An Incremental Algorithm for Non-Slicing Floorplan Based on Corner Block List Representation *

Yang Liu, Ma Yuchun, Hong Xianlong, Dong Sheqin, and Zhou Qiang

(*Department of Computer Science & Technology, Tsinghua University, Beijing* 100084, *China*)

Abstract: We present a novel incremental algorithm for non-slicing floorplans based on the corner block list representation. The horizontal and vertical adjacency graphs are derived from the packing of the initial floorplanning results. Based on the critical path and the accumulated slack distances we define, we choose the best position for insertion and do a series of operations incrementally, such as deleting modules, adding modules, and resizing modules quickly. This incremental floorplanning algorithm has a very high speed less than 1μm, which is one of the most important measures in this research. The algorithm preserves the original good performances on area and wire length. It can also supply other tools with good physical estimates for area, wire length, and other performance guidelines.

Key words: incremental floorplanning; corner block list; adjacency graph; balance node

EEACC: 2570 **CCACC:** 7410D

CLC number: TN47 **Document code:** A **Article ID:** 0253-4177(2005)12-2335-09

1 Introduction

In VLSI layout design, with the continuous shrinking of feature sizes, deep submicron effects cause more and more interactions between high-level and physical-level design. Recently, several synthesis investigations have been reported taking physical information into account[1~3]. Traditional independence among those design phases, which will cause iterative processes, should be avoided in order to deal with much more complex VLSI systems. To cope with the complexity of the merging of those design phases, incremental algorithms are becoming more and more important. Floorplan design is an important stage in the physical design cycle, which can provide necessary information to estimate quality metrics, such as area, wire length, and net capacitance (which is based on wire length). Those physical metrics are important estimations for high-level SOC synthesis[14], which is becoming more popular for design timing closure in the future. Thus research on the incremental floorplanning algorithms to cope with the interactions between floorplan and high-level synthesis is relevant. The problem with traditional floorplanners is that they are too slow to be invoked every time a potential move in the design space is to be evaluated. To address this problem, we study ways to update the floorplan incrementally since typical design space exploration involves small changes that only affect part of the layout. Our results show that the incremental approach for floorplanning is fast and can supply other tools with good physical estimates for area and wire length.

Floorplanning algorithms are normally based on a simulated annealing (SA) optimization scheme, using various floorplan representations.

* Project supported by the National Natural Science Foundation of China (Nos. 90407005, 60473126) and the Program About 3D Floorplanning and Placement by Intel Corporation

Yang Liu female, was born in 1980, PhD candidate. Her research interests focus on floorplanning and interconnect routing algorithms.

Received 15 May 2005, revised manuscript received 14 September 2005

Representations of the geometric relationships among modules significantly affect the floorplanning design process. In terms of block configuration, two categories of floorplan, slicing[4] and non-slicing, are identified. For general floorplans including both slicing and non-slicing, several encoding schemes have recently been proposed, namely, SP[5], BSG[6], O-tree[7], corner block list (CBL)[8], B*-tree[9], and TCG[10]. Among of them, CBL is an effective representation for non-slicing structures, and it has advantages in both time complexity and solution space over other floorplan representations. Compared with the previous representations, CBL has a smaller upper bound on the number of possible configurations, produces fewer redundancies, and needs only linear computations to generate a corresponding floorplan.

The incremental floorplan is becoming more important for the interaction between floorplanning and synthesis. Cong *et al*.[11] formulated incremental physical design problems and surveyed the existing solutions. Crenshaw *et al*.[12] proposed an incremental floorplanner which used a greedy method to apply local changes on a slicing floorplan tree. The algorithm was applied to a high-level synthesis framework. It used area-only criteria to assess the need for invoking a traditional floorplanner. Liu *et al*.[13] devised an incremental floorplanner based on genetic algorithms. It supported incremental changes on an existing floorplan and could find an acceptable solution ten times faster than traditional approaches. Li *et al*.[14] presented an incremental placement algorithm based on integer programming for reducing congestion.

2 Problem formulations

In the phase of high-level synthesis, there are different binding solutions that lead to changes of floorplanning configurations. The operations in synthesis include the binding moves of share, split, and swap[1].

The "share" operation merges two resources res1 and res2 into one single resource. Similarly, the "split" operation is the reverse of the "share" operation, that is, a single resource is split into two resources. The "swap" operation occurs when some modules are replaced by other modules that may have better performance. The operations on the floorplan corresponding to the three instances described above are "deleting modules", "adding modules", and "resizing modules".

During the design process, the synthesis tool provides an initial full floorplan which is created from a baseline netlist. During the next phase, as new design automation decisions are evaluated, updates are made incrementally on the floorplan. The packing results can be used as an indication of whether the incremental updates were sufficient (if it is a significantly better solution, then a better update threshold should be used).

So our problem can be described as follows: we are given an initial floorplan F using CBL and the modification request of the synthesis. Based on the given binding moves, we modify floorplan F locally into F' while changing the modules' topological relations as little as possible. As a result, we have preserved the good performance derived by the original packing while simultaneously getting the CBL list of the modified floorplan which can be used in the following optimizations. A diagram of our algorithm is shown in Fig. 1.

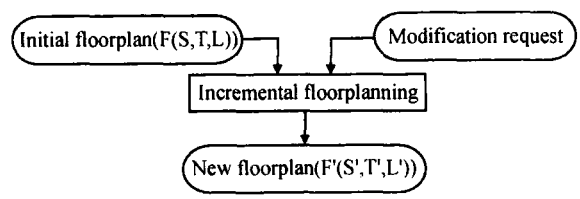

Fig. 1 Diagram of algorithm

We define a set $B = \{B_1, B_2, B_3, \cdots, B_n\}$ representing all rectangular modules from high-level synthesis, where n is the number of modules. Each circuit module B_i is defined by a tuple (w_i, h_i), where w_i and h_i are the width and the height of the module, respectively. The chip area ratio is h/w, where h is the overall height of the chip and w is

the overall width of the chip. Δh and Δw are the increments of overall height and overall width of the chip, respectively. A packing $P = \{(x_i, y_i) \mid 1 \leq i \leq n\}$ is an assignment of coordinates to the lower left corners of the modules such that there are no two rectangular modules overlapping.

The objective of the floorplanner is to minimize the area and wire length of the whole chip and the number of modified blocks.

3 Algorithms

3.1 CBL representation

In our algorithm, we adopt CBL to represent the floorplan which preserves the topological relations of the blocks.

The CBL is derived from a simplified version of a general floorplan called a mosaic structure, which has no empty space, and each block is represented by the rooms with only topological relationships between each other. The CBL represents the topological relations in a mosaic structure by a triple (S, L, T). It divides the chip into rectangular rooms and assigns one and only one block to each room according to (S, L, T).

The corner block (CB) is the block packed at the upper right corner of the floorplan. The vertex of the left and bottom edges of the CB is contained in a T-junction called the corner T-junction, and the CB's orientation is defined by the orientation of the corner T-junction. The T-junction has only two kinds of orientations: a T rotated by 90° (Fig. 2 (a)) and by 180° (Fig. 2 (b)) counterclockwise, respectively. If the T is rotated by 90° counterclockwise, we define the CB to be vertically oriented and denote it by a "0". Otherwise, the CB is horizontally oriented, and we denote it by a "1". The CBL is constructed from the record of a recursive CB deletion. In Fig. 3, the CB d is deleted and the attached T-junctions, whose crossing segments are the non-crossing segment of the corner T-junction, are moved up to the top boundary of the chip. The insertion of a CB is the inverse of the deletion. We use a binary list T_i to record the number of the attached T-junctions of the deleted CB M_i. The number of successive "1"s, which is followed by a terminating "0", corresponds to the number of attached T-junctions.

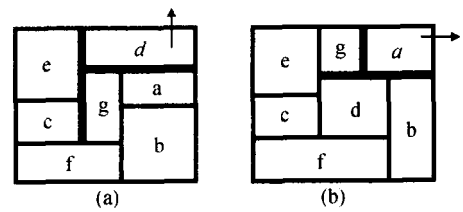

Fig. 2　Orientation of corner block　(a) Vertical CB; (b) Horizontal CB

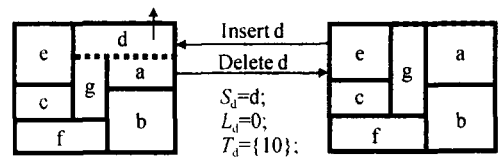

Fig. 3　Corner block d is deleted/inserted

For each block deletion, we keep a record of the block name, CB orientation, and the sequence of T_i. At the end of the deletion iterations, we can obtain three lists: the block name list $\{B_n, B_{n-1}, \cdots, B_1\}$, orientation list $\{L_n, L_{n-1}, \cdots, L_2\}$, and T-junction list $\{T_n, T_{n-1}, \cdots, T_2\}$. We reverse the order of these three lists, respectively. Thus, we have a sequence S of block names, a list L of orientations, and a list of $\{T_2, T_3, \cdots, T_n\}$ which is combined into a binary sequence T. The triple (S, L, T) is a corner block list. The process of inserting corner blocks based on a given (S, L, T) can construct the corresponding floorplan. Figure 4 shows a non-slicing floorplan and its corresponding CBL.

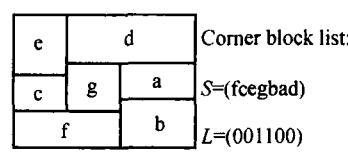

Fig. 4　A non-slicing floorplan and its CBL list

As an effective representation for non-slicing floorplanning, CBL has some advantages: first, it can represent arbitraty non-slicing structures, and the topological relations between blocks; second, it

has low time complexity.

3.2 Strategy of incremental floorplanning

3.2.1 Construct adjacency graphs and some definitions

We build up two directed acyclic graphs which record the topological relations between the blocks in the packing of the floorplan: the horizontal adjacency graph G_h and the vertical adjacency graph G_v (see Figs. 5 and 6).

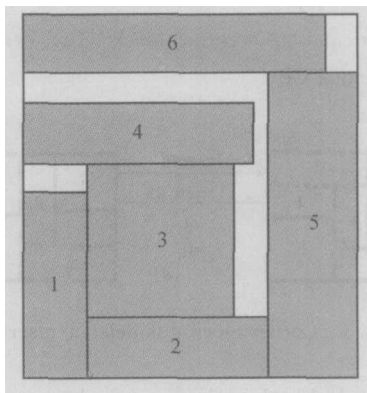

Fig. 5 An example of the packing of the floorplan on CBL

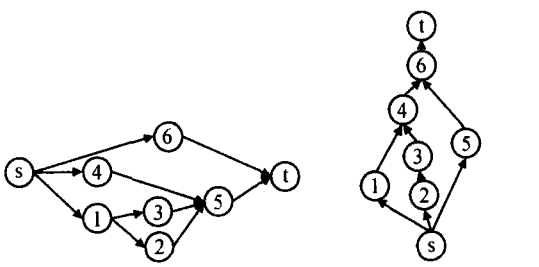

Fig. 6 Adjacency graphs of packing of floorplanning on CBL

We set $G_h = (V_h, E_h)$ and $V_h = \{s_h, v_1, v_2, v_3, \cdots, v_n, t_h\}$, where s_h is the source node in G_h, t_h is the target node in G_h, and v_i represents block B_i which records the information of width, height, room positions, etc. $E_h = E_1 \cup E_2 \cup E_3$, where $E_1 = \{e(i,j) \mid$ "block i and j are adjacent" & "block i is on the left of block j"$\}$, $E_2 = \{e(i,j) \mid$ "an edge from s_h to each of the vertices representing the leftmost blocks."$\}$, $E_3 = \{e(i,j) \mid$ "an edge from each of the vertices representing the rightmost blocks to t_h."$\}$.

$G_v = (V_v, E_v)$ can be defined similarly. $V_v = \{s_v, v_1, v_2, v_3, \cdots, v_n, t_v\}$, where s_v is the source node in G_v, t_v is the target node in G_v, v_i represents block B_i which records the information of width, height, and room position, etc. $E_v = E_1 \cup E_2 \cup E_3$, where $E_1 = \{e(i,j) \mid$ "block i and j are adjacent" & "block i is below block j"$\}$, $E_2 = \{e(i,j) \mid$ "an edge from s_v to each of the vertices representing the lowest blocks."$\}$, $E_3 = \{e(i,j) \mid$ "an edge from each of the vertices representing the uppermost blocks to t_h."$\}$.

In G_v and G_h, $d(i,j)$ denotes weight of $e(i,j)$, that is the slack distance between the corresponding block i and block j.

Based on adjacency graphs, we have:

Definition [horizontal critical path (CP_h)]: For each block B_i, the corresponding node v_i in G_h has: $CP_h(v_i) = \{e(i,j) \mid \forall e(i,j) \in CPh(v_i)$, we must have $d(i,j) = 0\}$.

Definition [vertical critical path (CP_v)]: For each block B_i, the corresponding node v_i in G_v has: $CP_v(v_i) = \{e(i,j) \mid \forall e(i,j)\ CP_v(v_i)$, we must have $d(i,j) = 0\}$.

Definition [critical node (CN)] For each block B_i, the corresponding node v_i in G_h and G_v has a critical node which is the preceding node of connection with v_i in $CP_h(v_i)$ and $CP_v(v_i)$ respectively.

For example in Fig. 6, $CP_h(v_5)$ is the path which consists of v_1, v_2, v_5, and $CN_h(v_5)$ is v_2, not v_3, v_4.

For any node $v_i \in G_h$, we also define the slack of v_i as the accumulated slack distances in all paths from v_i to the target t_h. The slack of v_i in G_h is given by

$$sl_h(i) = \begin{cases} w(i) * \alpha + d(i,k) + \min\{sl_h(k)\}, \\ \qquad \text{if set} \,!= \text{null} \\ w(i) * \alpha + d(i,k), \quad \text{if set} = \text{null} \end{cases} \tag{1}$$

where $k \in$ set, set = {the successors of node i in G_h}; $w(i)$ is the width of block i; and α is given by

$$\alpha = \begin{cases} 0, & \text{if } d(i,j) = 0, j \in \text{set}' \\ 1, & \text{if } d(i,j) \,!= 0, j \in \text{set}' \end{cases} \tag{2}$$

where set′ = {the successors of node i in G_v}.

Similarly, the slack of v_i in G_v is given by

$$sl_v(i) = \begin{cases} w(i) * \alpha + d(i,k) + \min\{sl_v(k)\}, \\ \qquad \text{if set} != \text{null} \\ w(i) * \alpha + d(i,k), \quad \text{if set} = \text{null} \end{cases} \tag{3}$$

where $k \in$ set, set = {the successors of node i in G_v}; $w(i)$ is the height of block i; and α is given by

$$\alpha = \begin{cases} 0, & \text{if } d(i,j) = 0, \quad j \in \text{set}' \\ 1, & \text{if } d(i,j) != 0, \quad j \in \text{set}' \end{cases} \tag{4}$$

where set′ = {the successors of node i in G_h}.

3.2.2 Operation of deleting modules

The operation of deleting modules is relatively easier than adding modules. We need to delete the node corresponding to the module appointed for deletion in both G_h and G_v and then compact the modules corresponding to the influenced succeeding nodes of the deleted node based on the "local max line".

If the deleted node is v_i, whether the succeeding nodes are influenced depends on the critical node belonging to v_i. For example, in Fig. 6, v_2 is the critical node of v_5, whereas v_3 is not the critical node of v_5. Therefore, when we delete block 2, block 5 should be moved close to the rightmost one among all connected adjacency blocks on the left of block 5. But when we delete block 3, block 5 and all of its succeeding connected blocks need not move. The rest may be deduced by analogy.

For facilitating the operation of "moving", we also define the local max line in two dimensions:

Definition [local_max_x]: During the process of the operations ("adding, deleting, and resizing"), there is a local_max_x in the horizontal direction, which is the maximal one's upper-right-point x-coordinate among those blocks on the left of the current module which will be determined to be compacted or not.

Definition [local_max_y]: During the process of the operations ("adding, deleting, and resizing"), there is a local_max_y in the vertical direction, which is the maximal one's upper-right-point y-coordinate among those blocks below the current module which will be determined to be compacted or not.

For example, in Fig. 6, after v_2 is deleted, block 5 should be moved close to block 4, because the current "local_max_x" in the horizontal direction is the x-coordinate of block 4's upper-right-point.

Δw = { The displacement of the rightmost blocks in whole chip. }

The height reduction in G_v can be performed similarly, and Δh is given by

Δh = { The displacement of the uppermost blocks in whole chip. }

Finally, the overall width and height of the chip is updated.

3.2.3 Operation of adding modules

The objective of function of the operation "adding modules" is minimizing the increment of area, namely min(Δarea). The increment of area is given by

$$\Delta\text{area} = (h + \Delta h)(w + \Delta w) - hw \tag{5}$$

Definition [balance node]: The "balance node" is the node BN. If the adding module is inserted above or on the right of BN, the increment of chip area is minimal.

So the question is focused on how the "balance node" is to be found. We assume that the width and height of the inserting module B_p are W_p and H_p, respectively. Therefore, for every node corresponding to the block as a selecting balance node, there is an increment of the whole chip width and height which can be given respectively by

$$\Delta h = \max(0, (H_p - sl_v(i))) \tag{6}$$

$$\Delta w = \max(0, (W_p - sl_h(i))) \tag{7}$$

where i is the node being considered and W_p and H_p are the width and height of the inserting module B_p, respectively.

So the corresponding equation of Δarea change to

$$\Delta\text{area} = [h + \max(0, (H_p - sl_v(i)))] \times [w + \max(0, (W_p - sl_h(i)))] - hw \tag{8}$$

Firstly, we scan all of the original dead spaces. If there is a dead space where the inserting module can be placed close, we need do no other operations

to fing the balance node. All of the original blocks need no movement. But if there is no a dead space, we continue to the next step.

(1) According to the calculation methods of the slack mentioned above, we calculate every block's sl_h and sl_v except for the source node and target node. Based on every block's sl in two directions, we can determine how much the whole chip area changes by the adding a module.

(2) We scan all of the blocks in reverse order of the "S" sequence, to minimize the number of the blocks which need to be moved. For every module of the sequence "S", we have a value of $\Delta\text{area}(i)$, which can be one of the factors by which we determine what node the insert node position can be based on.

The pseudo-code for the operation of "adding module" is shown below.

(1) For every block B_i in reverse order of "S" sequence:

If $(W_p < sl_h(i)) \&(H_p < sl_v(i))$

Select node i as "balance node";

Return True;

Else

Compute $\Delta\text{area}(i)$ and record it;

(2) Sort $\Delta\text{area}(i)$ of all blocks;

(3) Select node i which has the maximal value of $\Delta\text{area}(i)$ as "balance node";

(4) Add the given modules for insertion to the original packing of the floorplan and update the influenced modules' positions;

(5) Update the sequence series (S, T, L);

(6) Update the whole chip height and width and output the area.

In particular, our algorithm can deal with given inserted modules having boundary constraints. For example, we get the information from the high-level synthesis that the insertal module 7 has a high communication priority with blocks 1, 3, and 4, and thus we will select the best insert node position among these blocks. And we only search the minimal Δarea with the bounding limit including the mentioned blocks 1, 3, and 4. So this can minimize the increment of the whole wire length.

3.2.4 Operation of resizing modules

This operation can be divided into two stages: first, the corresponding original module is removed from the packing of the floorplan, that is to say, the original node is deleted in two graphs; second, we select an insertion node position by the method described in Sec. 3.2.2, and then add the swap module into it. This operation can be regarded as a combination of deleting modules and adding modules.

3.2.5 Algorithm time complexity analysis

From the above discussion, we can see that the algorithm time complexity is mainly decided by two processes: One is the process of building constraint graphs, and the other is finding the balance node during the inserting operation.

We reach the conclusion that the length of list T is no more than $2n - 3$ in Ref. [8]. The time complexity of transformation from CBL to floorplan is the same as the length of list T, so it is $O(n)$; and from the algorithm above, we can obviously see that the time complexity of algorithm 1 is the same as the time complexity of the packing process. Therefore, the time complexity of algorithm 1 is $O(n)$.

The process of finding the balance node is based on searching in reverse order of list S in CBL, which has n blocks. For every node, when we calculate Δarea, the minimal $\Delta\text{area}(i)$, and the current balance node, i is logged simultaneously, so it is unnecessary to sort Δarea values for all blocks. Therefore, the time complexity of the inserting operation depends on n, which is the length of S. That is to say, the time complexity is also $O(n)$.

Generally, the whole algorithm time complexity is $O(n)$.

4 Results

We have implemented the incremental floorplanning algorithm in the C programming language, and all experiments are performed on a SUN

sparc20 workstation. Some MCNC benchmarks are used in the experiments. Our experiments do not include soft blocks.

Table 1 shows the floorplan results based on the incremental algorithm and the one-shot algorithm. The "operation" column indicates the operation mentioned in this paper ("Delete 9" means we delete block 9 in the initial floorplan result, and "Add1 (246, 1900)" means we add a new block with a width of 246 and a height of 1900 into the initial floorplan result), which is derived from modification request of high-level synthesis. The "CN" column indicates the number of blocks which have been influenced during the incremental process.

Table 1 Results of initial floorplan and incremental floorplan

Circuit	Instance number	Operation	Area/ mm²			Wirelength/ μm			CN
			F	*F*′	Improved	*F*	*F*′	Improved	
apte	9	Delete 9	49.178208	48.276864	1.83 %	756169	751837	0.57 %	0
apte	9	Add1(246,1900)	49.178208	50.010372	- 1.69 %	756169	758364	- 0.29 %	3
xerox	10	Delete 9	20.638310	17.148284	16.9 %	1058078	871605	17.6 %	7
xerox	10	Add1(46,2000)	20.638310	20.762280	- 0.60 %	1058078	1062908	- 0.45 %	4
Ami33	33	Delete 30	1.296540	1.296540	0.00 %	127530	126452	0.85 %	8
Ami33	33	Add1(70,303)	1.296540	1.367338	- 5.46 %	127530	129205	- 1.31 %	9
Ami33	33	Add1(49,300)	1.296540	1.324666	- 2.17 %	127530	128316	- 0.62 %	1
Ami49	49	Delete 47	41.160000	41.160000	0.00 %	1510590	1504010	0.44 %	8
Ami49	49	Add1(1000,500)	41.160000	41.529600	0.90 %	1510590	1513355	- 0.18 %	1

Due to the $O(n)$ time complexity of our incremental algorithm, the CPU running time of an incremental modification is within microseconds. This advantage will become more obvious with the increase of problem size.

From Table 1 we can see that the area and wirelength of the deletion cases are all improved. The range of change in wirelength of those insertion cases after incremental modification is less than 5.46 %, and that of area is less than only 1.31 %. Our incremental floorplanning algorithm has a very high speed within microseconds, which is one of the most important measures in this research. Besides this, the algorithm preserves the original good performances on area and wire length. So these results show that our algorithm is effective and promising.

We also give an example for the case of inserting blocks. We assume a new block No. 34 with a width of 49 and a height of 300 is added into the initial floorplan result of circuit ami33. Figure 7(a) is the initial floorplan result and Figure 7(b) is the incremental algorithm result, in which the purple blocks are the ones which are influenced and reshuffled in our incremental approach, and the red block is the new inserted one. In this case, the balance node is block 30, above which the added block is inserted. The detailed data of area and wire length in the initial floorplan and incremental algorithm can be seen in Table 1. We can get that the increment area ratio is 2.17 % and the wire length

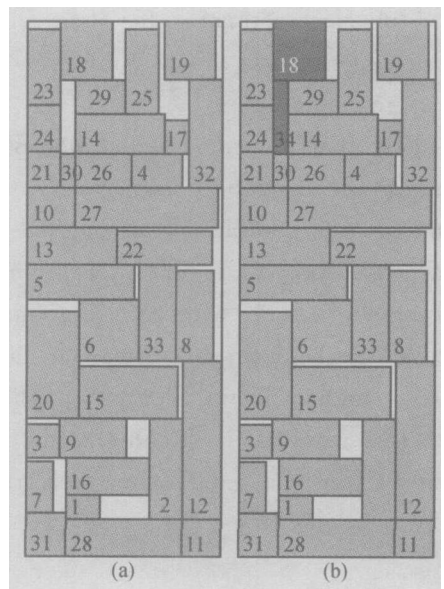

Fig. 7 (a) Initial floorplan result; (b) Incremental algorithm result

ratio is only 0.62 %, and one block has been influenced.

5 Conclusion

In this paper, we present a novel incremental algorithm for a non-slicing floorplan based on corner block list representation. The horizontal and vertical direction adjacency graphs are built up in the packing result of the initial floorplan. Based on the critical path and the accumulated slack distances which we define, we can determine the balance point of insertion and do a series of operations such as deleting modules, adding modules, and resizing modules quickly. Good performance of experimental results in dealing with instances taken from industry proves the effectiveness of our algorithm. We show that the incremental approach to floorplanning can be fast and still supply other tools with good physical estimates for area and wire length. The experimental results show that our algorithm is promising.

We show that our incremental floorplanning method can produce a good floorplan very quickly. In its current implementation, our method can be used as a good estimator for high level synthesis. The experimental results are promising enough to suggest that the next process after our floorplan will generate high quality designs. A possible extension to this work is to extend the incremental floorplanner to optimize interconnect performance.

References

[1] Stammennann A, Helms D, Schulte M. Binding, allocation and floorplanning in low power high-level. International Conference on Computer Aided Design, 2003: 544

[2] Prabhakaran P, Banerjee P, Crenshaw J, et al. Simultaneous scheduling, binding and floorplanning for interconnect power optimization. Proceedings of Twelfth International Conference on VLSI Design, 1999: 423

[3] Prabhakaran P, Banerjee P. Simultaneous scheduling, binding and floorplanning in high-level synthesis. Proceedings of Eleventh International Conference on VLSI Design, 1997: 428

[4] Wong D F, Liu C L. A new algorithm for floorplan design. Proceedings of 23rd ACM/IEEE Design Automation Conference, 1986: 101

[5] Murata H, Fujiyoshi K, Nakatake S, et al. VLSI block placement based on rectangle-packing by the sequence pair. IEEE Trans CAD, 1996, 15(15): 1518

[6] Nakatake S, Murata H, Fujiyoshi K, et al. Block placement on BSG-structure and IC layout application. Proc of International Conference on Computer Aided Design, 1996: 484

[7] Guo P N, Cheng C K. An O-tree representation of non-slicing floorplan and its applications. Proceedings of Design Automation Conference, 1999: 268

[8] Hong Xianlong, Dong sheqin, Huang Gang, et al. Corner block list: an effective and efficient topological representation of non-slicing floorplan. IEEE/ACM International Conference on Computer Aided Design, 2000: 8

[9] Chang Y C, Chang Y W, Wu G M, et al. B*-trees: a new representation for non-slicing floorplans. ACM/IEEE DAC, 2000: 458

[10] Lin J M, Chang Y W. TCG: A transitive closure graph-based representation for non-slicing floorplans. DAC, 2001: 764

[11] Cong J, Sarrafzadeh M. Incremental physical design. Proceedings of International Symposium on Physical Design, 2000: 84

[12] Creshaw J, et al. An incremental floorplanner. Proceeding of IEEE Great Lakes Sym on VLSI, 1999: 248

[13] Liu Yongpan, Yang Huazhong, Luo Rong. An incremental floorplanner based on genetic algorithm. 5th International Conference on ASIC, 2003, 1: 331

[14] Li Zhuoyuan, Wu Weimin, Hong Xianlong. New incremental placement algorithm based on integer programming for reducing congestion. Chinese Journal of Semiconductors, 2004, 25(1): 30

基于角模块布图表示的增量式布图规划算法 *

杨　柳　马昱春　洪先龙　董社勤　周　强

(清华大学计算机科学与技术系, 北京　100084)

摘要: 提出了一种基于 CBL 布图表示的新的增量式布图规划算法. 该算法能很好地解决包括不可二划分结构在内的布图规划问题. 针对现有增量式的一些需求,算法给出了相应的高速解决方案. 在已有的初始布局的基础上,基于 CBL 表示方法建立水平约束和垂直约束图,利用图中关键路径和各模块之间的累加的距离松弛量进行增量式操作. 对于新模块的插入,在力求面积最小,线长最短和移动模块数目最少的目标指引下能快速地找到最佳位置作为插入点,高效地完成相关操作,算法的时间复杂性仅为 $O(n)$. 通过对一组来自工业界的设计实例的测试结果表明,该算法在保证芯片的面积、线长等性能不降低甚至有所改善的情况下,运行速度相当快,仅在 μs 量级,满足了工业界对增量式布图规划算法在速度上的首要要求,同时保证了基本性能的稳定.

关键词: 增量式布图规划; 角模块布图表示; 连接图; 平衡点

EEACC: 2570　　**CCACC**: 7410D

中图分类号: TN47　　**文献标识码**: A　　**文章编号**: 0253-4177(2005)12-2335-09

* 国家自然科学基金(批准号:90407005,60473126)和 Intel 公司 3D 布图规划和布局资助项目

杨　柳　女,1980 年出生,博士研究生,从事布图规划、互连规划布线算法研究.

2005-05-15 收到,2005-09-14 定稿