Placement Constraints in Floorplan Design

Evangeline F.Y. Young, Chris C.N. Chu, and M.L. Ho

***Abstract*— In floorplan design, it is common that a designer will want to control the positions of some modules in the final packing for various purposes like data path alignment and I/O connection. There are several previous works [3], [5], [7], [8], [10], [12]–[14] focusing on some particular kinds of placement constraints. In this paper, we will present a unified method to handle all of them simultaneously, including preplace constraint, range constraint, boundary constraint, alignment, abutment and clustering, etc., in general non-slicing floorplans. We have used incremental updates and an interesting idea of *reduced graph* to improve the runtime of the method. We tested our method using some benchmark data with about one eighth of the modules having placement constraints and the results are very promising. Good packings with all the constraints satisfied can be obtained efficiently.**

***Index Terms*— VLSI CAD, Physical design, Floorplanning, Placement constraints, Optimization**

I. Introduction

Floorplan design is an important step in physical design of VLSI circuits to plan the positions of a set of circuit modules on a chip in order to optimize the circuit performance. In this floorplanning step, it is common that a designer will want to control the positions of some modules in the final packing for various reasons. For example, a designer may want to restrict the separation between two modules if there are many interconnections between them, or he may want to align them vertically in the middle of the chip for bus-based routing. This will also happen in design re-use in which a designer may want to keep the positions of some modules unchanged in the new floorplan. The analog designers will also be interested in a particular kind of placement constraint called symmetry, and some recent literature on this problem can be found from [1], [2]. However, an effective method to control the absolute or relative positions of the modules in floorplanning is non-trivial and this inadequacy has limited the application and usefulness of many floorplanning algorithms in practice.

Several previous works have been done to handle some particular kinds of placement constraints. The floorplanners in [3], [8], [12] can handle preplace constraint in which some modules are fixed in position. The paper [5], [7], [14] work on boundary constraint in which some modules are constrained to be placed along one of the four sides of the chip for I/O connection. The paper [13] generalizes the approach in [12] to handle range constraint in which some modules are restricted to be placed within some rectangular ranges. The floorplanner in [10] can handle alignment constraint which may arise in bus-based routing. Different approaches are used to handle different kinds of constraints and there is no unified method that can handle all of them simultaneously.

In this paper, we will present a unified method that can handle different kinds of placement constraints simultaneously, including preplace constraint, range constraint, boundary constraint, alignment, abutment and clustering, etc., in general non-slicing floorplans. Users can input a mixed set of constraints and our floorplanner will be able to address all of them simultaneously. (It is reasonable to assume that the input constraints are not contradictory to each other. However, we can also handle inconsistent user requirements by generating a packing that satisfies the requirements as much as possible.) We make use of constraint graphs to handle the constraints and can thus be used with any kind of floorplan representation that computes the module positions by constraint graphs, e.g., sequence pair, BSG, O-Tree, CBL, Q-seq, TBS, etc.. In a constraint graph, every module is represented by a vertex and the weighted directed edges represent the minimum displacement between two modules. We can find the x and y positions of a module by computing the longest path from a source to that module in the constraint graphs. In our approach, we modify the constraint graphs to enforce the required constraints in the resultant packing. This is done by augmenting the graphs with positive, negative or zero weighted edges. These augmented edges will restrict the modules to be placed correctly according to the requirements. This technique of adding edges to constraint graphs has been used before for layout compaction [6] and packing of rectilinear blocks [4]. In this paper, we apply and generalize this method to handle different kinds of placement constraints in floorplan design. In addition, we have devised an interesting idea of reduced graph to improve the runtime of the algorithm. A direct implementation of the method is very expensive computationally and is thus impractical. It will take $O(n^3)$ time for each iteration of the annealing process where n is the number of modules. We improved this runtime by reducing the size of the constraint graphs and by updating the constraint graphs incrementally. The time complexity of our algorithm is now $O(kn^2)$ on average for each iteration of the annealing process where k is the number of modules having placement constraints.

We tested our method with some MCNC benchmarks (ami33, ami49 and playout) and a randomly generated data set with 100 modules. Sequence pair representation [9] is used in our implementation. The results are promising and a tight packing with all the constraints satisfied can be obtained efficiently. In the following sections, we will first describe the problem and have a brief review of the sequence pair

The work described in this paper was substantially supported by a grant from the Direct Grant for Research of the Chinese University of Hong Kong (Project No. 2050219).

Evangeline Young is with the Department of Computer Science and Engineering, the Chinese University of Hong Kong, Shatin, N.T., Hong Kong (email: fyyoung@cse.cuhk.edu.hk).

Chris Chu is with the Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011 (email: cnchu@iastate.edu).

M.L. Ho is with ThizLinux Laboratory Limited, Kowloon Bay, Hong Kong (email: manlamho@hotmail.com).

representation and constraint graph. Section 4 will give a detailed explanation of our approach. Section 5 will explain the techniques to reduce the size of the constraint graphs and to update them incrementally. Experimental results will be shown in Section 6.

II. Problem Definition

In floorplanning, we are given the information of a set of modules, including their areas and interconnection and our goal is to plan their positions on a chip to minimize the total chip area and interconnect cost. In this paper, we address this floorplanning problem with placement constraint, i.e., besides the module information, we are also given some constraints in placement between the modules and our goal is to plan their positions on a chip such that all the placement constraints can be satisfied and the area and interconnect cost are minimized.

We consider two general kinds of placement constraints, absolute and relative. For relative placement constraint, users can restrict the horizontal or vertical distance between two modules to a certain value, or to a certain range of values. We use the notation $h(A, B)$ to denote the horizontal displacement from A's lower left corner to B's. Note that this value is positive if B's lower left corner is on the right hand side of A's and is negative otherwise. We use $v(A, B)$ to denote the vertical displacement from A's lower left corner to B's. Similarly, this value is positive if B's lower left corner is above A's and is negative otherwise. Figure 1 illustrates these definitions. A relative placement constraint between two modules A and B can be written as:

$$\begin{aligned} h(A,B) &= [\alpha,\beta] \quad \text{or} \\ v(A,B) &= [\alpha,\beta] \end{aligned}$$

where $\alpha, \beta \in \mathbf{R}$ and $\alpha \leq \beta$. When $\alpha = \beta$, we are restricting the distance between the two modules to a single value and we will write them simply as $h(A, B) = \alpha$ or $v(A, B) = \alpha$ respectively.

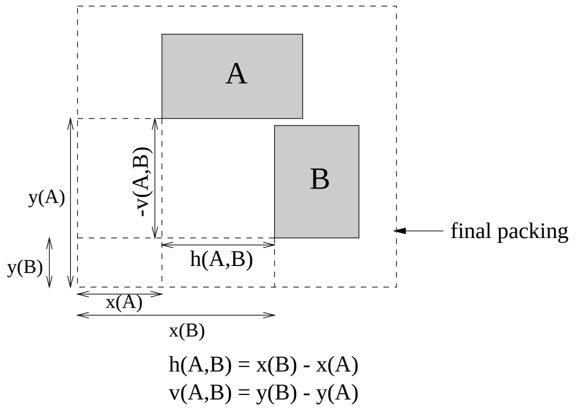


Fig. 1. Notations $h(A, B)$ and $v(A, B)$.

Absolute placement constraint is specified similarly except that one of the two modules in the relationship is a boundary of the chip. We use LL, RR, BB and TT to denote the left, right, bottom and top boundary of the chip respectively.

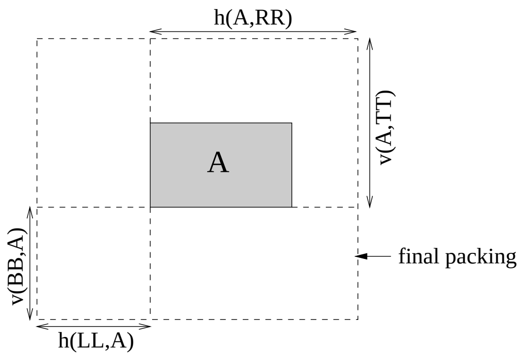


Fig. 2. Notations $h(LL, A)$, $h(A, RR)$, $v(BB, A)$ and $v(A, TT)$.

Therefore notations $h(LL, A)$ and $h(A, RR)$ denote the horizontal distances of the lower left corner of A from the left and the right boundary of the chip respectively. Similarly, we use $v(A, TT)$ and $v(BB, A)$ to denote the vertical distances of the lower left corner of A from the top and the bottom boundary of the chip respectively. Figure 2 illustrates these definitions. An absolute placement constraint of a module A can be written as:

$$\begin{aligned} h(LL,A) &= [\alpha,\beta] \quad \text{or} \\ h(A,RR) &= [\alpha,\beta] \quad \text{or} \\ v(BB,A) &= [\alpha,\beta] \quad \text{or} \\ v(A,TT) &= [\alpha,\beta] \end{aligned}$$

where $\alpha, \beta \in \mathbf{R}$ and $\alpha \leq \beta$. If $\alpha = \beta$, we are restricting the distance between the module and the boundary to a certain value and we will simply write it as $h(LL, A) = \alpha$, $h(A, RR) = \alpha$, $v(BB, A) = \alpha$ or $v(A, TT) = \alpha$ respectively.

These two types of specifications are general enough to express all common types of placement constraints. For example, if we want to restrict the placement of module A, B and C such that they all align horizontally, we can specify the following relative placement constraints:

$$\begin{aligned} v(A,B) &= 0 \\ v(B,C) &= 0 \end{aligned}$$

As another example, if we want to restrict the placement of module A at the lower right corner of the chip, we can specify the following absolute placement constraints:

$$\begin{aligned} h(A,RR) &= w_A \\ v(BB,A) &= 0 \end{aligned}$$

where w_A is the width of A. We can now define our floorplanning problem with placement constraint, **FP/PC**, as follows:

Problem FP/PC: *Given the information of a set of modules including their areas and interconnection, a set Γ_1 of relative placement constraints and a set Γ_2 of absolute placement constraints, the goal is to pack the modules in a rectangular region such that all the given placement constraints are satisfied and the area and interconnect costs are minimized.*

We assume that the input set of placement constraints will not be contradictory to each other, i.e., there exists a feasible

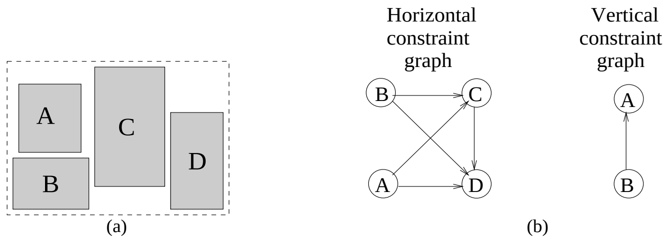


Fig. 3. An example of sequence pair {abcd, bacd}.

packing in which all the constraints can be satisfied simultaneously. However if the input requirements are inherently inconsistent, our floorplanner will still generate a packing that satisfies the requirements as much as possible.

III. Preliminaries

A. Sequence-Pair [9]

We use sequence-pair in our implementation to represent a general non-slicing floorplan. A sequence-pair of a set of modules is a pair of combinations of the module names. For example, $s = (abcd, bacd)$ is a sequence-pair of the module set $\{a, b, c, d\}$. We can derive the relative positions between the modules from a sequence-pair s by the following rules:

- If $s = (\ ..\ a\ ..\ b\ ..,\ ..\ a\ ..\ b\ ..\ )$, then module b is on the right of module a.
- If $s = (\ ..\ a\ ..\ b\ ..,\ ..\ b\ ..\ a\ ..\ )$, then module b is below module a.

Figure 3(a) shows a packing for the sequence pair $\{abcd, bacd\}$.

B. Constraint Graph

We can use a pair of constraint graphs to represent the horizontal and vertical relationships between the module positions imposed by a sequence pair. A horizontal (vertical) constraint graph G_h (G_v) for a set of n modules is a directed graph with n vertices, and the vertices represent the modules and the edges represent the horizontal (vertical) relationships between the module positions. We will have an edge from a to b labeled w_a in G_h where w_a is the width of a if and only if module b is on the right hand side of module a. Similarly, we will have an edge from a to b labeled h_a in G_v where h_a is the height of a if and only if module b is above module a. We can build these graphs directly from a sequence-pair representation s as follows:

- Insert an edge from a to b in G_h labeled w_a if and only if $s = (\ ..\ a\ ..\ b\ ..,\ ..\ a\ ..\ b\ ..\ )$.
- Insert an edge from b to a in G_v labeled h_b if and only if $s = (\ ..\ a\ ..\ b\ ..,\ ..\ b\ ..\ a\ ..\ )$.

Figure 3(b) shows the constraint graphs without edge labels for the sequence pair $\{abcd, bacd\}$.

We can compute the minimum area packing corresponding to a sequence pair efficiently by using the constraint graphs. In a horizontal constraint graph, a weight α on an edge (a, b) means that "b should be at least α units to the right of a". Similarly, in a vertical constraint graph, a weight β on an edge (a, b) means that "b should be at least β units above a". The minimum area packing can thus be obtained by putting the x-coordinate and y-coordinate of a module i as the length of the longest path from a source to i in the horizontal and vertical constraint graph respectively.

IV. Handling Placement Constraints in Constraint Graphs

There are two kinds of placement constraints, relative and absolute. A relative placement constraint describes the relationship between two modules, while an absolute placement constraint describes the relationship between a module and the chip. We will first discuss the approach to handle relative placement constraint and will later discuss how this approach can be used to handle absolute placement constraint by making a simple modification to the constraint graphs.

A. Relative Placement Constraint

In relative placement constraint, users can restrict the horizontal or vertical distance between two modules to a certain range of values. For example, users can specify that $h(A, B) = [\alpha, \beta]$ (or $v(A, B) = [\alpha, \beta]$) where $\alpha, \beta \in \mathbf{R}$ and $\alpha \leq \beta$ meaning that B is at a distance of α to β on the right hand side of A (B is at a distance of α to β above A). When $\alpha = \beta$, we are restricting the distance to a certain value. Notice that both α and β can be zero, positive, negative, $+\infty$ or $-\infty$. (It is trivial to have $\alpha = -\infty$ and $\beta = +\infty$, so we assume that this will not happen.) In order to realize the required constraints in the final packing, we will add a single edge or a pair of edges to the corresponding constraint graph G as described below. We use $w(e)$ to denote the weight of an edge e.

Case 1) If $\alpha = -\infty$, insert an edge $e = (B, A)$ into G with $w(e) = -\beta$.

Case 2) If $\beta = +\infty$, insert an edge $e = (A, B)$ into G with $w(e) = \alpha$.

Case 3) Otherwise, insert two edges $e_1 = (A, B)$ and $e_2 = (B, A)$ into G s.t. $w(e_1) = \alpha$ and $w(e_2) = -\beta$.

The correctness of the above steps follows from Theorem 1 which is proved by making use of Lemma 1 and Lemma 2.

Lemma 1: If there is an edge from A to B labeled s in G_h, $x(B) \in [x(A) + s, +\infty]$.

Proof: According to the definition of horizontal constraint graph, if there is an edge from A to B labeled s in the graph, the lower left corner of B is at a distance of at least s from that of A horizontally to the right, i.e., $x(B) \geq x(A) + s$ which is equivalent to $x(B) \in [x(A) + s, +\infty]$. ■

Lemma 2: The conditions $x(B) \in [x(A) + s, +\infty]$ and $x(A) \in [-\infty, x(B) - s]$ are equivalent.

Proof: The condition $x(B) \in [x(A) + s, +\infty]$ is equivalent to $x(B) \geq x(A) + s$. We can then write $x(B) \geq x(A) + s$ as $x(A) \leq x(B) - s$, which is equivalent to $x(A) \in [-\infty, x(B) - s]$. ■

Theorem 1: The relative placement constraint $h(A, B) = [\alpha, \beta]$ (or $v(A, B) = [\alpha, \beta]$) can be achieved in the final packing by inserting edges into the horizontal (vertical) constraint graph as described in the above cases if the packing is feasible.

Proof: Without loss of generality, we only prove the correctness for the horizontal direction. The proof for the vertical direction follows similarly. To prove the correctness of these steps, we need to show that if the packing is feasible after inserting these edges, the constraint $h(A,B) = [\alpha, \beta]$ will be satisfied in the packing. In the following, G_h denotes the horizontal constraint graph and $x(A)$ denotes the x-coordinate of the lower left corner of module A. Assume that the packing is feasible, i.e., both constraint graphs have no *positive cycles* (a positive cycle in a weighted directed graph is a directed cycle in the graph with positive total weight) and the position of each module can be found by computing the longest path from a source to its corresponding vertex in the two constraint graphs.

Consider the three different cases for the constraint $h(A,B) = [\alpha, \beta]$:

Case 1) $\alpha = -\infty$, i.e., we want $x(B)$ to lie in $[-\infty, x(A)+\beta]$. According to Lemma 2, this condition is equivalent to $x(A) \in [x(B) - \beta, +\infty]$, which, by Lemma 1, can be achieved by inserting an edge from B to A labeled $-\beta$.

Case 2) $\beta = +\infty$, i.e., we want B to lie in $[x(A)+\alpha, +\infty]$. According to Lemma 1, this can be achieved by inserting an edge from A to B labeled α.

Case 3) $-\infty < \alpha \le \beta < +\infty$, i.e., we want B to lie in the range $[x(A) + \alpha, x(A) + \beta]$. Notice that the range $[x(A) + \alpha, x(A) + \beta]$ is equivalent to the range $[x(A) + \alpha, +\infty] \cap [-\infty, x(A) + \beta]$. The first condition can be achieved by inserting an edge from A to B labeled α. The second condition $x(B) \in [-\infty, x(A) + \beta]$ is equivalent to $x(A) \in [x(B) - \beta, +\infty]$ according to Lemma 2 and can be achieved by inserting an edge from B to A labeled $-\beta$. Therefore we need to insert a pair of edges, one from A to B labeled α and the other one from B to A labeled $-\beta$.

■

B. Absolute Placement Constraint

Absolute placement constraint restricts the absolute placement of a module with respect to the whole chip. Users can restrict the placement of a module such that its distance from the boundary of the chip is within a certain range of values. We can handle these kinds of constraints using a method similar to that for relative placement constraints, i.e., by inserting a single edge or a pair of edges to the constraint graphs. To achieve this, we augment the horizontal and vertical constraint graphs each with two extra nodes. For the horizontal constraint graph, we add two nodes: one is a source with zero weighted out-going edges to all the other nodes, and the other one is a sink with zero weighted in-coming edges from all the other nodes. The source represents the left boundary and the sink represents the right boundary of the final packing. Similarly, we add two nodes to the vertical constraint graph: one is a source with zero weighted out-going edges to all the other nodes and one is a sink with zero weighted in-coming edges from all the other nodes. The source represents the bottom boundary and the sink represents the top boundary of the final packing.

In the following, we use v_l and v_r to denote the two additional nodes in the horizontal constraint graph: v_l represents the left boundary and v_r represents the right boundary. Similarly, we use v_t and v_b to denote the two additional nodes in the vertical constraint graph: v_t represents the top boundary and v_b represents the bottom boundary. After adding these nodes, we can handle absolute placement constraint easily as described below. Notice that there is no such cases as $h(A, LL)$, $h(RR, A)$, $v(A, BB)$ or $v(TT, A)$ in the following and α and β are non-negative numbers because we will not consider packing modules outside the boundary of the chip:

- $h(LL, A) \in [\alpha, \beta]$:
  - If $\beta = +\infty$, insert an edge $e_1 = (v_l, A)$ in G_h with $w(e_1) = \alpha$;
  - else, insert edges $e_1 = (v_l, A)$ and $e_2 = (A, v_l)$ in G_h with $w(e_1) = \alpha$ and $w(e_2) = -\beta$.
- $h(A, RR) \in [\alpha, \beta]$:
  - If $\beta = +\infty$, insert an edge $e_1 = (A, v_r)$ in G_h with $w(e_1) = \alpha$;
  - else, insert edges $e_1 = (A, v_r)$ and $e_2 = (v_r, A)$ in G_h with $w(e_1) = \alpha$ and $w(e_2) = -\beta$.
- $v(BB, A) \in [\alpha, \beta]$:
  - If $\beta = +\infty$, insert an edge $e_1 = (v_b, A)$ in G_v with $w(e_1) = \alpha$;
  - else, insert edges $e_1 = (v_b, A)$ and $e_2 = (A, v_b)$ in G_v with $w(e_1) = \alpha$ and $w(e_2) = -\beta$.
- $v(A, TT) \in [\alpha, \beta]$:
  - If $\beta = +\infty$, insert an edge $e_1 = (A, v_t)$ in G_v with $w(e_1) = \alpha$;
  - else, insert edges $e_1 = (A, v_t)$ and $e_2 = (v_t, A)$ in G_v with $w(e_1) = \alpha$ and $w(e_2) = -\beta$.

The proof of correctness of these steps for absolute placement constraint follows directly from that for relative placement constraint and we will not repeat it here.

C. Examples of some Commonly Used Placement Constraint

Using the above specifications for absolute and relative placement constraint, we can describe many different kinds of placement constraints. In this section, we will pick a few commonly used ones and show how each can be specified using a combination of the relative and absolute placement constraints. In the following, we use $x(A)$ and $y(A)$ to denote the x and y coordinates of the lower left corner of module A respectively and we use h_A and w_A to denote the height and width of A respectively.

1) Alignment: To align module A, B, C and D horizontally (Figure 4), we can impose the following constraints:

$$\begin{aligned} v(A,B) &= 0 \\ v(B,C) &= 0 \\ v(C,D) &= 0 \end{aligned}$$

We restrict the vertical distances between these modules to be zero, they will thus all align horizontally. Six additional edges will be inserted into the vertical constraint graph.

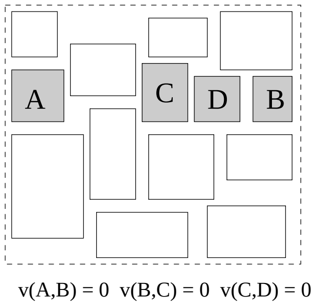


Fig. 4. Examples of alignment constraint.

2) Abutment: To abut module A, B and C horizontally (Figure 5), we can impose the following constraints:

$$\begin{aligned} v(A,B) &= 0 \\ v(B,C) &= 0 \\ h(A,B) &= w_A \\ h(B,C) &= w_B \end{aligned}$$

where w_A and w_B are the widths of module A and B respectively. In this formulation, the vertical distances between these modules are zero, so they will align horizontally. On the other hand, B is restricted to be on the right hand side of A by w_A units and C on the right hand side of B by w_B units, so they will be abutting with each other horizontally. Four additional edges will be inserted into each constraint graph.

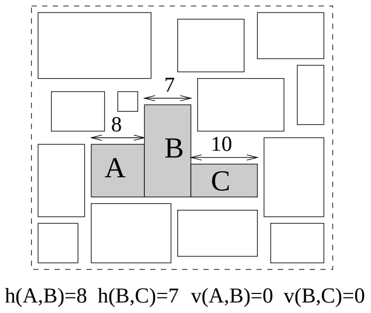


Fig. 5. Examples of abutment constraint.

3) Preplace Constraint: To preplace module A with its lower left corner at (p,q) (Figure 6), we can impose the following constraints:

$$\begin{aligned} h(LL,A) &= p \\ v(BB,A) &= q \end{aligned}$$

We restrict $x(A)$ to be p units from the left boundary and $y(A)$ to be q units from the bottom boundary, so A will be preplaced with its lower left corner at (p,q) in the final packing. Two additional edges will be inserted into each constraint graph.

4) Range Constraint: To restrict the position of A to within the range $\{(x,y)|x_1 \le x \le x_2, y_1 \le y \le y_2\}$ (Figure 7), we can impose the following constraints:

$$\begin{aligned} h(LL,A) &= [x_1,x_2] \\ v(BB,A) &= [y_1,y_2] \end{aligned}$$

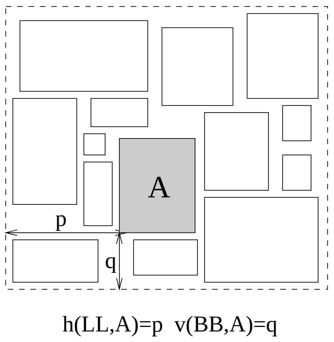


Fig. 6. Examples of preplace constraint.

In this formulation, we restrict $x(A)$ to be x_1 to x_2 units from the left boundary and $y(A)$ to be y_1 to y_2 units from the bottom boundary, so A will lie in the required rectangular region $\{(x,y)|x_1 \le x \le x_2, y_1 \le y \le y_2\}$. Two additional edges will be inserted into each constraint graph.

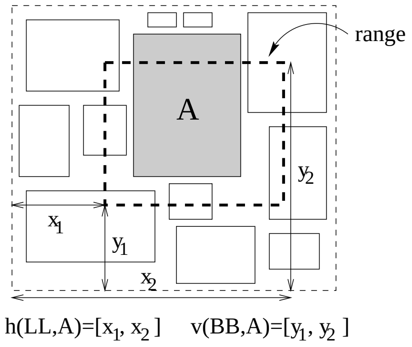


Fig. 7. Examples of range constraint.

5) Boundary Constraint: To place module A at the upper right corner of the final packing, and place B along the top boundary (Figure 8), we can impose the following constraints:

$$\begin{aligned} h(A,RR) &= w_A \\ v(A,TT) &= h_A \\ v(B,TT) &= h_B \end{aligned}$$

In this formulation, we restrict the horizontal distance between A and the right boundary to be the width of A and the vertical distance between A and the top boundary to be the height of A, so module A will be placed at the upper right corner in the final packing. Besides, B is restricted to be h_B units from the top boundary, so B will abut with the top boundary as required. We need to insert two edges into the horizontal constraint graph and four edges into the vertical constraint graph.

6) Clustering: To cluster module A, B and C around D at a distance of at most p units away vertically or horizontally (Figure 9), we can impose the following constraints:

$$\begin{aligned} h(D,A) &= [-p,+p] \\ h(D,B) &= [-p,+p] \end{aligned}$$

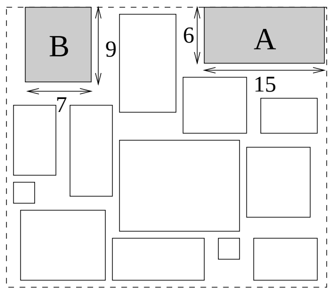


Fig. 8. Examples of boundary constraint.

$$
\begin{aligned}
h(D,C) &= [-p,+p] \\
v(D,A) &= [-p,+p] \\
v(D,B) &= [-p,+p] \\
v(D,C) &= [-p,+p]
\end{aligned}
$$

In this formulation, we restrict the horizontal and vertical distances of A, B and C from D to be at most p units in both directions, so they will cluster around D at a distance of at most p units away. Six additional edges will be inserted into each constraint graph.

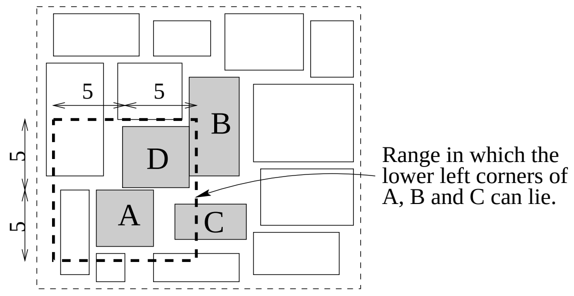

5
5
B
5
D
5
A
C
Range in which the lower left corners of A, B and C can lie.
h(D,A)=[-5,5] h(D,B)=[-5,5] h(D,C)=[-5,5]
v(D,A)=[-5,5] v(D,B)=[-5,5] v(D,C)=[-5,5]

Fig. 9. Examples of clustering constraint.

7) General Placement Constraint: We can use combinations of the above relative and absolute placement constraints to specify different kinds of mixed constraints in general. For example, to restrict the placement such that module B and C align with each other horizontally and they cluster around A at a distance of at most 20 units away (Figure 10), we can impose the following constraints:

$$
\begin{aligned}
v(B,C) &= 0 \\
v(A,B) &= [-20,+20] \\
v(A,C) &= [-20,+20] \\
h(A,B) &= [-20,+20] \\
h(A,C) &= [-20,+20]
\end{aligned}
$$

The first constraint aligns B and C horizontally and the next four cluster B and C around A to within a distance of 20 units away. We need to add four additional edges to the horizontal constraint graph and six to the vertical constraint graph.

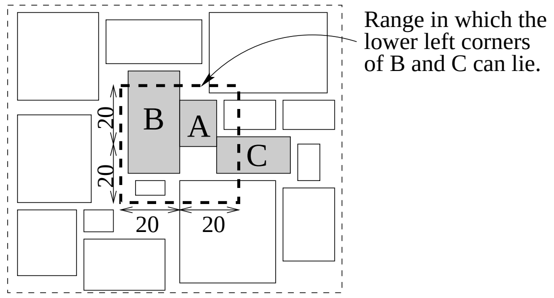

Range in which the lower left corners of B and C can lie.
20
B
A
20
C
20
20
v(B,C) = 0
v(A,B) = [-20,20] v(A,C) = [-20,20]
h(A,B) = [-20,20] h(A,C) = [-20,20]

Fig. 10. An example of an arbitrarily set of mixed constraints.

V. Algorithm and Implementation

We use simulated annealing with sequence pair representation. In each step of the annealing process, we will generate a new packing and compute its area and interconnect cost. We use the vertical and horizontal constraint graphs to compute the position of each module. In order to satisfy the given placement constraints, we will augment the graphs with edges as described in the above section. We call these edges *constraining edges*. If the packing is feasible after adding these edges, i.e., no positive cycle exists in the constraint graphs, we will compute the position of each module as usual and all the constrained modules will be placed at the correct positions. However it is possible that some constraints cannot be satisfied after adding those constraining edges, the packing is then infeasible (note that a packing can be infeasible because the input set of placement constraints are inherently contradictory to each other or the relative positions implied by the sequence pair are contradictory to the input set of constraints). Feasibility of a packing can be checked by detecting positive cycles in the constraint graphs. If a packing is infeasible, we will pack the modules as if there is no placement constraint and compute a penalty term in the cost function to penalize the violations. This strategy ensures that all feasible solutions are reachable, and can drive the packing solution to one that satisfies the constraints as much as possible in case the user requirements are inherently inconsistent. We observed a stable convergence in the annealing process using this scheme and all the placement constraints can be satisfied at the end of the annealing process in all our experiments. We will describe the algorithm in details in the following subsections.

A. Detecting Positive Cycles by Reduced Graphs

After augmenting those constraint graphs with constraining edges, we need to test their feasibility by detecting positive cycles in them. A direct implementation of some classical algorithm (e.g., the modified Floyd-Warshall algorithm [11]) to check positive cycles will take $O(n^3)$ time where n is the total number of modules. In order to improve the runtime, we will reduce the size of the constraint graphs before checking for cycles. This is possible because of the following lemma and theorem. We use $G_h(V,E_h)$ and $G_v(V,E_v)$ to denote the original horizontal and vertical constraint graphs obtained from a sequence pair respectively. $G'_h(V, E'_h)$ and $G'_v(V, E'_v)$ are obtained from G_h and G_v respectively by adding the constraining edges.

Lemma 3: Any cycle in G'_h (G'_v) must contain some edges in $E'_h - E_h$ ($E'_v - E_v$).

Proof: The original constraint graphs G_h and G_v obtained from a sequence pair representation must be acyclic. Therefore, any cycle in G'_h and G'_v must contain at least one constraining edge. ■

From this lemma, we can infer that any cycle in G'_h (G'_v) must contain at least two modules which have placement constraints. Therefore, instead of detecting positive cycles in G'_h and G'_v, we will construct two reduced graphs $H_h(V^*, E^*_h)$ and $H_v(V^*, E^*_v)$ from G_h and G_v respectively where V^* is the set of all modules with placement constraints, E^*_h is the set of all edges $\{e(i,j) | i,j \in V^* \wedge \exists \text{ a path from } i \text{ to } j \text{ in } G_h\}$ and E^*_v is the set of all edges $\{e(i,j) | i,j \in V^* \wedge \exists \text{ a path from } i \text{ to } j \text{ in } G_v\}$. For any edge $e(i,j) \in E^*_h$, $w(e) = d_{G_h}(i,j)$ where $d_{G_h}(i,j)$ denotes the longest path from i to j in G_h, and, similarly, for any edge $e(i,j) \in E^*_v$, $w(e) = d_{G_v}(i,j)$ where $d_{G_v}(i,j)$ denotes the longest path from i to j in G_v. The constraining edges will be inserted into H_h and H_v to give H'_h and H'_v respectively. We will then check for positive cycles in H'_h and H'_v and this is equivalent to checking cycles in G'_h and G'_v according to the following theorem.

Theorem 2: A positive cycle exists in H'_h (H'_v) if and only if a positive cycle exists in G'_h (G'_v).

Proof: Without loss of generality, we will only prove for the horizontal constraint graph. The proof for the vertical constraint graph follows similarly.

"If" condition: If there exists a positive cycle C in H'_h, the edges in C must either be a constraining edge or an edge in H_h. However every edge $e(i,j)$ in H_h actually corresponds to a sequence of edges in G_h (the longest path from i to j in G_h). Since G'_h is obtained from G_h by inserting the constraining edges, a positive cycle must also exist in G'_h.

"Only if" condition: If there exists a positive cycle $C = v_1 v_2 \dots v_k v_1$ in G'_h where the $v'_i s$ for $i = 1 \dots k$ are vertices in G'_h, at least two of these vertices must correspond to modules with placement constraint according to Lemma 3. Denote these vertices corresponding to modules with placement constraint by $v_{\pi(1)}, v_{\pi(2)} \dots v_{\pi(j)}$ where $2 \leq j \leq k$ and $\pi(i) < \pi(i+1)$ for all $i = 1, 2 \dots j-1$. Note that $v_{\pi(1)}, v_{\pi(2)} \dots v_{\pi(j)}$ also exist in H_h because they correspond to modules with placement constraints. Consider the path p_i along C from $v_{\pi(i)}$ to $v_{\pi(i+1)}$ where $1 \leq i < j$. This path will either be a single constraining edge or a sequence of edges from G_h. If p_i is a single constraining edge, this edge will also exist in H'_h since H'_h is obtained from H_h by inserting the constraining edges. If p_i is a sequence of edges from G_h, there must also be an edge $e(v_{\pi(i)}, v_{\pi(i+1)})$ in H_h such that $w(e)$ is not less than the total weight of the path p_i. It is because the weight of the edge $e(v_{\pi(i)}, v_{\pi(i+1)})$ in H_h is computed as the longest path from $v_{\pi(i)}$ to $v_{\pi(i+1)}$ in G_h and it must be at least as long as the path p_i which runs from $v_{\pi(i)}$ to $v_{\pi(i+1)}$ along the cycle C in G_h. This edge $e(v_{\pi(i)}, v_{\pi(i+1)})$ will also exist in H'_h since H'_h is obtained from H_h by inserting the constraining edges. Therefore, every path p_i from $v_{\pi(i)}$ to $v_{\pi(i+1)}$ on the cycle C in G'_h where $i = 1, 2 \dots j-1$ will correspond to an edge from $v_{\pi(i)}$ to $v_{\pi(i+1)}$ in H'_h and the weight of the edge is not less than the total weight of the path p_i. We can conclude that a positive cycle must also exist in H'_h. ■

Constructing H_h and H_v takes $O(kn^2 + c)$ time where c is the total number of constraining edges and k is number of modules with placement constraints. Notice that the number of constraining edges c is usually much smaller than kn^2, so the construction time for H_h and H_v is $O(kn^2)$. The construction can be done by performing a single-source-longest-path algorithm in G_h and G_v once for each v where $v \in V^*$. Checking cycles in H'_h and H'_v by the modified Floyd-Warshall algorithm [11] takes $O(k(k^2+c))$ time because k^2+c is an upper bound on the number of edges in H'_h and H'_v. This time complexity can be further reduced in practice by performing incremental updates as described in the following.

B. Moves and Incremental Updates

In every iteration of the annealing process, we will modify the sequence pair by one of the following three kinds of moves:

[M1] Change the width and height of a module.
[M2] Exchange two modules in both sequences.
[M3] Exchange two modules in the first sequence.

The constraint graphs will not change much after each move, so we do not need to reconstruct them once in every iteration. We can take advantage of this incremental updates in two different places: the construction of G_h and G_v, and the construction of H_h and H_v.

1) Incremental Updates of G_h and G_v: In move M1, a module A is picked and changed in its width and height, so the structures of the constraint graphs will remain the same except that all the out-going edges from A will have their weights changed. In our implementation, the weights on the edges are stored at the source vertices because all the edges out-going from the same vertex will have the same weight. Therefore, we only need to update the weight of vertex A in both G_h and G_v after M1 and this will take constant time. In move M2, two modules A and B are picked and switched in position in both sequences. The structure of the constraint graphs will again remain the same except that the vertices corresponding to A and B will be switched in position. This will affect the weights of the out-going edges from these two vertices. Therefore we only need to update the weights in these two vertices in both G_h and G_v and this will again take constant time. In move M3, two modules A and B are picked and switched in position in the first sequence. The structure of the constraint graphs will change after this move. However, only those modules lying between A and B in the first sequence will be affected and there are $\frac{n}{3}$ of them on average. Besides, each update can be done very efficiently (either an edge $e(i,j)$ in G_v is deleted and a new edge $e(i,j)$ is inserted into G_h, or an edge $e(i,j)$ in G_h is deleted and a new edge $e(i,j)$ is inserted into G_v. Therefore, G_h and G_v can be updated very efficiently in $O(n)$ time.

2) Incremental Updates of H_h and H_v: H_h and H_v are obtained from G_h and G_v by keeping only those vertices with placement constraints. The weight of an edge $e(i, j)$ in H_h (H_v) is the longest path from i to j in G_h (G_v). After move M1, M2 or M3 of the annealing process, the edge weights in H_h and H_v may change because the longest path between two vertices in G_h and G_v will have changed. Fortunately this will only affect a fraction of the edges in H_h and H_v.

In move M1, a module A is selected and changed in width and height. The weight of an edge $e(i, j)$ in H_h or H_v will be affected if i can reach A in the constraint graphs G_h or G_v. This happens if i is lying before A in the second sequence and there are $\frac{k-1}{2}$ of them on average. We need to perform once the single-source-longest-path algorithm in G_h or G_v for each of them and update the weights of all the edges $e(i, j)$ in H_h or H_v for all $j \in V^*$. In M2 and M3, two modules A and B are selected and switched in position in the sequence pair. Similarly, an edge $e(i, j)$ in H_h or H_v will be affected if i can reach A or B in G_h or G_v before or after the move. This happens if i is lying before A or B in the second sequence and there are about $\frac{k-2}{2}$ of them on the average. Similarly, we need to perform once the single-source-longest-path algorithm for each of these affected modules and update the weights of the corresponding edges in H_h and H_v. Therefore updating H_h and H_v takes $O(kn^2)$ time on average.

C. Time Complexity

In each iteration of the annealing process, we modify the sequence pair by performing move M1, M2 or M3. After the move, we need to update G_h, G_v, H_h and H_v. Updating G_h and G_v takes $O(n)$ as explained above. Updating H_h and H_v takes $O(kn^2)$ time on average. After updating these graphs, we need to check for positive cycles in H'_h and H'_v which are obtained from H_h and H_v respectively by inserting the constraining edges. The cycle checking step takes $O(k(k^2+c))$ time. Therefore the total time taken per iteration is $O(n + kn^2 + k(k^2 + c))$ on average, i.e., $O(kn^2)$, since the number of constraining edges c is usually much smaller than n^2.

D. Annealing Schedule and Cost Function

The temperature schedule of the annealing process is of the form $T(k) = rT(k-1)$ for all $k \geq 1$. At each temperature step, enough number of moves are attempted until the total number of moves exceeds a certain number N where N is a user defined constant. The temperature is initialized to a large value at the beginning and the annealing process terminates when the temperature is low enough. The best solution found will then be used to go through a "final baking" process in which only better solutions will be accepted.

The cost function is defined as $A + \lambda W + \gamma P$ where A is the total area of the packing. In our current implementation, W is the half perimeter estimation of the interconnect cost but this term can be replaced by other more sophisticated interconnect cost estimations. P is a penalty term which is zero when all the placement constraints are satisfied, and is otherwise the sum of the squares of each violation, where a violation is measured by the distance the constrained module is from its desired position. This penalty term will be discussed in more details in the following paragraph.

E. Handling Infeasible Packings

If a packing is infeasible, i.e., positive cycles exist in the constraint graphs, we will pack the modules as if there is no constraint and compute a penalty term P. For example, if an edge $e = (A, B)$ labeled α is inserted into the horizontal constraint graph because of a given placement constraint, the penalty term due to this edge in case of an infeasible packing will be $(\min\{x(B) - x(A) - \alpha, 0\})^2$. This gives a good estimation of how far the modules are from their desired positions. Notice that we need to accept infeasible intermediate solutions in the annealing process because it may happen in some cases that a good feasible solution can only be reached from an initial starting point with some infeasible intermediate solutions in between during the searching process. (If the input set of placement constraints are inherently contradictory to each other, there will always be positive cycles in the constraint graphs. The floorplanner will then pack the modules as if there is no constraints and the penalty term will drive the result towards one that satisfies the requirements as much as possible.) The convergence of the annealing process is very stable using this scheme and all the placement constraints can be satisfied at the end of the process in all our experiments.

VI. Experimental Results

We tested our floorplanner on a set of MCNC benchmark data (ami33, ami49 and playout) and a randomly generated data set with 100 modules[1]. Ami33, ami49 and playout were chosen because they are the largest (with 33, 49 and 62 modules respectively) among all the MCNC benchmarks. For each experiment, the temperature is set to 1.5×10^6 initially and is lowered at a constant rate of 0.95 to 0.98 until it is below 1×10^{-10}. The number of iterations at one temperature step is 80. λ in the cost function is set such that the costs of the wirelength and total area are approximately equal. γ is set at a high value (30 to 40) to ensure that all the placement constraints can be satisfied at the end. All the experiments were carried out on a 400 MHz Sun Ultra IIi.

We tested our floorplanner using the benchmark data and a randomly generated data set (random100) by imposing different combinations of placement constraints to the modules. The results are shown in Table I. The result reported in each row is an average obtained by running the experiment six times using three different sets of placement constraints. Notice that the number of constraints refers to the number of constraining edges in the graphs. We can see from the table that the algorithm is very efficient. The percentage deadspace ranges from 5.9% to 8.4% and all the placement constraints can be satisfied in all the experiments. Besides, we can see that the changes in deadspace area and total wirelength in order to handle the placement constraints are very small. Figure 11, 12, 13, 14 and 15 show five resultant packings

[1]The data sets are available in `http://www.cse.cuhk.edu.hk/~fyyoung/data`.

for ami33, ami49 and playout. (Notice that the origin $(0, 0)$ is at the upper right corner in all these packings.)

We have also compared our results with [13] that focuses on handling range constraint in slicing floorplan. We repeated the same experiments on range constraint using our new unified method and the results are shown in Table II. The result reported in each row is an average obtained by running the experiment five times using the benchmark data, ami33, ami49 and playout. The scaled runtimes in the fourth column were obtained by dividing the original runtimes from [13] by a factor of 2.46, the ratio between the speeds of floating point computation of the two machines used. We can see that the performance of the two methods are very similar in both runtime and deadspace. The floorplanner in [13] was a little bit faster because it considered slicing floorplans only. Our floorplanner could give smaller deadspace although the floorplanner in [13] have actually allowed the modules to be very flexible in shape (with aspect ratio in the range of $[0.25, 4.0]$), while we considered a discrete number of shapes for each module only in our experiments. However, the most important difference is that unlike the method in [13] which focuses on only one particular kind of placement constraint, our method is more general and can handle different kinds of placement constraints simultaneously. We have also tried another data set from the paper [14], which can handle boundary constraint in slicing floorplan. A resultant packing is shown in Figure 16. The amount of deadspace obtained in [14] is smaller because optimal shaping was done in their slicing floorplanner.

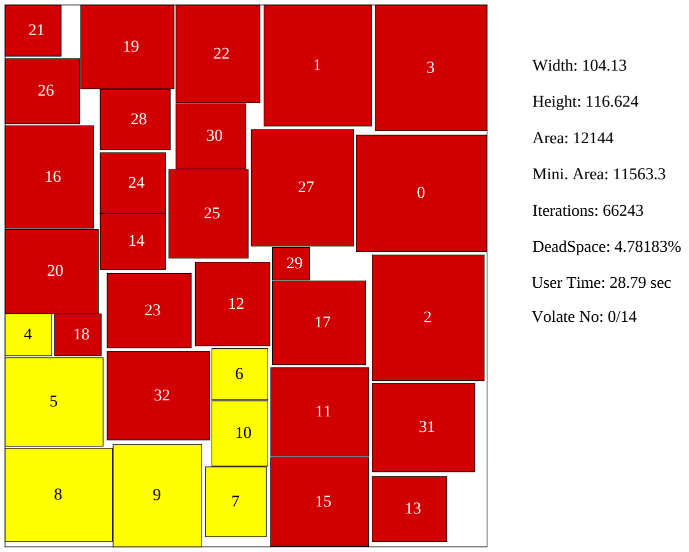
In order to demonstrate the effects of an input set of infeasible constraints, we performed an experiment in which the required set of constraints were contradictory to each other. The resultant packing is shown in Figure 17. In this example, we require module 5 to be packed along the left boundary, and on the right hand side of module 4 at the same time. These contradictory requirements will always lead to positive cycles in the constraint graphs. The floorplanner will then pack the modules as if there is no constraints and the penalty term will drive the result towards one that satisfies the requirements as much as possible, as we can see from the example in Figure 17. Figure 18 shows the growth in runtime with respect to the number of placement constraints and we can see that the relationship is almost linear.

VII. Conclusion

In this paper, we presented a method to handle different kinds of placement constraints in floorplanning simultaneously. In this method, placement constraints are handled by augmenting the constraint graphs with edges of positive, negative or zero weights. We have used incremental updates and an interesting idea of reduced graph to improve the runtime of the algorithm. Several benchmark data are used for testing and the results are very promising. Good packings with all the constraints satisfied can be obtained efficiently.

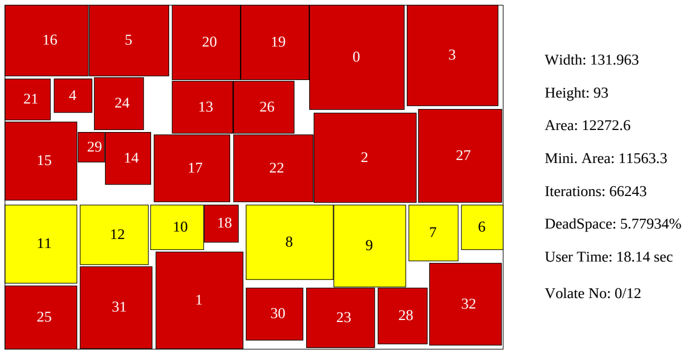
References

[1] F. Balasa and K. Lampert. Symmetry within the Sequence-Pair Representation in the Context of Placement for Analog Design. *IEEE*



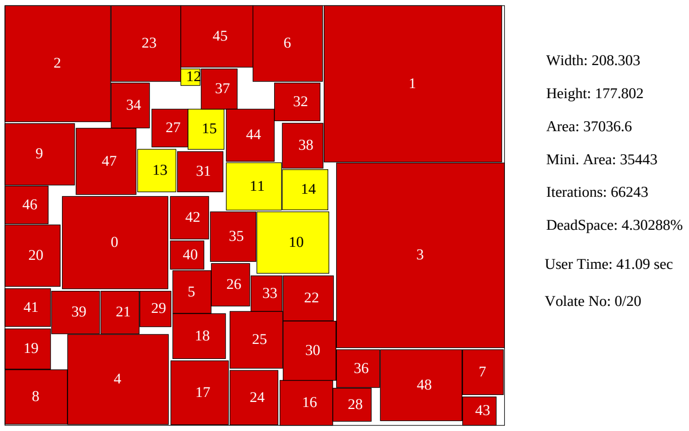
Width: 104.13
Height: 116.624
Area: 12144
Mini. Area: 11563.3
Iterations: 66243
DeadSpace: 4.78183%
User Time: 28.79 sec
Volate No: 0/14

Fig. 11. Modules 4, 5, 6, 7, 8, 9 and 10 cluster at the lower left corner of the chip.



Width: 131.963
Height: 93
Area: 12272.6
Mini. Area: 11563.3
Iterations: 66243
DeadSpace: 5.77934%
User Time: 18.14 sec
Volate No: 0/12

Fig. 12. Modules 6, 7, 8, 9, 10, 11 and 12 align horizontally.



Width: 208.303
Height: 177.802
Area: 37036.6
Mini. Area: 35443
Iterations: 66243
DeadSpace: 4.30288%
User Time: 41.09 sec
Volate No: 0/20

Fig. 13. Modules 10, 12, 13, 14 and 15 cluster around module 11.

	[13] (300MHz Pentium II)			Our Method	
Data Set	Deadspace(%)	Original time (sec)	Scaled time (sec)	Deadspace(%)	Time (sec)
ami33	1.56	53.85	21.89	2.95	43.62
ami49	3.14	118.02	47.98	2.92	65.41
playout	3.00	230.85	93.84	2.76	109.46

TABLE II
COMPARISONS WITH THE RESULTS IN [13].

No. of Placement Constraints	Time (sec)	Deadspace %	Wirelength 10^6	No. of Constraints Violated
ami33 (#module = 33; #net = 123)				
0	15.19	7.20	0.0251	0
4	18.05	6.57	0.0248	0
8	20.04	6.60	0.0252	0
12	19.89	7.56	0.0252	0
14	25.63	7.26	0.0252	0
16	24.05	6.99	0.0252	0
ami49 (#module = 49; #net = 408)				
0	32.45	6.15	1.932	0
4	34.69	6.29	1.934	0
8	36.90	7.37	1.957	0
12	38.63	7.75	1.962	0
16	39.53	7.48	1.957	0
20	42.37	6.08	1.931	0
playout (#module = 62; #net = 1611)				
4	31.33	7.21	0.0191	0
10	36.18	6.61	0.0190	0
16	41.77	7.27	0.0191	0
20	41.77	7.96	0.0193	0
24	45.95	8.35	0.0194	0
random100 (#module = 100; #net = 1611)				
0	279.12	8.26	1.469	0
4	289.03	8.06	1.465	0
10	310.61	8.06	1.431	0
16	325.51	7.41	1.466	0
22	357.39	7.98	1.467	0
28	374.71	6.92	1.448	0

TABLE I
RESULTS FOR AMI33, AMI49, PLAYOUT AND RANDOM100.

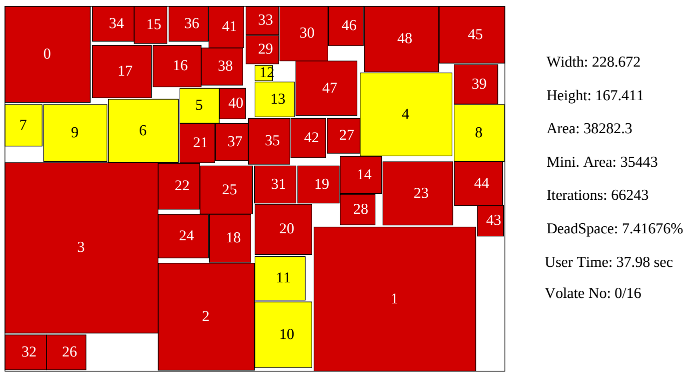


Fig. 14. Modules 4, 5, 6, 7, 8 and 9 almost align horizontally. Modules 10, 11, 12 and 13 align vertically.

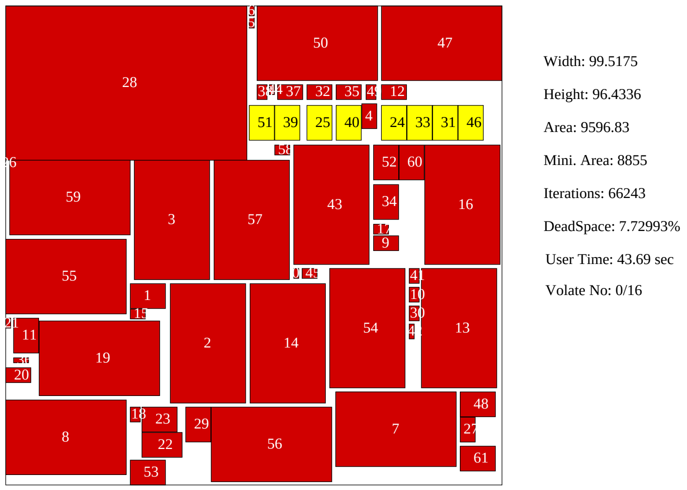


Fig. 15. Module 24 is placed at $y = 20$ and modules 24, 25, 31, 33, 39, 40, 46 and 51 align horizontally.

Transactions on Computer-Aided Design of Integrated Circuits and Systems, 19(7):712–731, 2000.

[2] F. Balasa, S. C. Maruvada, and K. Krishnamoorthy. Efficient Solution Space Exploration Based on Segment Trees in Analog Placement with Symmetry Constraints. *Proceedings of the International Conference on Computer-Aided Design*, pages 497–502, 2002.

[3] Y. C. Chang, Y. W. Chang, G. M. Wu, and S. W. Wu. B*-Trees: A New Representation for Non-Slicing Floorplans. *Proceedings of the 37th ACM/IEEE Design Automation Conference*, 2000.

[4] K. Fujiyoshi and H. Murata. Arbitrary Convex and Concave Rectilinear Block Packing Using Sequence-Pair. *International Symposium on Physical Design*, pages 103–110, 1999.

[5] Jianbang Lai, Ming-Shiun Lin, Ting-Chi Wong, and Li-C. Wang. Module Placement with Boundary Constraints Using the Sequence-Pair Representation. *IEEE Asia and South Pacific Design Automation Conference*, pages 515–520, 2001.

[6] Y-Z. Liao and C. K. Wong. An Algorithm to Compact a VLSI Symbolic Layout with Mixed Constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2(2):62–69, 1983.

[7] Yuchun Ma, Sheqin Dong, Xianlong Hong, Yici Cai, Chung-Kuan Cheng, and Jun Gu. VLSI Floorplanning with Boundary Constraints Based on Corner Block List. *IEEE Asia and South Pacific Design Automation Conference*, pages 509–514, 2001.

[8] H. Murata, K. Fujiyoushi, and M. Kaneko. VLSI/PCB Placement with Obstacles Based on Sequence-Pair. *International Symposium on Physical Design*, pages 26–31, 1997.

[9] H. Murata, K. Fujiyoushi, S. Nakatake, and Y. Kajitani. Rectangle-Packing-Based Module Placement. *Proceedings IEEE International Conference on Computer-Aided Design*, pages 472–479, 1995.

[10] Xiaoping Tang and D. F. Wong. Floorplanning with Alignment and Performance Constraints. *Proceedings of the 39th ACM/IEEE Design Automation Conference*, pages 848–853, 2002.

[11] Thomas H. Cormen and Charles E. Leiserson and Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill, eighth edition, 1992.

[12] F. Y. Young and D. F. Wong. Slicing Floorplans with Pre-placed Modules. *Proceedings IEEE International Conference on Computer-Aided Design*, pages 252–258, 1998.

[13] F. Y. Young and D. F. Wong. Slicing Floorplans with Range Constraints. *International Symposium on Physical Design*, pages 97–102, 1999.

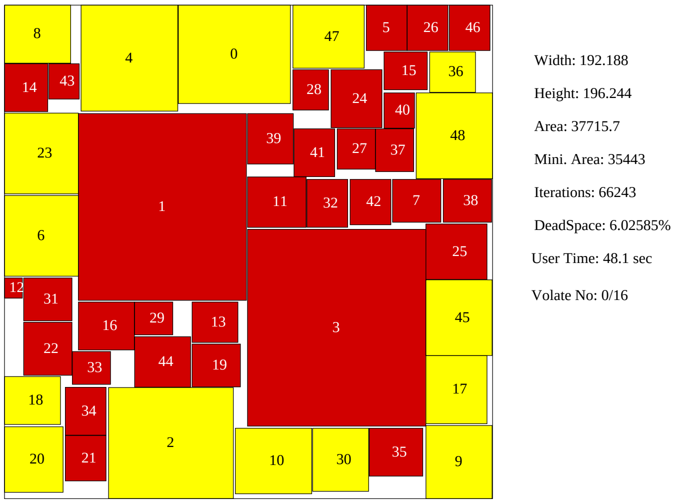


Fig. 16. A resultant packing of the data set ami49-bc1 from [14] in which module 6, 18, 20 and 23 are required to be on the left, module 17, 36, 45 and 48 on the right, module 0, 4, 8 and 47 at the top, and module 2, 9, 10 and 30 at the bottom. The deadspace obtained in [14] was 1.51%.

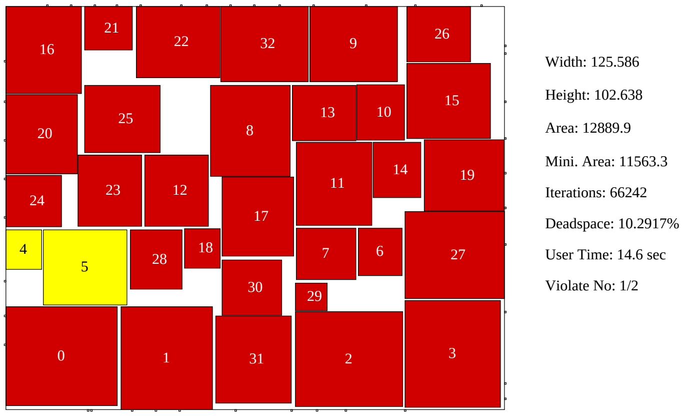


Fig. 17. An input set of infeasible constraints that requires module 5 to be placed along the left boundary, and on the right hand side module 4 at the same time.

[14] F. Y. Young, D. F. Wong, and Hannah H. Yang. Slicing Floorplans with Boundary Constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(9):1385–1389, 1999. Also appeared in ASP-DAC 1999.

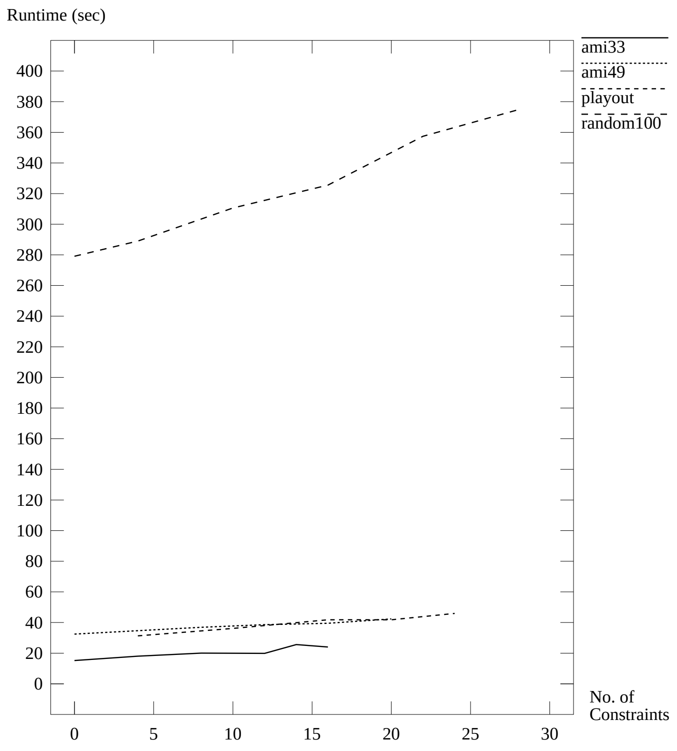


Fig. 18. The relationship between runtime and the number of placement constraints.

PLACE PHOTO HERE

Chris Chu received the B.S. degree in computer science from the University of Hong Kong, Hong Kong, in 1993. He received the M.S. degree and the Ph.D. degree in computer science from the University of Texas at Austin in 1994 and 1999, respectively.

Dr. Chu is currently an Assistant Professor in the Electrical and Computer Engineering Department at Iowa State University. His research interests include design and analysis of algorithms, CAD of VLSI physical design, and performance-driven interconnect optimization. He received the IEEE TCAD best paper award at 1999 for his work in performance-driven interconnect optimization. He also received the Bert Kay Best Dissertation Award for 1998-1999 from the Department of Computer Sciences in the University of Texas at Austin.

PLACE PHOTO HERE

Evangeline Young received her B.Sc. degree and M.Phil. degree in Computer Science from the Chinese University of Hong Kong (CUHK). She received her Ph.D. degree from the University of Texas at Austin in 1999. Currently, she is an Assistant Professor in the Department of Computer Science and Engineering in CUHK. Her research interests include algorithms and CAD of VLSI circuits. She is now working actively on floorplan design optimization, circuit partitioning, circuit retiming and packing representation.

PLACE PHOTO HERE

M.L. Ho received his B.Sc. and M.Phil. degree from the Computer Science and Engineering Department in the Chinese University of Hong Kong in 2001 and 2003, respectively. He is currently working in ThizLinux Laboratory Limited.