference of the size of two rows. It is computed as :

$$RD(c_{x,i->j}) = RS(c_{x,i}) - RS(c_{x,j}) \tag{3.9}$$

where $RS(c_{x,i})$ is the size of the original row that x is located in, and $RS(c_{x,j})$ is the size of the new row that x is moved to.

In Equation (3.7), $W(c_{x,i->j})$ is used to reduce the wirelength overhead and $RD(c_{x,i->j})$ is used to reduce the maximum size of all rows, namely the chip size. Because our main objective here is to reduce the wirelength, we give the factor $W(c_{x,i->j})$ a higher weight.

Step 3. Having computed the gains of each cell, we now choose the *base cell*. The base cell, $c_{x,i->j}$, is the one that has a maximum gain greater than zero and does not violate the *moving constraint*. The *moving constraint* is defined to limit the increase of one row. It is set to be the maximum row size of all rows in the placement. For a base cell $c_{x,i->j}$, we assume that cell x moves to cluster j, and cluster j is in row m. We compute the size of row

m. If the size of row m is greater than the maximum size of all rows, this base cell violate the *moving constraint*. If a base cell $c_{x,i->j}$ is chosen, then the cell x moves from cluster i to cluster j. If no base cell is found, then the *inner while loop of iterative cell moving* ends and the procedure goes to step 5.

Step 4. After each move, the selected cell is locked. Then the gains of cells are updated. The procedure continuously runs in the *inner while loop of iterative cell moving* until all cells are locked or the *choosing_number* reaches the *user_constraint_number*. The *choosing_number* is used to count the times that base cells are selected. The *user_constraint_number* given by user is the number of cells that can be moved in the current moving iteration. It is used to limit the number of cells that can be moved, because we take the center of clusters as the new position for the chosen base cell. This assumption causes more and more inaccurate estimation of wirelength when more and more cells are chosen to move to other clusters. So, after a number of base cells are selected, the procedure ends *inner while loop of iterative cell moving*.

Step 5. After the *inner while loop of iterative cell moving*, a number of cells were moved to other clusters. For each modified cluster we perform the local cell placement in a cluster and obtain the new placement information.

We calculate the total wirelength of the circuit and the chip area, and record them. The *outer while loop* continues until T (the user defined parameter) number of times is reached. Then we select the placement which has the minimum total wirelength in the process as our final placement.

Algorithm : Cell Moving among Clusters Algorithm()
Input : Initial Placement Information
Output : New Placement Information

/* Outer while loop of cell replacement */
While(**NOT** Continuous T times that the wirelength can not be improved)
Step 1. Partition initial placement to k clusters according the given area constraint;

Step 2. Compute gains of all cells;
(Each cell has *k-1* gains for moving to *k-1* different clusters.)

/* Inner while loop of iterative cell moving */
While(unlock cells $\neq \emptyset$ and choosing_number < user_constraint_number)
Step 3. Select *base cell* and call it c_{ij};
(i is the cell number, and j is the cluster number moving to.)
If no base cell **Then Goto** step 5;
A base cell is the one which
(1) has maximum gains (gain>0);
(2) satisfies the *moving constraint*;

Step 4. Lock cell c_i;
choosing_number = choosing_number + 1;
Update gains;
End inner while loop

Step 5. Obtain new placement information;
(Replace the cells for the modified clusters.)
Calculate the wirelength and the chip area, and record them;
End outer while loop

Figure 3.8: Cell moving among clusters algorithm

3.3 Direct Placement with Iterative Cell Moving

The *direct placement* means using "Dragon" to perform cell placement without taking functionality into consideration. In Section 3.2, *functionality directed placement*, we use the the information of mutual exclusive discharge clusters to obtain initial placement. Comparing to *direct placement*, it needs less sleep transistor cells. This implies the maximum row size of initial placement using cluster information is less than that of direct placement. Hence, we get smaller chip area by considering cluster information. However, the limit of connecting the cells in the same cluster together causes the significant increase of wirelength overhead. Therefore, we propose to use a direct placement result as the initial placement. Then, we perform the same *cell moving among clusters algorithm* in Section 3.2.3 to fine tune the initial placement.

Because the initial placement from direct placement has better wirelength, yet worse chip size, we will increase the weight of $RD(c_{x,i->j})$ in Equation (3.7) to reduce the chip area. Using this cost function, the *iteratively cell moving among clusters algorithm* will gradually trade total wirelength for area.