In this report, we first give a short introduction about the problem we are going to solve, then we describe our detailed procedure and explain why we use those methods. Afterwards, we discuss our experimental result, and at the end, we conclusively discuss the result and give some recommendations for this problem. And the job division can be seen in the last paragraph.

I. Introduction

The idea of the problem is to optimize the performance on routing, which here is approximated by the total HPWL of all nets, while two dies for placing and hybrid bonding terminals are available. However, in addition to the placing strategy, several constraints need to be satisfied, such as the maximum utility of each die and the minimum spacing between two terminals. In the following discussion, we call cells as instances, and abbreviate hybrid bonding terminals as terminals..

II. Entire Procedure

To deal with the problem, all we have done can be divided into 3 parts: partition, instance placing and terminal placing. In partition, we divide all instances into 2 groups, one to be placed in Die 1, while the other in Die 2. Next, we use simulated annealing to optimize instance placing on Die 1. Following this result, we use heuristic to place all terminals to derive a good result considering the placement on Die 1. Lastly, we again use simulated annealing to derive the placement on Die 2 considering the result of terminal placement. The simulated annealing on Die 2 is almost the same as that of Die1, but considering all terminals as extra pins when calculating HPWL.

i. Partitioning

For partitioning, our goal is to separate all instances into 2 groups, and that the number of terminals should be ranged in an interval we desire. This is because result with too few terminals loses the benefit of die-to-die(D2D) placement (since instances of a net need to be crammed on the same die)(low flexibility for instances), while one with too much terminals faces the terminal spacing- constraint (low flexibility for terminals). The ideal number of terminals(INT) we set is:

$$INT = 0.5 \times \frac{area\ of\ die}{equivalent\ area\ of\ terminal}$$

To achieve this goal, we construct a graph and use the idea of Fiduccia-Mattheyses (FM) algorithm, followings talk about how we construct the graph and why it works:

$$G(V, E):$$

$$V = \{\ collection\ of\ all\ instances\ \}$$

$$E = \{(u, v) \\ :\ exist\ a\ net\ including\ u\ and\ v\}$$

$$cost\ function\ for\ edge\ e, c(e): E \to \Re:$$

$$c(e) = \sum_{net\ k\ that\ e\ included} \frac{4}{(\#\ of\ instances\ on\ net\ k)^2}$$

In heuristic, the net through many instances is very likely to have a terminal, thus the effect of edge should be small whether the edge crosses the dies or not. More closely, in figure (II)(i)(a), we can find that under the cost of edge we set, the maximum external cost contributed by a net is 1. We then use this relation while tuning parameters to model the ideal external cost (IEC) to be:

$$IEC = INT * 0.8$$

Later on, we use the idea of FM partition algorithm, which moves one instance to

another die once at a time. Firstly, we

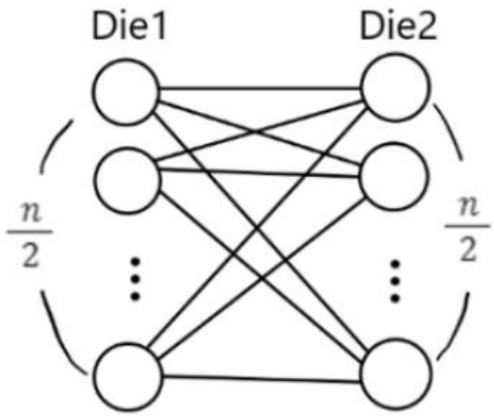


Figure (II)(i)(a)

compute the external cost reduction (external cost - internal cost) of all instances when they are moved. Then, we construct a list which indexed by the cost reduction. This works because the max cost reduction is limited according to the analysis we just go by. We select the closest cost reduction that leads our total cost to IEC, while no area utilization constraints are violated. When an instance is moved, cost reduction of all its neighbors should be added or subtracted by $2 \times$ $(edge\ cost\ between\ the\ two)$. We will repeat this procedure until the total cost has reached IEC or no such instance to move produce a better cost. This procedure ends up with a good performance under our experimental tests.

ii. Instance Placing

When designing the annealing process, we first simplify the placing space on the die by turning it to grids. Because the position in Y-direction has already been constrained to specific row height, we only have to determine the grid size in X-direction. We set the average width of instances on the die as the grid width. We also maintained an array keeping track of the sum of widths of instances in the single row to prevent the situation of total width exceeds the die size constraint. Our neighborhood structure is M1: moving to an empty grid, and M2: swap location with another instance in the same die. And our cost function is $C = C1(hpwl) + C2$ Where $C2 =$ $\{infinite\ if\ \ die\ width\ constraint\ exceed; 0\ otherwise\}$, so that we can reject the movement that would cause the total width exceed the die width. Lastly, our cooling schedule $T_k =$ $T_0 * 0.9995^k$, total iterations =276303, and the process would terminate if it isn't improved for 1000 consecutive iteration.

We originally tried the annealing process with 3 recorded cost, which are "currentCost","previousCost" and "bestCost".And the accept probability $P =$ $\left\{1\ if\ \Delta cost < 0; e^{-\frac{\Delta cost}{temperature}}\ if\ \Delta cost \geq 0\right\}$, where $\Delta cost = currentCost - bestCost$.The recorded bestCost and best layout would be renew only if currentCost-bestCost <0.The best layout would be outputted after whole annealing process. However, we observed once the current layout accepted a layout with higher cost, it is very hard to decrease the cost back to the value lower then bestCost in the process afterward, moreover because the temperature is higher in the beginning, so it is very likely that the process takes such movement in the beginning, hence the cost rarely improves after whole process. Then we figured out two solutions to this problem. The first one is letting the layout return to the best layout if the bestCost isn't renewed after several round of annealing. The second is make the probability for positive $\Delta cost$ became $e^{-\frac{A\Delta cost}{temperature}}$ where A is a large constant, in order to reject most of the cost-increasing movement. After several testing, the second solution is more efficient, the worse performance is owing to the poor probability to reduce the cost after the cost went up, and the O(n) time

complexity returning process. Thus, the second one is our solution for final version.

After all iterations of annealing, out last step is to convert the grid-based placement back to original placing problem. My method is focusing on a single row, move all instances to the right (resp. left) like moving blocks in a slot, to get the rightmost (resp. leftmost) position. After that, we would check if the instances' current positions aren't lies between it's leftmost and rightmost position, and replace them to the right/leftmost if so. The last step of recover is check if overlap exists from the start of a row to the end. For any two overlapped instances, we would move the right one towards right until they aren't overlap anymore, and because we already ensure all instances' right space is enough for other instances in the right, this process won't have die size excess problem.

iii. Terminal Placing

Terminal placing starts right after cell placing on Die 1 is done. Here, we first determine all nets if they need a terminal, then we compute the square of those nets on Die 1, as shown in figure (II)(iii)(a). Afterwards we desire to initialize the terminals then to adjust detaily. Thus, our strategy is to compact all terminals into a grid situated in the middle of the die. The grid will be the smallest grid that affords all terminals to be placed.

We put terminals into the grid in order of the sum of x-position and y-position of the middle point of square of the net on Die1. Each terminal selects the greedy lattice of the current degree(diagonal) if the degree is not filled completely.

We place terminals diagonally because: after we put all terminals into the grid, there will be some spaces not filled in the middle, those spaces produce the flexibility along both x and y direction for movement later.

Lastly, we try to move all terminals by a big step gradually to a small step to reduce the run time while not giving up the better performance we can get. However, during the attempts, checking the spacing constraint is necessary before each movement.

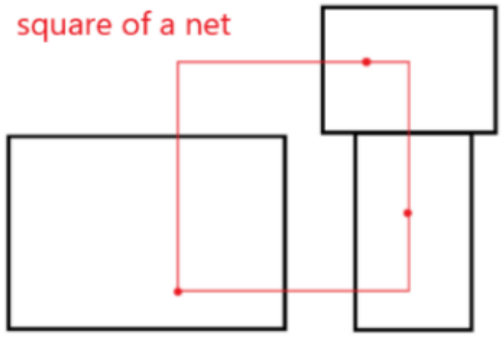


Figure (II)(iii)(a)

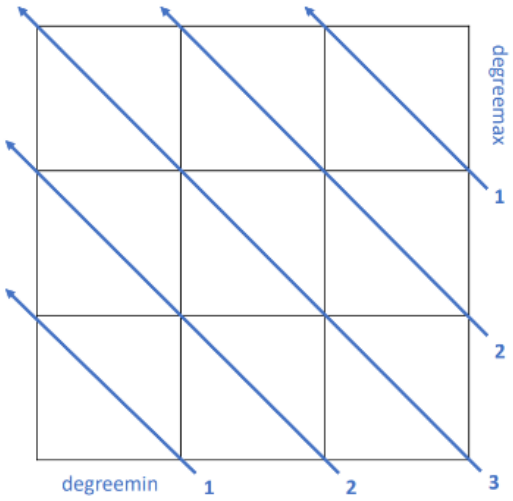


Figure (II)(iii)(b)

III. Experimental Result

Because our partitioning method would have space problem on larger input size, so we only run our program on case1 and case 2

For case 1, here's the output of evaluator , and the plot of placing result is in appendix.
[TopDie] Area(used/max): 670/900 Util: 74.44 Max Util: 80
[BottomDie] Area(used/max): 585/900 Util: 65.00 Max Util: 90
Total HPWL for this design is 148

For case 2, here's the output of evaluator , and the plot of placing result is in appendix.

[TopDie] Area(used/max): 54622304/82936425 Util: 65.86 Max Util: 70
[BottomDie] Area(used/max): 58618728/82936425 Util: 70.68 Max Util: 75
Total HPWL for this design is 12493025

IV. Conclusion and Recommendation for Future Work

i. About Partitioning

During the discussion, in our opinion, partition will be the part which mostly affects the overall performance. I think there are three points that we haven't improved.

First, although the actual number of terminals in test-case-results is quite close to INT, our procedure might accidently fail when the input is not so average, or say if the given input is extremely unbalanced.

Second, the formal implementation of FM partition requires the cost of each edge to be an integer, but the graph we build works badly when we convert those floating points directly to integers. This is a trade of between runtime and memory usage. In the end, we give up the running time while keeping the performance and memory good.

Moreover, our procedure works well on controlling the number of terminals, but we do not consider the performance on instance placing.

Therefore, for future works, we suggest that: (1) the cost function in graph can be redesigned (2) to construct a feedback system from Die 1 placing to partitioning to improve the overall performance.

ii. About Cell Placing

When running on these two test cases, the cost barely decreases when the annealing process is close to the end. So we guess the result isn't far from the best case under such a partition. Our future work on cell placing is to try more annealing prototypes to increase its efficiency. Also adds some runtime sensitive mechanism to output earlier to meet the runtime constraint.

iii. About Terminal Placing

In our original idea, we were not going to let the terminals move toward arbitrary directions, instead, we moved terminals only outward to reduce runtime and to avoid overlaps. However, after implementation, we realize that overlaps are hard to detect efficiently even if we restrict the direction of movement. Thus we consume a lot of runtime in overlap detection and give terminals more flexibility to be moved toward any direction.

Conclusively, we observe that terminal placing is hard to be implemented with a fixed and efficient procedure due to the difficulty of overlap detection. Therefore, we suggest that terminal placing can be implemented by simulated annealing too,

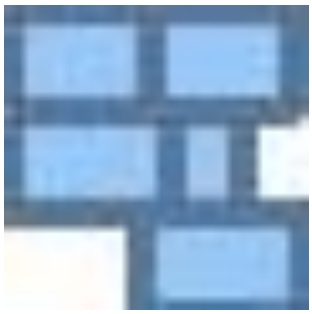
which is easier to control the runtime and maybe leads to a better performance.

V. Job Division

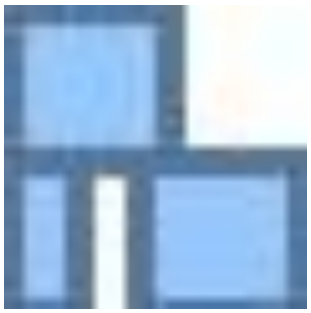
B09901089:terminal placing method designing;instances placing method designing & implementing; objects structure designing; result drawing.

B09901165: partitioning method designing & implementation;objects structure designing terminal placing implementation

VI. Appendix

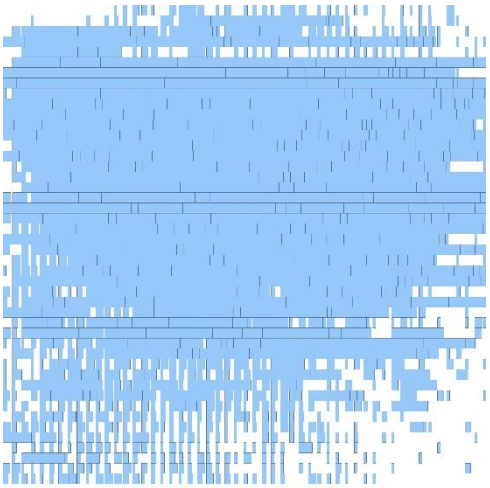


Figure(VI)(i) die1 of case 1

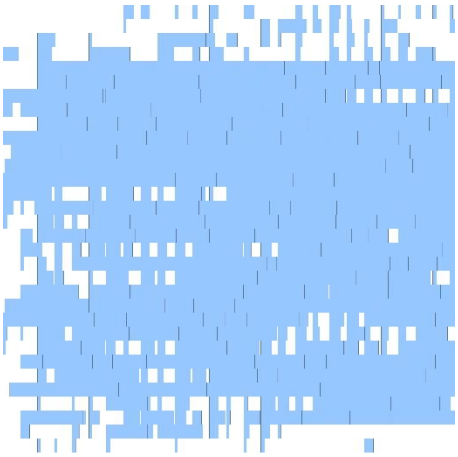


Figure(VI)(ii) die2 of case 1

Figure(VI)(iii) terminal of case 1



Figure(VI)(iv) die1 of case 2



Figure(VI)(v) die2 of case 2

Figure(VI)(vi) terminals of case2