Scalability and Interconnection Issues in Floorplan Design and Floorplan Representations

YUEN Wing-seung

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Philosophy
in
Computer Science and Engineering

July 2001

Scalability and Interconnection Issues in Floorplan Design and Floorplan Representations

submitted by

YUEN Wing Seung

for the degree of Master of Philosophy

at The Chinese University of Hong Kong in July 2001

Abstract

Floorplan Design is the problem of planning the positions and shapes of the modules at a very early designing stage of VLSI circuits in order to optimize the circuit performance. During this floorplanning phase, the circuit performance like layout area, interconnect cost, heat dissipation and power consumption, etc, should be minimized. There are several aspects that a floorplanning algorithm has to deal with: the flexibilities of the modules, total area of the chip, routability and delays. All these are essential for optimizing the circuit performance. With the scaling down of the IC technology, the number of transistors that can be built into a standard size chip has increased rapidly and interconnect delay has become a dominant factor in circuit delay. Both scalability and interconnect optimization are currently one of the most important issues in floorplan design.

Placement constraints are useful that some modules are constrainted to be placed at some specify positions of the chip to reduce the interconnect cost and to improve the circuit performance. Clustering constraint in slicing floorplan is considered. It is a constraint in which some modules are required to be placed geometrically adjacent to each other forming a cluster in the floorplan. A linear time algorithm is devised to locate all the neighbors of a module in a normalized Polish expression. By making use of this algorithm, we can address the Clustering Constraint in slicing floorplan effectively.

Multilevel approach is also studied to address the scalability problem in floorplanning. The floorplanning process is divided into two phases: the Clustering Phase and the Refinement Phase. In the Clustering Phase, heavily connected modules are clustered together recursively to reduce the interconnect cost. The modules are then unclustered and packed in the Refinement Phase. Experimental results show that this approach can speedup the floorplanning process for even very large circuits and can reduce the interconnect cost significantly.

Slicing floorplans have two important advantages: they have small solution space and the best dimensions of the modules can be computed in linear time. However, slicing floorplan can only represent a small subset of packings. Many representations are proposed in order to represent non-slicing floorplans effectively and we have studied and analyzed different floorplan representations. Twin binary tree is one of the newly proposed representations for mosaic floorplan. We have proved an interesting property in non-slicing floorplan and devised an efficient algorithm to generate pairs of twin binary trees and to convert a twin binary tree to its corresponding packing.

佈圖的可變性與互連及佈圖的代表式

作者: 袁詠湘

論文摘要

佈圖是應用在超大規模集成電路結構設計的較早階段，主要通過計算模件的擺放位置與其尺寸，把版面面積、互連、熱能與能量消耗等影響電路性能的因素盡量減低，從而提高電路的性能。此外，佈圖的演算法也會考慮模件的形狀、大小、延時及可佈線性等問題。由於集成電路的特徵尺寸日益縮小，一片晶片可容納的電晶體數目大大增加，互連延時因而成了電路延時的主要因素；可變比例性與互連則成了現今佈圖面對的主要問題。

佈局限制可以把所需的模件放置於指定的區域，從而縮短模件與連接口以及多互連模件之間的距離。我們把群集限制用於分片佈圖中。這限制是在佈圖上把指定的模件放於一起。我們設計了一個線性的演算法，此演算法能從分片佈圖表達式中找出在佈圖上一個模件的周圍模件，利用此演算法，群集限制便可用於分片佈圖中。

多層法是一個解決佈圖上可變比例性問題的有效方法。我們把佈圖的過程分成兩部份，分別爲群集和求精。在群集的過程中，多互連的模件會組成一組，從而提早減低互連的成本。之後，模件會在求精的過程中作佈圖。從實驗結果得知，這方法可以減少運算的時間與互連接距離。

分片佈圖有著解空間小及可以有效地計算模件的最佳尺寸等特點，這是非分片佈圖所不能做到的。現有很多的非分片佈圖表達式，都沒有分片佈圖的這些優點。我們分析了不同佈圖表達式，而且更證明了一個非分片佈圖的特性。

Acknowledgments

I would like to acknowledge the support and guidance of my supervisor, Prof. Evangeline F. Y. Young. A lot of ideas and problems are raised by her and I can just finish some of them. She has allowed me to explore different techniques in solving the problems, and I have developed deep interest in the field of Computer Aided Design during this research work even through I did not know much of it at the beginning. She is nice, patient and enthusiastic. I warmly thank her.

I would also like to thank the CAD research group in our department. The supervisors of the group include my supervisor and Prof. David Y. L. Wu. In the meeting, we have shared our research works, problems, comments and supports. It is interesting and supporting to raise and discuss the problems together. I would like to thank all the members of the group, including Mr. Ray C. C. Cheung, Mr. Bruce C. W. Sham, Mr. Cliff C. N. Sze and Mr. Keith W. C. Wong.

I warmly thank all my classmates including Mr. Ivan K. H. Leung, Mr. Norris M. P. Leong, Ms. Polly P. M. Wan, Mr. Hiu-Yung Wong who are working in the same office with me days and nights. They have made my postgraduate school life enjoyable. Also, I kindly thank all my students in the courses. They are smart and excellent, that allows me to have abundant time to concentrate on my research works.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivations and Aims

Integrated Circuit (IC) Technology is a revolution of this century. All the electronics devices including computers, phones, appliances, etc, are using ICs as the microprocessor, memory and interface chips. Devices become smaller, faster and more advanced due to the scaling down of the ICs technology.

Very Large Scale Integration (VLSI) is a term describing a chip that is integrated of up to millions of transistors. It is difficult to handle up to millions or even billions of transistors. The transistors have to be placed, the connections between them have to be routed, and all these have to be done within a limited area on a chip. Optimization is very important in Physical Design of VLSI circuits and automation is unavoidable in speeding up this very large scale and complicated optimization.

Computer Aided Design (CAD) in VLSI is playing an important role in the IC industry. Tools and algorithms are developed to solve different kinds of optimization problem in the Physical Design phase. This can help the engineers to develop their chips and products efficiently.

Floorplan Design is an important step in VLSI Physical Design. The objective of floorplanning is to plan the positions, shapes and dimensions of the modules on a chip to optimize circuit performance like total chip area,

routability, circuit delay, power consumption, etc., at a very early designing stage.

With the scaling down of the technology in the IC development, the number of transistors that can be built into a standard size chip has increased rapidly. The functionality of the chip has also become more and more complicated. We need to handle large problem size in floorplanning and scalability has become an important issue [Kah00]. Most traditional floorplanners are unscalable since the process to obtain a good packing when everything is still very flexible is non-trivial and time consuming. They are unable to handle large problem size with even a few hundreds of modules in practical time. The runtime required will grow exponentially with the problem size.

Interconnect optimization is another major concern in Floorplan Design. In some advanced systems today, as much as 80% of a clock cycle is consumed by interconnect and interconnect delay has dominated the system performance. We should consider interconnect as early as possible so that timing closure can be achieved more effectively in the later designing stages. In the floorplanning stage, we only got some preliminary information like module area, net information and pin positions. Interconnect cost can only be estimated roughly by some simple methods at this stage such as half-perimeter and center-to-center estimations. We want to minimize these values in the floorplanning stage to improve the interconnect cost of the final circuit.

The objective of this project is to investigate and study the methods to address the scalability and interconnect issues in floorplan design, and to study the representations of floorplan in general to achieve a better understanding of this important problem.

Placement constraints in floorplan design are useful for specifying the placement relationship between the modules according to their functionality in order to improve the circuit performance like interconnect cost and delay. Some previous works on placement constraints in slicing floorplans [YW99b, YW99a]

have been done. Clustering constraint is considered in our research work in which some modules are required to be placed next to each other. The cost of routing can be reduced by imposing clustering constraint to the modules which are heavily connected.

Multilevel approach is a good solution to address the scalability and interconnect issues in floorplan design. Multilevel approach has been used in circuit partitioning [KK95, AHK98, WA98, KAKS99] to handle large circuits. It consists of two phases: clustering and refinement. The Clustering Phase groups modules with heavy interconnection together and the Refinement Phase performs partitioning. We are pioneers in applying this multilevel technique in floorplan design and its applicability is strongly supported by the very promising experimental results.

1.2 Contributions

This thesis will present three pieces of work: slicing floorplan with clustering constraint, interconnect driven multilevel floorplanner and some studies of the properties and representations of non-slicing floorplan.

- Clustering Constraint is considered in slicing floorplan. Given a set of modules M and a subset of modules $S \subseteq M$, we want to pack those modules in M such that the modules in S will be geometrical adjacent to each other. The wiring cost can be reduced by putting modules with a lot of connections closely together. Designers may also need this type of placement constraint to pack the modules according to their functionality.

  A method addressing clustering constraint in slicing floorplan is presented. A linear time algorithm is devised to locate neighboring modules in a normalized Polish expression and to re-arrange the modules such that

the floorplan generated is feasible and satisfys the given constraints.

Experiments were performed for one to five clusters in a packing on different MCNC benchmarks and the results are promising. The runtime has only increased by 10% on average and the resulting deadspace is similar to that of the original floorplanner when no clustering constraints is imposed.

- Multilevel technique is applied to floorplanning in order to reduce runtime and to better optimize the interconnect cost of the final packing. This technique has been found to be very efficient in reducing the runtime for the circuit partitioning problem and we want to find out if this technique is also applicable to floorplanning.

  The clustering and refinement methods for the multilevel approach are devised. Experiments were performed for up to a thousand of modules in the packing. The wirelength of the result packing is improved by about 10% on average in comparison with those without using multilevel approach. The runtime is halved and a small deadspace can be obtained.

- Some interesting properties of non-slicing floorplan are studied. We have proved that a non-slicing floorplan must contain at least a module (or a supermodule) with four T-junctions of different orientations at its four corners. We also devised an efficient algorithm to generate pairs of valid twin binary tree and to convert a twin binary tree to its corresponding packing.

1.3 Dissertation Overview

This thesis is consisted of nine chapters. Introduction to Computer Aided Design in VLSI circuits and Floorplanning is given in Chapter 2. Chapter 3 describes different types of floorplan representations. For each representation,

the transformation from the abstract representation to the real packing will be described. Some optimization techniques commonly used in floorplan design will be presented in Chapter 4. Chapter 5 is a literature review on interconnect driven floorplan design which includes the placement constraint, buffer block planning and wirelength estimation model. Chapter 6 presents the work on clustering constraint in slicing floorplan. The algorithm details and procedure will be given in this chapter. Chapter 7 describes the work on multilevel floorplanner. The clustering and refinement procedures are applied to the floorplanning problem. Chapter 8 is a study of the properties and representations of non-slicing floorplans. Finally, the conclusion will be given in the last chapter.

Chapter 2

Physical Design and Floorplanning in VLSI Circuits

This chapter will briefly introduce several important stages in Physical Design of VLSI circuits. They are partitioning, floorplanning, placement, routing and compaction. These steps are essential for optimization can optimize and layout of the chips. Problem formulations and types of floorplan will also be discussed in this chapter.

2.1 VLSI Design Flow

Physical Design translates a circuit diagram into its layout. During this process, we need to minimize the chip area and signal delay in order to improve the circuit performance. It consists of several steps including Partitioning, Floorplanning, Placement, Routing and Compaction [She99]. Figure 2.1 shows the flow diagram of these processes. For each circuit, it will go through these steps to obtain a good layout before fabrication.

The partitioning step divides a circuit into several parts such that the net connections between the parts are minimized. Partitioning is significant because the number of pins on the chip is limited and also a good partition can reduce the circuit complexity significantly. Some systems today may consist of

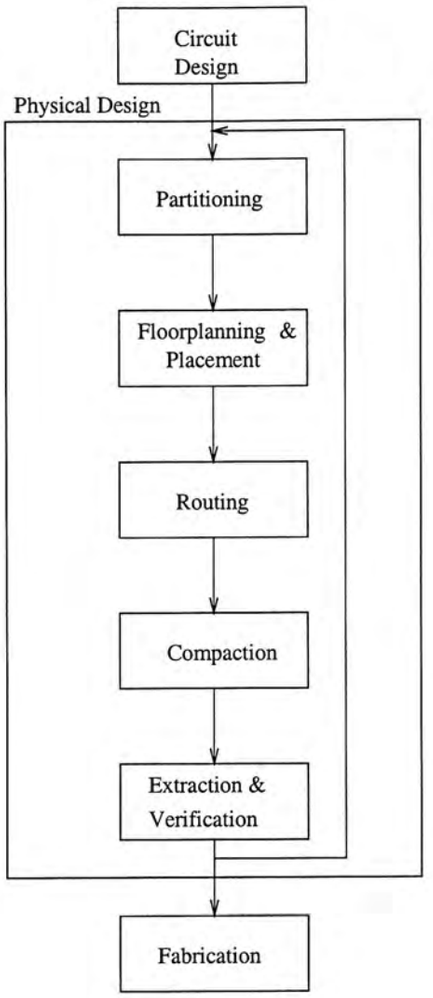


Figure 2.1: Physical Design Cycle

millions of transistors on one single chip. It is impossible to layout the entire chip fast due to the limitation in computational power and memory. Circuits are usually partitioned recursively into smaller sub-circuits. Actually, the process will also consider the size of the modules, the number of modules and the number of interconnections between the modules. It is beneficial to minimize the number of connections between the partitions and keep the size of the partitions similar. Kernighan-Lin [KL70] and Fiduccia-Mattheyses [FM82] methods are two famous algorithms in solving the graph bi-partitioning problem in which we want to divide a graph into two partitions with similar size.

Floorplanning and Placement are an important step that determine the

shapes and positions of the modules on a chip. This will greatly affect the area and the dalay of the chip. The area of each module can be estimated since the amount of gates and logics are known after circuit partitioning. The interconnect information are also obtained from the partitioning process. These are the input to the floorplanning process and the output is the packing that with the chip area and interconnect cost optimized. The dimensions and shapes of the modules are also varies. It is a computational intensive problem to determine both the dimensions and positions of the modules such that the chip area and interconnect cost are minimized. Placement determines the exact positions of the modules on the chip with more detailed information like pin assignment, module's dimensions, such that the chip area and delay constraint are satisfied.

After the modules are placed, routing completes the interconnect between the pins of the modules. Routing may include two phases – global routing and detailed routing. Global routing determines a rough route for each net. It determines the regions in which the route should go to complete the connection. Detailed routing completes the route within the regions. The regions can be a channel, a box or over the modules. It is a hard problem because the resources for routing is limited and routablity is an important issue.

The area of the layout will be further reduced in the Compaction stage. Compaction is to compress the layout in all directions such that the deadspace within a chip can be reduced, resulting in a smaller total chip area. After all the above stages, the layout have to be verified and checked whether it satisfy the timing and function requirement. Otherwise, the design flow have to be repeated until all requirements are satisfied.

2.2 Floorplan Design

Floorplanning is the problem of placing and sizing the modules after the circuit is partitioned into different units. During this floorplanning phase, the total

area of the layout and the interconnect cost should be minimized.

The input to this floorplanning problem are the areas of the modules, the possible shapes of each module, possibly the I/O pins of each module and the netlist between the modules.

There are two types of modules, namely hard modules and soft modules. Hard modules are the modules whose dimensions and shapes are fixed. Soft modules are the modules which dimensions and shapes are not known though their areas can be estimated by knowing how much logic each contains. There may be restrictions to the aspect ratio and orientation of the soft modules. In floorplanning, most of the modules are flexible and the floorplanning step is to determine the positions, dimensions and shapes of the modules.

There are several aspects that a floorplanning algorithm has to deal with: the shapes of the modules, routablity, area and delays. All these are essential for optimization of the circuit performance.

2.2.1 Problem Formulation

A floorplan with n modules (1, 2, ..., n) is an enveloping rectangle R subdivided by horizontal and vertical line segments into n or more non-overlapping rectilinear regions [WL86] such that each region R_i must be large enough to accommodate the corresponding module i.

A floorplan is evaluated by its packing area and interconnect cost. An $n \times n$ matrix C can represent the interconnections between n modules. $C = (c_{ij})_{n \times n}$ with $c_{ij} \geq 0, 1 \leq i, j \leq n$, is the number of wire between each pair of modules. The most common method is that for every pair of modules i and j, we use d_{ij} to denote the distance between i and j. Then W, an estimate of the total interconnection wire length, can be computed as $\sum_{1 \leq i,j \leq n}(c_{ij}d_{ij})$. In most iterative methods, a floorplan is evaluated by the function $A + \lambda W$. where A is the area and W is the wire length. The overall aspect ratio of the floorplan

also needs to be considered.

The aspect ratio for each module will be limited to a range so that the routing inside each module will not be very long. For each rectangular modules i, there are three input values A_i, r_i and s_i. where A_i is the area of the module, r_i and s_i are the minimum and maximum aspect ratio of the module respectively. The list of three tuples (A_1, r_1, s_1), (A_2, r_2, s_2), ..., (A_n, r_n, s_n) thus represent n modules to be packed. Let w_i and h_i be the width and height of module i, then $A_i = w_i h_i$ and $r_i \leq \frac{h_i}{w_i} \leq s_i$.

The input for non-rectangular modules will be different. For example, an L-shaped module will be represented by a five tuples (x_1, x_2, y_1, y_2, s) to describe the geometric figure of the module where s denotes orientation of the module.

2.2.2 Types of Floorplan

There are two types of floorplan: slicing and non-slicing. A slicing floorplan is a floorplan which can be obtained by recursively partitioning a rectangle into two parts either by a vertical line or a horizontal line. An example is shown in Figure 2.2.

A non-slicing floorplan is a floorplan which is not slicing. An example is shown in Figure 2.3. Non-slicing floorplans are flexible and more general that they can represent any packing. However, it is difficult to find a good representation for non-slicing floorplan to handle shaping efficiently.

Bound-sliceline-grid [NFMK96], sequence-pair [MFNK95, KD98, MFK98], O-tree [GCY99], B*-tree [CCWW00] and corner block list [HHC+00] have been proposed to represent non-slicing floorplans. Polish expression [WL86] is used to represent slicing floorplans.

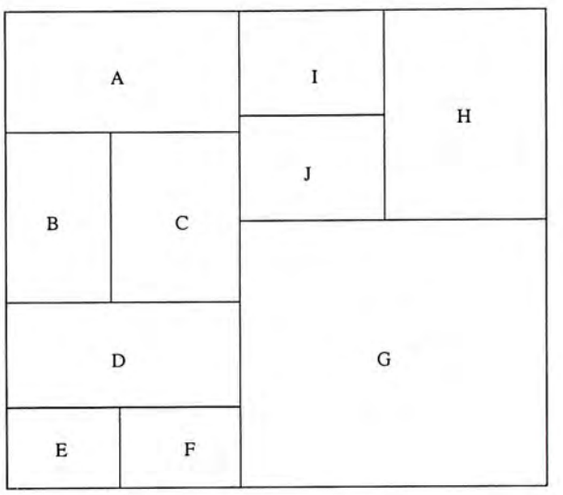


Figure 2.2: A Slicing Floorplan

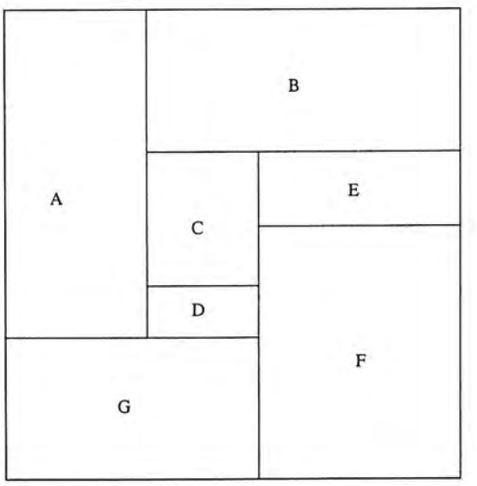


Figure 2.3: A Non-slicing Floorplan

Chapter 3

Floorplanning Representations

This chapter will introduce several floorplan representations including Polish expression(PE), Bounded-Sliceline-Grid(BSG), Sequence Pair(SP), O-tree(OT), B*-tree(BT) and Corner Block List(CBL). For each representation, the floorplan construction method, moves and some extension works will be briefly introduced.

3.1 Polish Expression(PE) [WL86]

This representation is for slicing floorplans. Otten has proposed a pioneer representation of slicing floorplan using slicing trees [Ott82]. The leaf nodes of the slicing tree represent the modules in the floorplan. The internal nodes, labelled either with a '+' or a '*', represent the directions of the cuts in the floorplan. If an internal node is labelled by a '+', the cut will be horizontal. The module represented by the left child will be below that of the right child in the floorplan. Similarly, a '*' represents the vertical cut, and the module represented by the left child will be on the left of that of the right child. This is shown in Figure 3.1 and Figure 3.2. If we read the slicing tree in postorder we will obtain an expression called the Polish expression which can represent the floorplan structure [WL86].

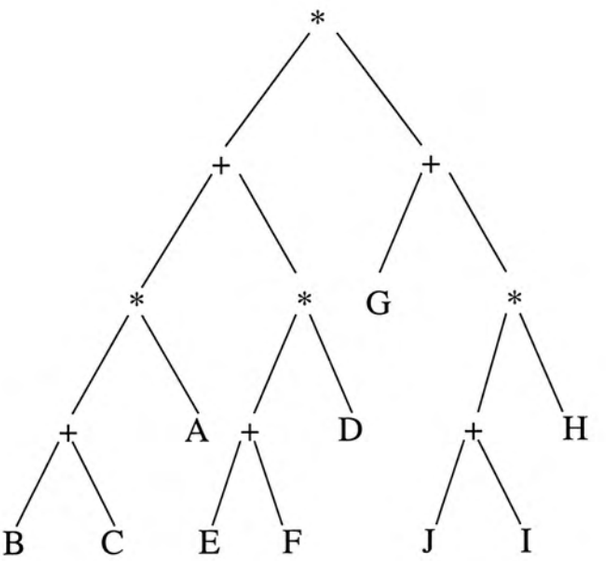


Figure 3.1: Slicing Tree

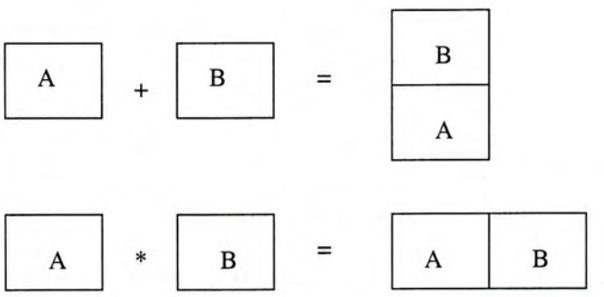


Figure 3.2: Binary operations for slicing floorplans

The normalized Polish expressions is used to represent a solution. This expression is particularly suitable for the method of simulated annealing such that the floorplan can be modified by simply changing and swapping the operands and operators. The expression is said to be normalized if and only if there is no consecutive *'s and +'s.

The area of the whole packing can be computed recursively from the leaves to the root of the slicing tree in linear time. The slicing tree is a binary tree. All the leaf nodes of the slicing tree are operands which represent the modules in the packing. The parents of the modules are the operators. For every two modules with the same parents in the slicing tree, they are either placing horizontally or vertically adjacent to each other to form a supermodule. The computation is done recursively at every nodes in a bottom up manner. The whole packing can be obtained at the root finally using this method.

There are three kinds of moves. (1) Swap two adjacent operands,(2) Interchange the operators '*' and '+' in a chain. (3) Swap two adjacent operand and operator [WL86].The operands are from $1, 2, \ldots, n$ which represent the rectangular modules in the floorplan. The operands are either '+' or '*' in the Polish expression. For each iteration, one of the move will be selected and the cost of the floorplan will be calculated. The three types of moves are enough to transform an initial solution into any other expressions [WL86].

3.2 Bounded-Sliceline-Grid(BSG) [NFMK96]

Bounded-Sliceline-Grid(BSG) is a representation for non-slicing floorplan. This method is based on the bounded-sliceline grid structure (shown in Figure 3.3) with size $p \times q$ where $p \times q \geq n$ and n is the number of the modules of the packing problem. We may notice that BSG suffers a major problem that there are huge redundancies if the size of the grid structure is large.

Horizontal and vertical unit adjacency graphs can be obtained from the

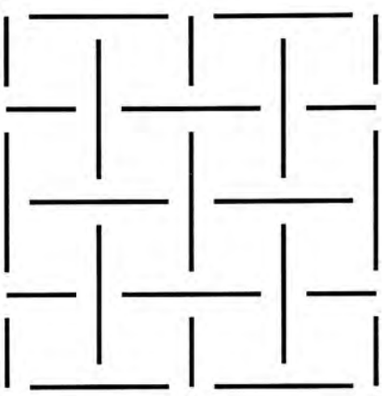


Figure 3.3: A 4×4 Bounded Sliceline Grid structure

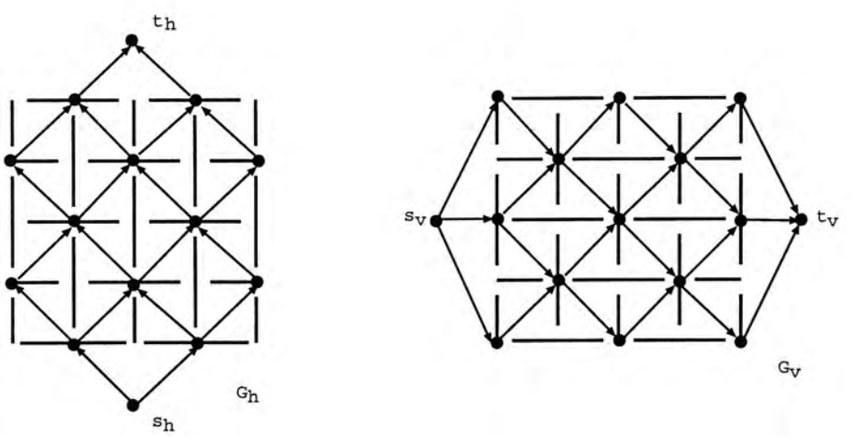


Figure 3.4: Horizontal and vertical unit adjacency graphs

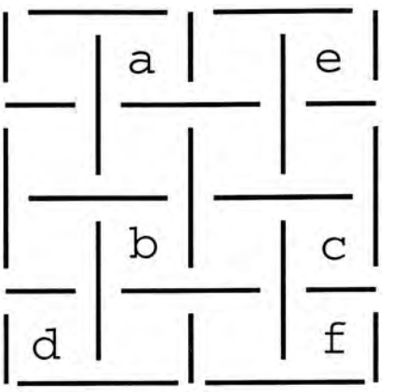


Figure 3.5: One of the assignment of module into the grid

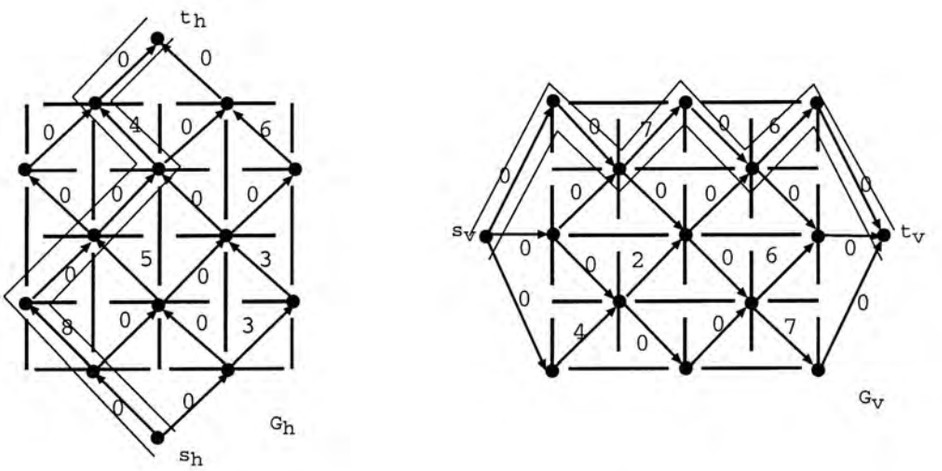


Figure 3.6: Horizontal unit and vertical unit adjacency graphs of the BSG assignment in Figure 3.5

BSG as shown in Figure 3.4. In $G_h = (V_h, E_h)$ and $G_v = (V_v, E_v)$, a source node s_h and s_v and a destination node t_h and t_v are added respectively.

Given a set M with n modules, the modules are assigned into the rooms of the BSG. Figure 3.5 gives one of the assignments of the modules in $M = \{a, b, c, d, e, f\}$ into the grid. The weights on the edges of the adjacency graphs can be obtained from the widths and heights of the modules. We have a weighted adjacency graph shown in Figure 3.6. The longest paths in the graph G_h and G_v are the width and height of the final packing respectively.

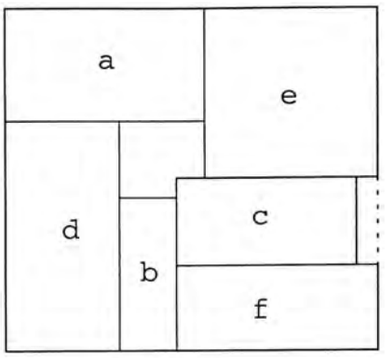


Figure 3.7: The packing correspond to the BSG assignment in Figure 3.5

Let $l_h(u)$ be the longest path length from s_h to u in G_h. It is the x-coordinate of the bottom left corner of u in the packing. Similarly, $l_v(u)$ is the longest path length from s_v to u in G_v. By finding all pairs of $l_h(u)$ and $l_v(u)$, we know the positions of all the modules and the packing is obtained. Figure 3.7 gives a packing corresponding to the BSG assignment in Figure 3.5. Packing is changed by assigning the modules into a different set of rooms in the BSG.

3.3 Sequence Pair(SP) [MFNK95]

A sequence-pair (Γ_+, Γ_-) is an ordered pair of module names. Given a set of module $M = \{a.b, c, d\}$, $(abcd, dbca)$ is one sequence pair for the modules in M. A sequence pair can represent all kinds of packings according to the following two rules:

H-constraint: If $s = (\ ..\ a\ ..\ b\ ..,\ ..\ a\ ..\ b\ ..\ )$, module b is on the right hand side of module a.

V-constraint: If $s = (\ ..\ a\ ..\ b\ ..,\ ..\ b\ ..\ a\ ..\ )$, module b is below module a.

From the rules, we can construct the vertical constraint graph G_v and the horizontal constraint graph G_h in $O(n^2)$ time. There is an example shown in

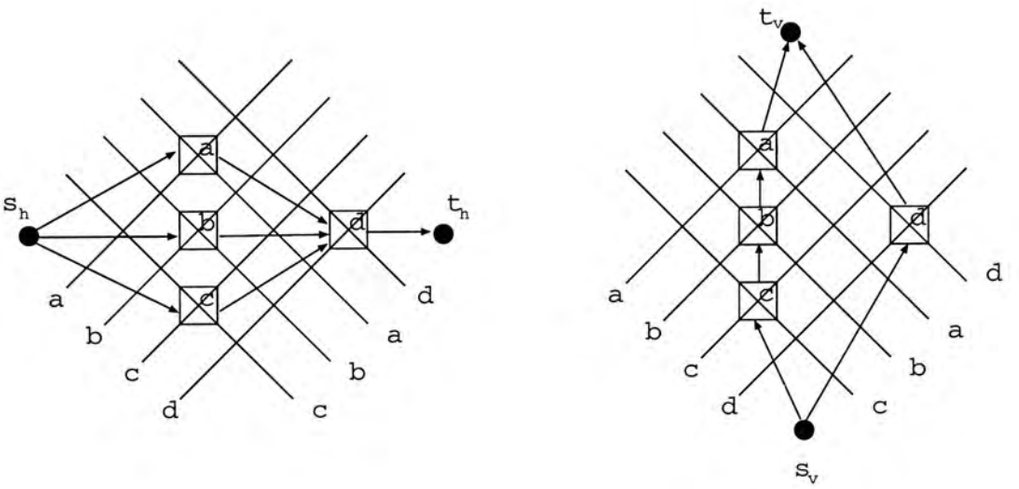


Figure 3.8: H-constraint and V-constraint graph by the given sequence pair (abcd, cbad)

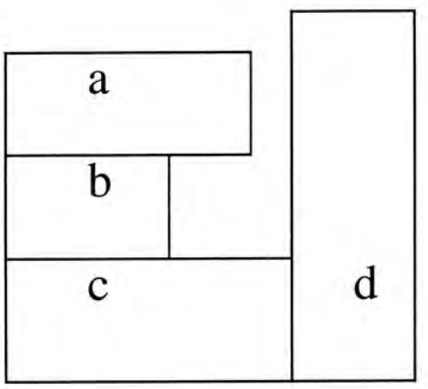


Figure 3.9: The packing corresponding to the sequence pair in Figure 3.8

Figure 3.8. The vertices of the graphs represent the modules. The directed edges between the vertices represent the H-constraint and the V-constraint. In G_v, an edge $e(i, j)$ of weight y denotes that module i is below module j by at least a distance of y. Similarly, an edge $e(i, j)$ of weight x in G_h denotes that module i is on the left hand side of module j by at least a distance of x. Using the graph representations, we are able to determine the dimensions of the packing by computing the longest paths in the graphs. From the constraint graphs in Figure 3.8, a packing can be constructed as in Figure 3.9.

There are two kinds of moves in the sequence pair representation in the annealing process. They are exchanging two modules in the first sequence, and exchanging two modules in both sequences.

Besides, there are some research works focusing on improving the complexity of the packing construction algorithm of sequence pair using some sophisticated data structure. The papers [TTW00] and [TW01] presented the work that improved the the complexity of the algorithm to $O(n \log n)$ and $O(n \log \log n)$ respectively. The method in [TTW00] is based on some properties of the longest common subsequence in a sequence pair. [TW00] improved the method in [TTW00] by the priority queuing technique and give a better complexity.

3.4 O-tree(OT) [GCY99]

These representations of non-slicing floorplan are based on ordered trees. So, it is called O-tree.

An ordered tree consists of an ordered set of subtrees $T = \{T_1, T_2, \ldots, T_m\}$ where $m \geq 0$. The root of the tree has zero or more children, and every node of the tree can be visited using depth-first-search.

The topology of the tree can be represented by a $2(n-1)$-bits string. Together with the sequence of nodes visited, we can represent an ordered tree using two strings. The two tuples (T, π) where T is a bit string and π is a sequence of nodes is used to represent an ordered tree. Figure 3.10 is an ordered tree represented by the two tuples (00110100010111, dacfgeb).

A horizontal O-tree (T, π) can be used to represent a packing. We use x_i and w_i to denote as the x-coordinate and width of node i on the tree. The root of the tree represents the left boundary of the packing, i.e., $x_{root} = 0$ and $w_{root} = 0$. For other nodes in the horizontal O-tree, $x_j = x_i + w_i$ if node i is the parent of node j. The x-position of module j is the sum of w_i of all the

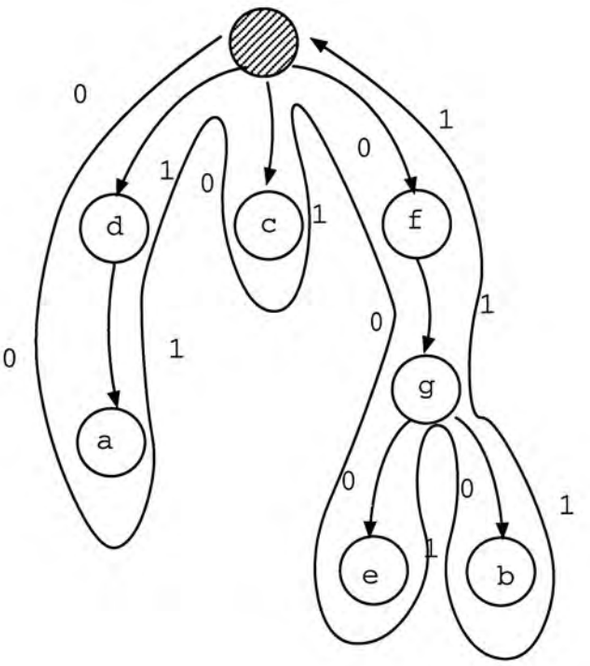


Figure 3.10: The depth first search of an ordered tree

modules lying on the path from the root to j. The y-position of the modules are determined by the permutation π. Let $\psi(i)$ be the set of modules M_k with its order lower than M_i in the permutation π and its interval $(x_k, x_k + w_k)$ overlaps with the interval$(x_i, x_i + w_i)$ If there is no such $\psi(i)$ for node i, $y_i = 0$ in which there is no module with lower order of permutation is placed at its x-interval. The module i will then be placed along the lower boundary. Otherwise, $y_i = max_{k \in \psi(i)}(y_k + h_k)$. According to above transformation, we can convert an O-tree representation into its corresponding packing in linear time. Figure 3.11 shows the packing corresponding to the tree in Figure 3.10.

However, this representation use sequence encoding that inevitably limits the insertion position and solution quality. Besides, this packing scheme only runs in a one-dimensional manner which may not lead to a good placement since the modules are placed in a two-dimensional plane.

The move of the O-tree representation consists of several steps. First, a module M_i in the original tree is selected. M_i is removed from the tree and

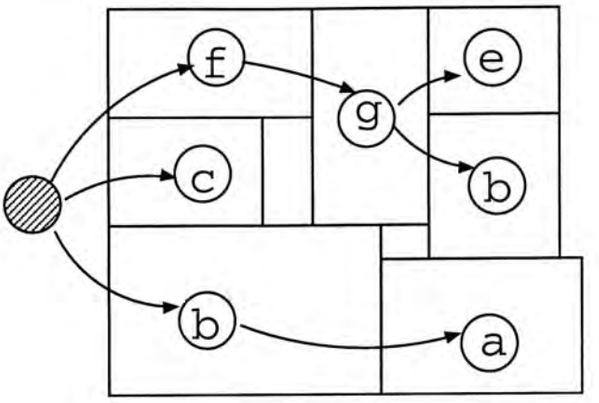


Figure 3.11: A pack correspond to the O-tree in Figure 3.11

then inserted back to the position with the best value of cost function among all the other possible inserting positions.

3.5 B*-tree(BT) [CCWW00]

It is based on ordered binary tree and the admissible placement O-tree discussed previously. With the nice properties of order binary tree, it gives a better complexity in insertion, deletion and searching in comparison with the O-tree representation.

The packing can be constructed from a B*-tree representation using a recursive method. The root node of a B*-tree corresponds to the module at bottom left corner. Let a node n_i in the B*-tree corresponds to the module b_i in the packing. Let R_i be the set of modules located on the right and adjacent to b_i. The left child of n_i will correspond to the lowest module in R_i that is unvisited. If a node n_j is the right child of a node n_i module b_j will be placed above b_i. If a node n_j is the right child of a node n_i, module b_j will be placed on the right of b_i. Figure 3.12 shows a B*-tree and its corresponding packing.

The packing can be changed by following methods: (1) Swapping two modules; (2) Delete a node from the tree and insert it to another place.

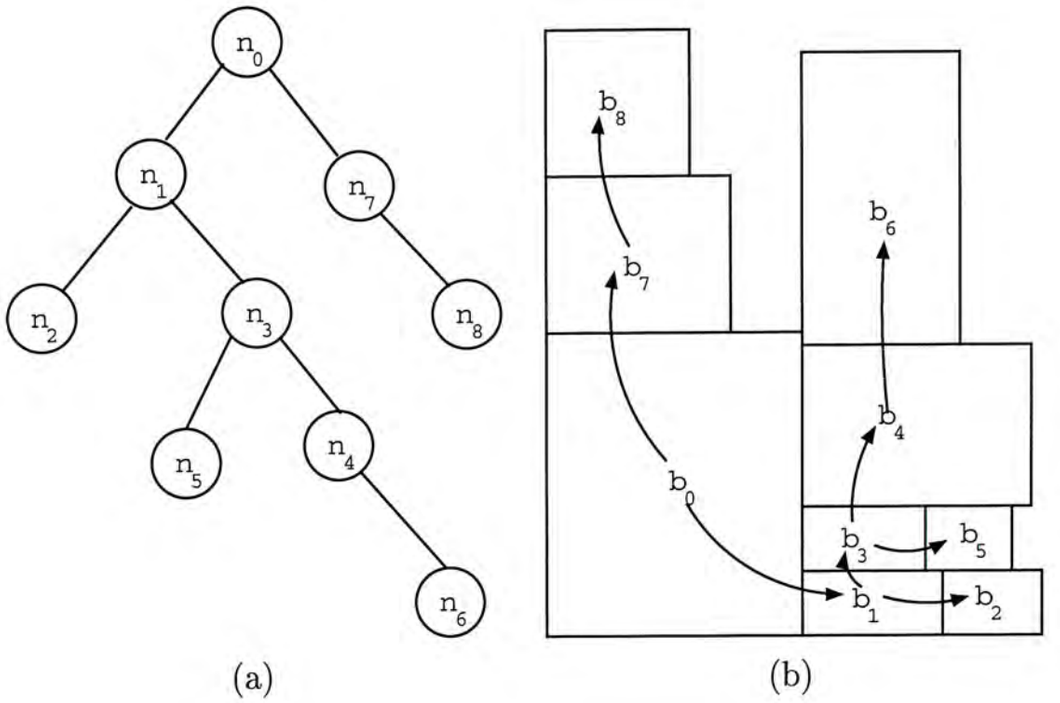


Figure 3.12: An example of B*-tree and its correspond packing

3.6 Corner Block List(CBL) [HHC+00]

It is a topological representation for non-slicing floorplan. The representation consists of three tuples (S, L, T) called corner block list. S is a sequence of module names, L is a list of orientations and T is a list of T-junction information.

S is the order in which modules are inserted into the packing from the right top corner. L consists of $(n-1)$ bits where n is the number of modules in S that indicates the orientations of the modules when being inserted into the packing. There are two types of orientation: horizontal and vertical, denoted by a '1' and a '0' respectively. For example, module g in Figure 3.13 is horizontally oriented and the T-junction at its bottom left corner is rotated by 180 degrees. If the T-junction is rotated by 90 degrees in anti-clockwise, the modules is vertically oriented (like module d and e).

T is a sequence of '0' and '1' bits that counts the number of T-junctions

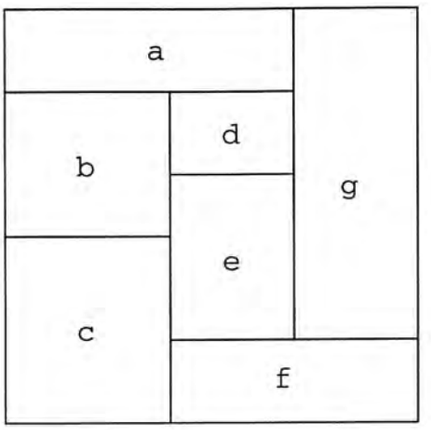


Figure 3.13: An example non-slicing floorplan

attached to a corner block. The number of consecutive '1's corresponds to the number of T-junctions attached to a corner block and the '0's are delimiters to separate one sequence of '1's from another.

The corner block list representation of a packing can be done by removing the corner block from the packing one by one. If the corner block is horizontally oriented, the left segment of the module is shifted to the right boundary of the packing. For example, in the first deletion step of Figure 3.14, the left segment of g is shift to the right boundary such that modules a, d and e become adjacent to the right boundary of the packing. Similarly, if the corner block is vertical oriented, the bottom segment of the module will be shifted to the top boundary of the packing. The attaching T-junctions will be pulled along with the segment.

While deleting a corner block from the packing, its module's name, orientation and number of attaching T-junctions will be recorded into the list S, L and T respectively. Figure 3.14 shows the deletion step of a packing. Therefore, for the packing in Figure 3.13, we will obtain the corner block list (cbfedag, 010001, 0010010110). A packing can be obtained from the corner block list by applying the procedures reversely, i.e., inserting the modules in the sequence of S back to the packing using the information in L and T.

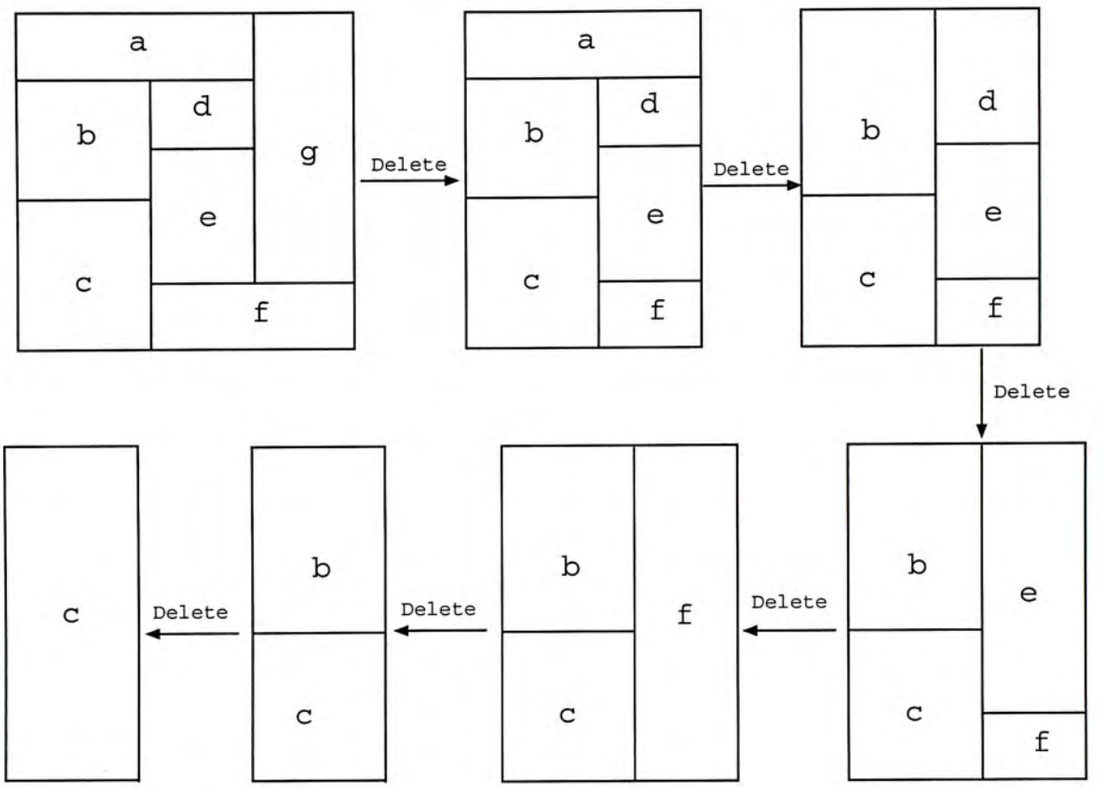


Figure 3.14: Illustrates deletion of corner block of the packing in Figure 3.13

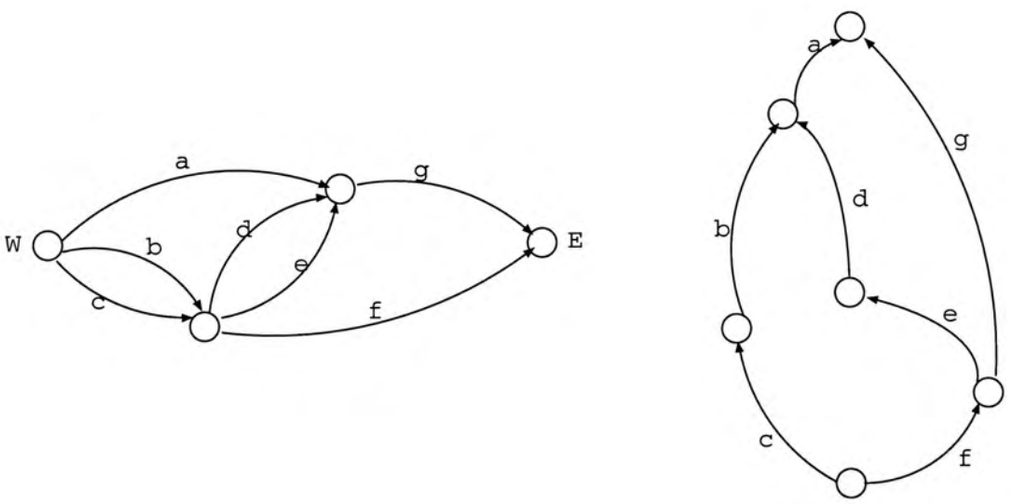


Figure 3.15: Horizontal constraint graph and vertical constraint graph of packing in Figure 3.13

The horizontal constraint graph (HCG) and vertical constraint graph (VCG) (shown in Figure 3.15) are introduced to perform the insertion and deletion of the corner block. In HCG, the west pole (W) and east pole (E) represent the left and right boundary of the packing respectively. The top and the bottom boundaries of the packing is represented by the south pole (S) and north pole (N) in VCG. In the constraint graphs, the edges represent the modules and the nodes represent the segments in the packing. Note that the constraint graphs are planar.

If a module is a corner block, its corresponding edges in HCG and VCG will be pointing to the destination nodes. As we remove the corner block g from the packing, the edges will be removed and the graph will be changed If the source node of a deleted edge e has no outgoing edge after the removal of e, all edges pointing this node have to be changed to point to the destination node of e after the removal. This node can then be removed from the graph.

The corner block list representation has a disadvantage that it can only represent mosaic floorplans. A mosaic floorplan is a floorplan without empty room such that each room is assigned one and only one module. Also, it is topological equivalence on segment sliding. This means that slides of a non-crossing segment of T-junction give the same representation. Besides, it is non-degenerate topology that the case of two distinct T-junctions meet at the same point is not considered. Therefore, this representation cannot represent all possible packings especially for those with empty space in the packing. The Extended Corner Block List (ECBL) is introduced to overcome this problem [ZDH+01].

In ECBL, some dummy modules are introduced into the packing called false block(FB). Their widths and heights are zero. The FB are included in the insertion and deletion procedure in the corner block list. An extending factor λ is introduced such that the total number of rooms in the packing is λn and the number of FB is $\lambda n - n$. This factor will affect the quality of

packing significantly.

There are three kinds of moves in CBL representation during the annealing process. (1) Randomly exchange the order of the modules in S. (2) Randomly change an orientation in L. (3) Randomly change a bit in T. Other moves include changing the orientations and the shape is of the modules. However, the moves may lead to some infeasible representations which cannot be converted to a packing.

Chapter 4

Optimization Technique in Floorplan Design

There are two main types of optimization techniques used in our approaches for floorplan design: deterministic and non-deterministic. For the packing process, we used simulated annealing which is a non-deterministic method to optimize the packing results by the cost function. Other commonly used techniques like genetic algorithm and integer programming will also be discussed. For further optimization of the resultant packing, we use some deterministic methods like the shape curve computation and the Lagrangian Relaxation technique, to compute the best dimensions of the modules and to reduce the total chip area. These techniques will be discussed in this chapter.

4.1 General Optimization Methods

4.1.1 Simulated Annealing

Simulated annealing simulates the behavior of a complex system consisting of a large number of interacting atoms in thermal equilibrium at a certain temperate. This technique was widely applied to placement, floorplan design, channel routing and layout optimization. The method can usually produce

high quality solutions although the annealing process may take a long time. The following is a generic simulated annealing algorithm:

Algorithm: SA

Begin

S :=*Initial solution* S_0

T :=*Initial temperature* T_0

While stopping criterion is not satisfied do

Begin

While not yet in equilibrium do

Begin

S' :=*Some random neighboring solution of* S

$\Delta := Cost(S') - Cost(S)$

$Prob := min(1, e^{-\Delta/T})$

If $random(0,1) \leq Prob$ *then* $S := S'$

End;

Update T

End

Output best solution

End

By applying different function $Cost(S)$ to the simulated annealing algorithm, we can optimize different aspects of the problem. In floorplanning, $Cost(S)$ is usually equal to the cost function $A + \lambda W$ where A is the total area of the packing, W is the wirlength and λ is a constant. Area and wirelength are the two main aspects to be optimized. The cost function equation can be changed to satisfy the requirement of any specific floorplanning problem.

In simulated annealing, one important issue is to have a concise representation and description of the solution configuration. Also, the neighbors of

each solution have to be defined such that the optimal solution is reachable. Any particular annealing process will include initial temperature, moves and temperature range [WLL88]. For each representation described in the previous chapter, they have defined the set of moves in an annealing process such that every solution is reachable.

4.1.2 Genetic Algorithm

Genetic Algorithm is commonly use in solving a wide range of problems including control system, function optimization and combinatorial problems. For this algorithm, a population of solutions is maintained and is allowed to evolve through successive generations.

The solutions in the next generation can be formed by two operations:(1) crossover, i.e., merging two solutions from the current generation; (2)mutation, i.e., modifying an individual solution. The algorithm can be parallelized in order to speed up the computational time [CHMR91]. The difference between simulated annealing and genetic algorithm method is that there can be more that one solution in the solution set at a time for genetic algorithms while simulated annealing only obtain one best solution. In a genetic algorithm a population of solution is used to generate the offspring (next generation of solution).

In the paper [CHMR91], the Polish expression is used as the floorplan representation. There are four types of crossover operations in Figure 4.1 which demonstrates the crossover operations: CO_1, CO_2, CO_3, CO_4.

CO_1: It first copy the operand from parent P_1 into the corresponding positions in the offspring O. Afterwards, it copies the operators ('+', '*') from P_2, by a left-to-right scan to fill up the missing spaces in O.

CO_2: It is similar to CO_1, it copies the operators from P_1 first and fill the missing space of O with the operands in P_2.

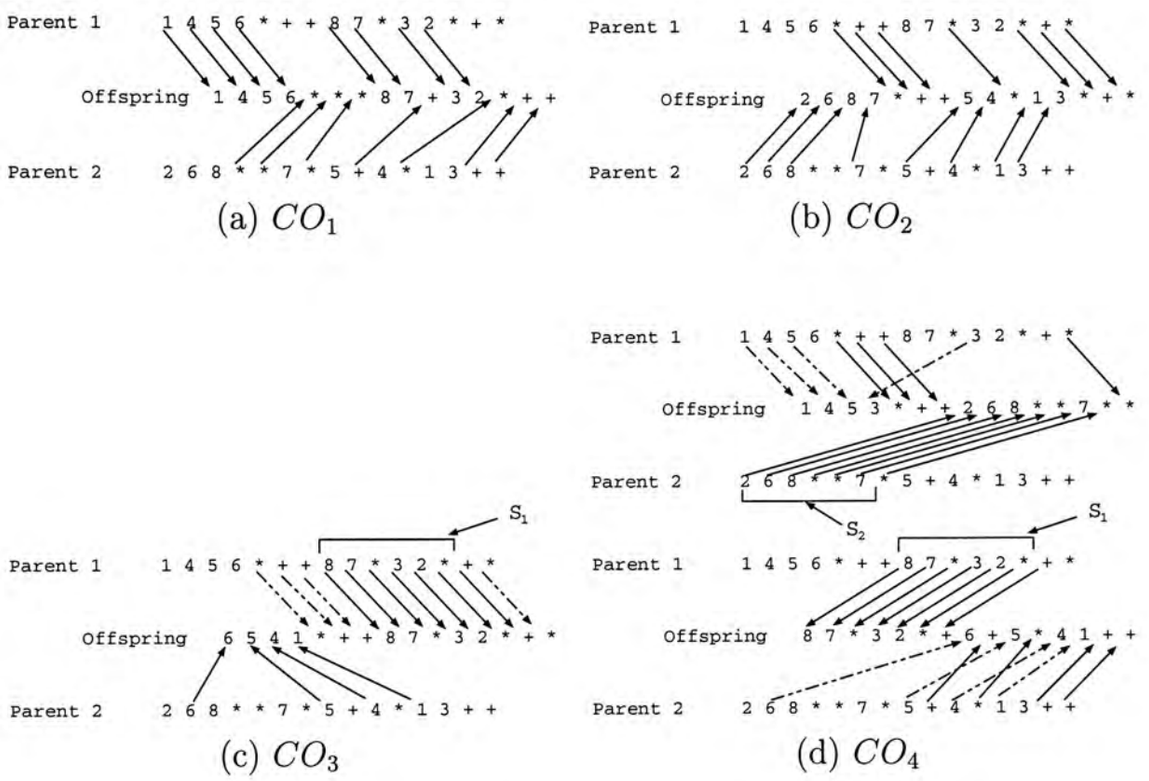


Figure 4.1: Crossover operation of the genetic algorithm in [CHMR91]

CO_3: A subtree in P_1 is selected and copied it to the correspond position of the offspring. The spaces are filled up by operators from P_1 and operands from P_2.

CO_4: It uses two parents to produce two offspring by interchanging the subtree of the parents. s_1 and s_2 of the same size are selected from parents P_1 and P_2. It fails if there is no s_1 and s_2 in P_1 and P_2. The next procedure will be similar to CO_3. The offspring O_1 will be created by copying s_2 from P_2 to the correspond position of s_1 in P_1. O_2 is obtained similarly with the role of P_1 and P_2 interchanged.

For the result shown in the paper [CHMR91], the performance is slightly better that of the simulated annealing method. Genetic algorithm gives a faster computation than the simulated annealing method since it can be implemented on distribution system. Selecting, crossing over pairs of solutions and mutating are done in parallel.

4.1.3 Integer Programming Method

An analytical method for general floorplan design was proposed by Sutanthavibul, Shargowitz and Rosen in 1991 [SSR91]. This method is based on a mixed integer programming model and a standard mathematical software which can be applied on both rigid and flexible modules.

Consider two modules i and j. Let (w_i, h_i), (w_j, h_j) be the dimensions of the modules. (x_i, y_i) and (x_j, y_j) are the coordinates of the lower left corners of the modules. Since i and j are non-overlapping, at least one of the linear inequalities shown below holds:

$$\begin{aligned} x_i + w_i &\leq x_j, && i \text{ is to the left of } j \\ x_i - w_i &\geq x_j, && i \text{ is to the right of } j \\ y_i + h_i &\leq y_j, && i \text{ is below } j \\ y_i - h_i &\geq y_j, && i \text{ is above } j \end{aligned} \tag{4.1}$$

Two variables x_{ij} and y_{ij} of value either 0 or 1 are used for each pair of blocks. Two bounding function W and H are defined where $|x_i - x_j| \leq W$ and $|y_i - y_j| \leq H$. W and H is either equal to W_{max} and H_{max} where the maximal width and height. If W_{max} and H_{max} is not given, then $W = \sum_{i=1}^{K} w_i$ and $H = \sum_{i=1}^{K} h_i$.

The modules are allowed to rotate by making use of a variable z_i of value either 0 or 1. If $z_i = 0$, module i is placed in its initial orientation. If $z_i = 1$, module i will be rotated by 90°. Then, equation 4.1.3 can be rewritten as follow:

$$\begin{aligned} x_i + z_i h_i + (1 - z_i) w_i &\leq x_j + M(x_{ij} + y_{ij}) \\ x_i - z_j h_j - (1 - z_j) w_j &\geq x_j - M(1 - x_{ij} + y_{ij}) \\ y_i + z_i w_i + (1 - z_i) h_i &\leq y_j + M(1 + x_{ij} - y_{ij}) \\ y_i - z_j w_j - (1 - z_j) h_j &\geq y_j - M(2 - x_{ij} - y_{ij}) \end{aligned} \tag{4.2}$$

$$y^* \geq y_i + (1 - zi)w_i + z_i h_i \tag{4.3}$$

where $M = max(W, H)$ and y^* is the height to be minimized. The model above does not consider flexible modules and interconnect lengths, and it requires a substantial number of variables and constraints for the optimization [SSR91].

For a flexible blocks, its width and height w_i and h_i can be varied but its aspect ratio is bounded and its area A_i remain fixed such that $A_i = w_i h_i$ This function can be linearized about the point of maximum allowable width w_{max} by applying the first two member of its Taylor series [SSR91].

Δw_i is a continuous variable for each flexible module. For a flexible module i and a rigid module j, the set of inequalities 4.1.3 can be rewritten as follows:

$$\begin{aligned} x_i + w_{i,max} - \Delta w_i \leq x_j, &\quad i \text{ is to the left of } j \\ x_i - w_j \geq x_j, &\quad i \text{ is to the right of } j \\ y_i + h_{i0} + \Delta w_i \lambda_i \leq y_j, &\quad i \text{ is below } j \\ y_i - h_i \geq y_j, &\quad i \text{ is above } j \end{aligned} \tag{4.4}$$

where $h_i 0 = \frac{S_i}{w_{i,max}}$, $\lambda_i = \frac{S_i}{w_{i,max}^2}$. The constraints on routablity can also be formulated. A chip is routable if the length of the available routing tracks is 1.5 to 2.0 times longer than the actual length of the required interconnections [SSR91]. This allows the constraint to be formulated as inequalities.

A greedy procedure used to solve the floorplanning problem is given below where k is the total number of modules.

Procedure FloorplanDesign

Begin

Select a group of m modules as a seed

Formulate a system of linear constraints for these unpositioned modules

Call an integer programming procedure to obtain the first partial floorplan

While ($m \leq k$) do

Begin

1. *Select a new group of e modules based on time consideration the connectivity to the already fixed modules in the partial floorplan*
2. *Find a set of d covering rectangles for the partial floorplans, where* $d \leq m$
3. *Formulate a system of linear constraints for d covering rectangles and e unpositioned modules*
4. *Call an integer programming procedure to obtain a new partial floorplan*

End

Perform global routing

Adjust floorplan

End

4.2 Shape Optimization

4.2.1 Shape Curve

Shape Curve computation is used in Polish expression. The total area of a floorplan can be computed and optimized by using shape curve as shown in Figure 4.2. The values on the curve represent the possible dimension of a module where x is the width and y is the height. For the curve in Figure 4.2(a), there are only two possible dimensions: $\{a_1, b_1\}$ and $\{a_2, b_2\}$. The shape curve of a module can also be a smooth curve and the module will be very flexible in this case as shown in Figure 4.2(b).

Shape curves can be used to compute the packing area optimally. Let Γ and Λ be two shape curves. The new shape curve Ψ that represent the supermodule (a sub-floorplan that consists of more that one basic modules) by combining the two modules can be computed from Γ and Λ.

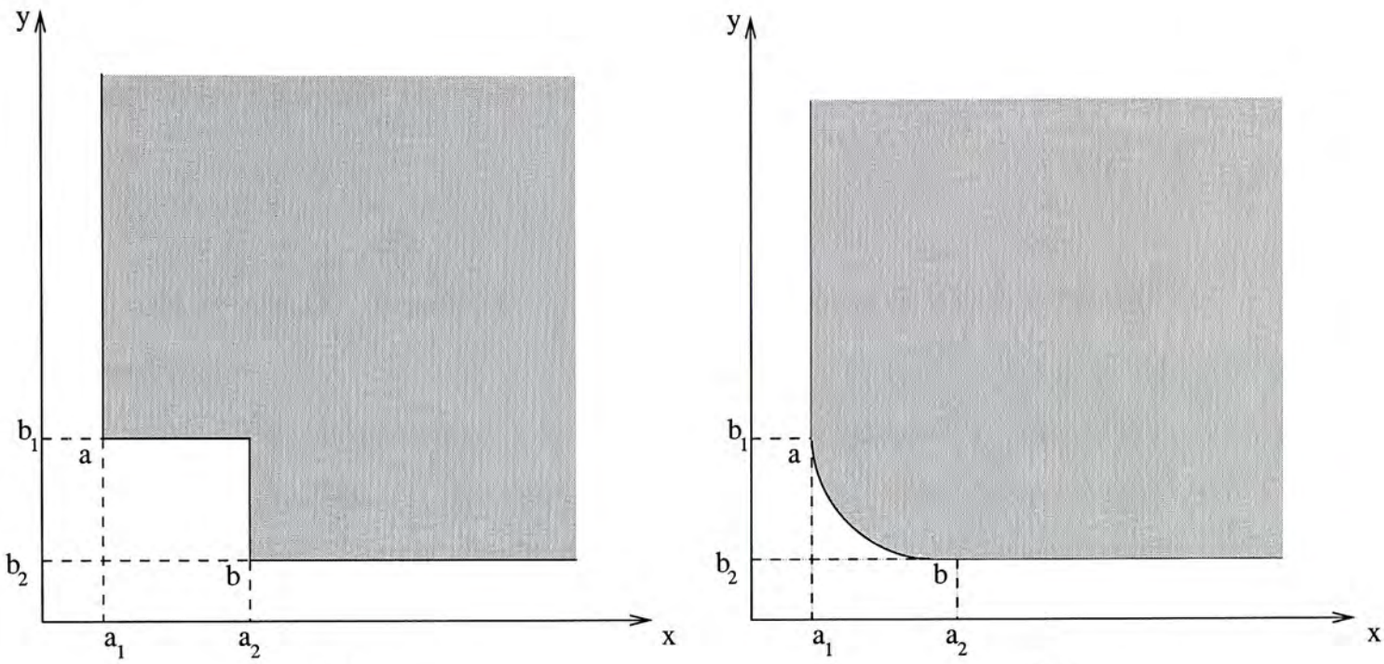


Figure 4.2: Shape curve of a rectangular module

If the cut is vertical, $\Psi = \Gamma + \Lambda$ such that

$$\Psi : \{(u, v + w) | (u, v) \in \Gamma \text{ and } (u, w) \in \Lambda\}$$

Similarly, if the cut is horizontal, $\Psi = \Gamma * \Lambda$ such that

$$\Psi : \{(u + v, w) | (u, w) \in \Gamma \text{ and } (v, w) \in \Lambda\}$$

4.2.2 Lagrangian Relaxation

We are given n modules $\{M_1, M_2, \ldots, M_n\}$ where each modules M_i has an area A_i. Let w_i and h_i be the width and height of the modules M_i and r_i and s_i be the minimum and maximum aspect ratio of M_i respectively. We denote the minimum and maximum width of M_i by $L_i = \sqrt{A_i/r_i}$ and $U_i = \sqrt{A_i/s_i}$ respectively.

As mention above, we can construct a horizontal constraint graph and a vertical constraint graph from any sequence pair. For every edge $e(i, j)$ in G_h, we have the following constraint:

$$x_i + w_i \leq x_j$$

where x_i and x_j are the x positions of the low left corners of module M_i and M_j respectively. Similarly, for every edge $e(i,j)$ in G_v , we have the following constraint:

$$y_i + \frac{A_i}{w_i} \leq y_j$$

where y_i and y_j are the y positions of the low left corners of module M_i and M_j respectively.

A dummy vertex labeled $n+1$ is added to both G_h and G_v. For every vertex i without any out-going edge in both G_v and G_h, an edge $e(i, n+1)$ with zero weight is added. Thus, the problem can be formulated as a geometric program(PP):

$$\begin{array}{lll} \text{Minimize} & x_{n+1}y_{n+1} & \\ \text{Subject to} & x_i + w_i \leq x_j & \forall e(i,j) \in G_h \\ & y_i + \frac{A_i}{w_i} \leq y_i & \forall e(i,j) \in G_v \\ & L_i \leq w_i \leq U_i & \forall 1 \leq i \leq n \end{array}$$

Let $\lambda_{i,j}$ denote the multiplier which is used for Lagrangian relaxation for the constraint $x_i + w_i \leq x_j$ and $\mu_{i,j}$ denote the multipliers for the constraint $y_i + \frac{A_i}{w_i} \leq y_i$. Let $\overrightarrow{\lambda}$ and $\overrightarrow{\mu}$ be the vectors of all the Lagrangian multipliers introduced into the constraints. Then, we can formulate the Lagrangian relaxation subproblem associated with the multiplier $\overrightarrow{\lambda}$ and $\overrightarrow{\mu}$, denoted by $LRS/(\overrightarrow{\lambda}, \overrightarrow{\mu})$, as shown below:

$$\begin{array}{ll} \text{Minimize} & x_{n+1}y_{n+1} + \\ & \sum_{e(i,j)\in G_h} \lambda_{i,j}(x_i + w_i - x_j) + \\ & \sum_{e(i,j)\in G_v} \mu_{i,j}(y_i + \frac{A_i}{w_i} - y_i) \\ \text{Subject to} & L_i \leq w_i \leq U_i \quad \forall 1 \leq i \leq n \end{array}$$

A corresponding Lagrangian dual problem LDP of PP can be formulated for the optimal solution , denoted by $Q(\overrightarrow{\lambda}, \overrightarrow{\mu})$, of the subproblem $LRS/(\overrightarrow{\lambda}, \overrightarrow{\mu})$ for a particular pair of $\overrightarrow{\lambda}$ and $\overrightarrow{\mu}$ as shown below:

$$\begin{aligned} &\text{Maximum} \quad Q(\overrightarrow{\lambda}, \overrightarrow{\mu}) \\ &\text{Subject to} \quad \overrightarrow{\lambda} \geq 0 \text{ and } \overrightarrow{\mu} \geq 0 \end{aligned}$$

Since the original problem PP is convex, we can imply that if $(\overrightarrow{\lambda}, \overrightarrow{\mu})$ is the optimal solution to LDP. The corresponding solution of $LRS/(\overrightarrow{\lambda}, \overrightarrow{\mu})$ is the optimal solution to PP.

Considering the Lagrangian ζ of PP, we have

$$\begin{aligned} \zeta =& x_{n+1}y_{n+1} + \sum_{e(i,j)\in G_h} \lambda_{i,j}(x_i + w_i - x_j) + \sum_{e(i,j)\in G_v} \mu_{i,j}(y_i + \frac{A_i}{w_i} - y_j) + \\ & \sum_{1\leq i\leq n} u_i(L_i - w_i) + \sum_{1\leq i\leq n} v_i(w_i - U_i) \end{aligned}$$

Using the Kuhn-Tucker conditions, we can imply the $\frac{\partial \zeta}{\partial x_i} = 0$ and $\frac{\partial \zeta}{\partial y_i} = 0$ for all $1 \leq i \leq n+1$ at the optimal solution of PP. Taking the partial derivatives of ζ, we have the following optimality conditions:

$$\sum_{e(i,j)\in G_h} \lambda_{i,j} = \sum_{e(j,i)\in G_h} \lambda_{j,i}$$

$$\sum_{e(i,j)\in G_v} \mu_{i,j} = \sum_{e(j,i)\in G_v} \mu_{j,i}$$

LDP is convex due to PP, so the optimal $(\overrightarrow{\lambda}, \overrightarrow{\mu})$ can be found by sub-gradient optimization. However, the solution may not satisfy the optimality condition and we have to project a new pair of $(\overrightarrow{\lambda}, \overrightarrow{\mu})$ to the nearest point $(\overrightarrow{\lambda^*}, \overrightarrow{\mu^*})$ that satisfy the optimality conditions. These steps continue until the solution converges.

Chapter 5

Literature Review on Interconnect Driven Floorplanning

This chapter will have a review of the papers related to the interconnect driven floorplanning. They include placement constraint, timing analysis driven, buffer block planning and congestion control. Placement constraints are useful that allow some modules are constrained to be placed in some specified positions to reduce the interconnect cost and to improve the circuit performance. Buffer block planning optimize the delay by inserting buffers within the deadspace of a packing. Instead of minimizing the distance of wiring, we also conserve the problem of congestion in the floorplanning stage.

5.1 Placement Constraint in Floorplan Design

5.1.1 Boundary Constraints

Boundary Constraints in floorplan design have been applied on slicing floorplan [YW99a], sequence pair [LLWW01, TW01] and corner block list [MDH+01].

Boundary Constraint is a constraint in which some modules required to be

placed along placed at the certain boundary of the packing. This allows the modules to be connected to the I/O pads much more easier and thus reduces the interconnect cost. For this problem, the modules are divided in to five sets, M^F, M^T, M^L, M^R and M^B. M^F is a set of modules which can be placed freely at any position of the packing. The other four sets of modules are under boundary constraints such that the modules in M^T, M^L, M^R and M^B have to be placed along the top, left, right and bottom boundary of the packing respectively. Figure 5.1 illustrates the boundary constraint. The modules are assigned to each of the five sets according to its I/O pin connection.

The papers [YW99a] and [MDH+01] use a similar technique to solve the problem in different floorplan representation. Methods are devised to find the modules which are located at the boundaries of the packing by just looking at the representation. In this way, they are able to check if the boundary constraints are satisfied. If some constraints are not satisfied, the modules under boundary constraint will be swapped with those modules lying at the boundary positions. There is though a problem this method which is, it may happen that the constraint cannot be satisfied in a packing since there are only a limit number of boundary positions for each packing. Solving this problem, a penalty term is introduced to the cost function to penalize violating the constraints during in the annealing process.

The paper [LLWW01] using a different method to solve the same problem in floorplan design. The method is based on sequence pair representation. A set of rules are devised from the properties of sequence pair. For example, if module x is in M_L and module y is in M_B, they have to obey the lb-property in the sequence pair(Γ_+, Γ_-). The lb-property is that position of y in Γ_+ should be greater than that of x in Γ_+. In this way, all the constraints can be satisfied in every packing after apply the rules and a feasible floorplan can always be obtained.

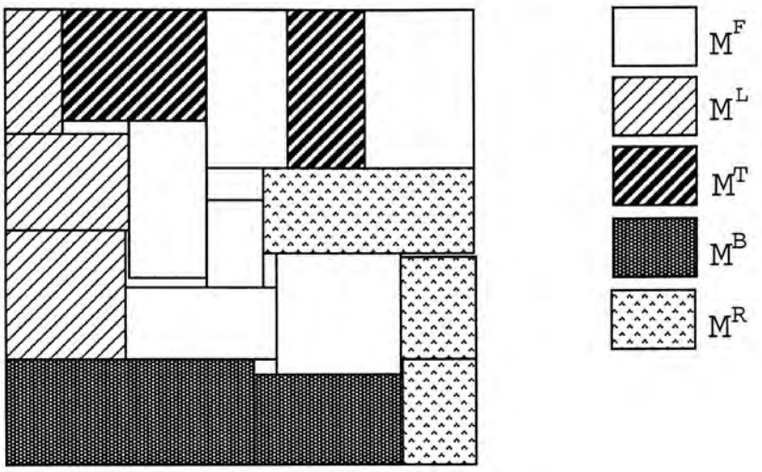


Figure 5.1: The Boundary Constraints

5.1.2 Pre-placed Constraints

Pre-placed constraints in floorplan design have been applied on slicing floorplans [YW98] and non-slicing floorplans [FMK97, MK98, TW01]

Pre-place constraints is a constraint in which some modules are required to be placed at certain positions of the packing. It is important that a floorplanner can handle pre-placed modules since some modules may have their on the positions fixed in some practical problems. Figure 5.2 illustrates the pre-placed constraints.

In the paper [YW98], a reference point $ref(X)$ is associated with each supermodule X if it contains at least one pre-placed module. All possible positions of $ref(X)$ within the supermodule X can be kept using four numbers, $bottom(X), top(X), left(X)$ and $right(X)$ (Figure 5.3) and the reference point of a supermodule can be computed from the reference points of its children. Figure 5.3 illustrates the steps of updating the locations of the reference point. Simulated annealing is used and a penalty term is introduced into the cost function such that the pre-placed modules will be pulled to the desired positions.

The paper [MK98] use an efficient method to solve the problem based on

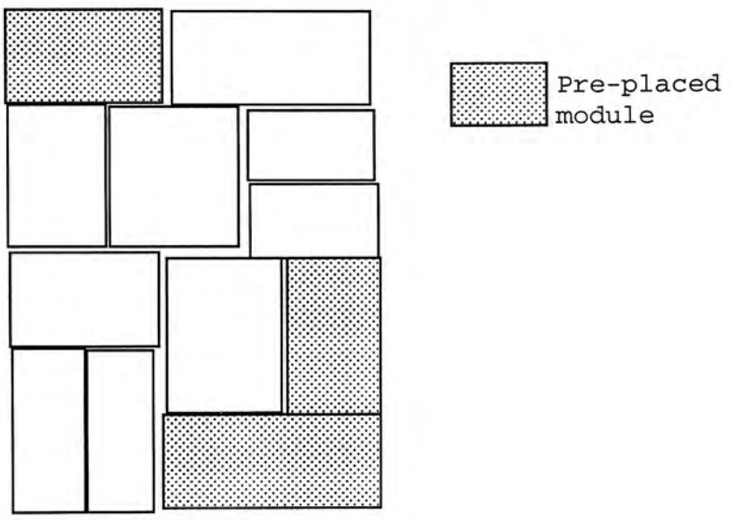


Figure 5.2: The Pre-placed Constraints

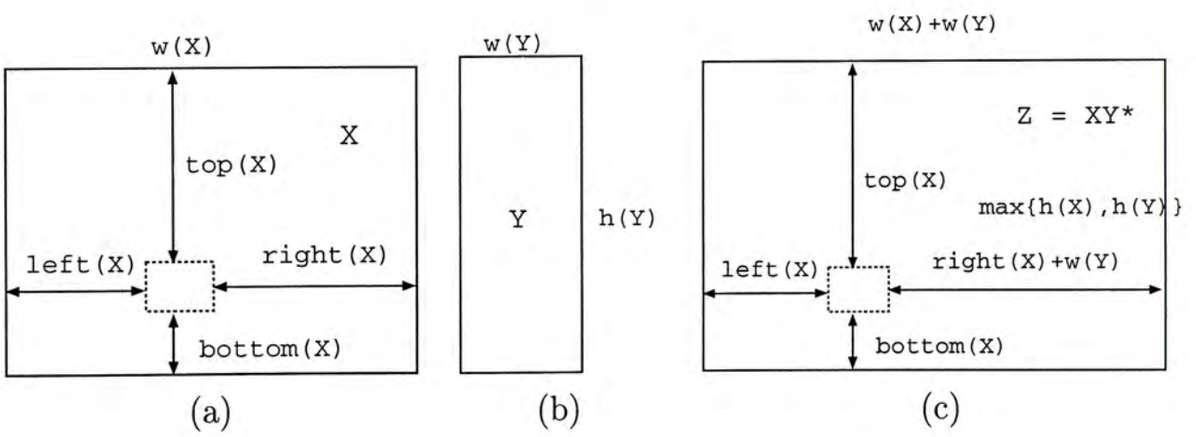


Figure 5.3: Reference point inherit for the supermodule

the sequence pair representation. An additional edge of a weight equal to the pre-assigned coordinate is added from the source node to the pre-placed modules in the horizontal and vertical constraint graph respectively. The resultant placement is called propped-realization. The placement may still not be a feasible one since the additional edges can only constraint the pre-placed module to be place at least as far as the pre-assigned coordinates. An adaptation procedure is devised to test the placement of a module violating the constraints and fixed them.

5.1.3 Range Constraints

Range Constraints in floorplan design have been applied on slicing floorplans [YW99b] and non-slicing floorplans [TW01].

Given a rectangular region $R_1 = \{(x, y) | x_1 \leq x \leq x_2, y_1 \leq y \leq y_2\}$, some modules with range constraint are required to be placed within R_1 in the final packing. The range constraint is more general than the pre-placed constraint since very pre-placed constraint can be specified as a range constraint. It is useful to consider range constraint in the practical floorplanning problem. Figure 5.4 shows an example of a feasible floorplan in which a module is with range constraint is placed in the region R_1.

The paper [YW99b] addresses the problem of range constraint using a similar technique for pre-placed constraint. There are four variables for each supermodule X ($right(X), left(X), top(X)$ and $bottom(X)$) that indicates in which X should be placed. For example, if a module Y with width w and height h is constrained to be placed within the region $\{(x, y) \mid x_1 \leq x \leq x_2, y_1 \leq y \leq y_2\}$, $right(X) = x_1 + w, left(X) = x_2 - w, top(X) = y_1 + h$ and $bottom(X) = y_2 - h$. The range for a supermodule can be inherited from its children. After the shape curve computation step, it can be determined whether a certain module is placed within the desired range constraint. A

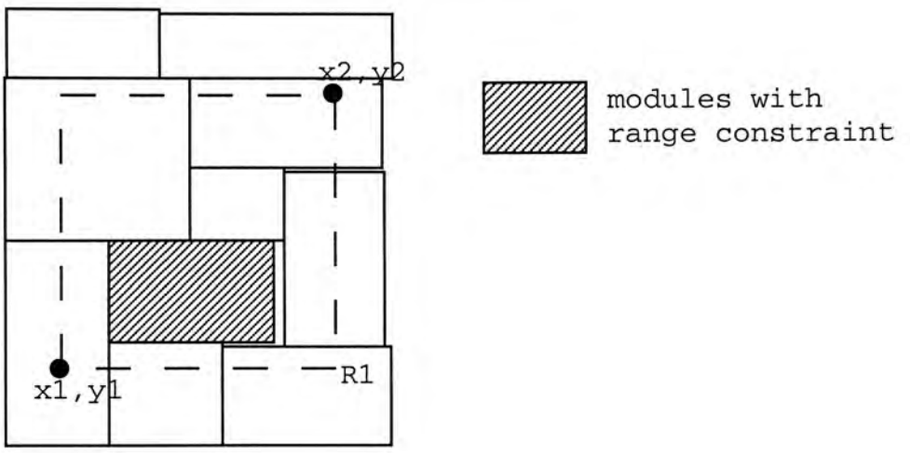


Figure 5.4: The Range Constraints

penalty term is introduced into the cost function so that the packing will converge to satisfy the range constraint finally.

5.1.4 Symmetry Constraints

Symmetry Constraints in floorplan design have been applied using O-tree representation [PBLC00].

Symmetry Constraints is a constraint in which some modules are required to be placed symmetrically along an axis. Two terms, symmetry pair and symmetry groups, are introduced to define the symmetry constraints clearly in the floorplan design problem. Symmetry pair is a pair of modules of the same dimensions and have to be placed symmetrically with respect to an axis. Symmetry group is a set of symmetry pairs which share the same axis. Given a symmetry group, $S = \{(M_{a_1}, M_{b_1}), (M_{a_2}, M_{b_2}), \ldots, (M_{a_k}, M_{b_k})\}$ where (M_{a_i}, M_{b_i}) is a symmetry pair for $i = 1, \ldots, k$, then for the case of horizontal symmetry

$$x_{a_i} = x_{b_i}$$

$$y_{a_i} + y_{b_i} + h_{a_i} = 2y_s$$

where y_s is the position of the common symmetry axis, and x_{a_i}, y_{a_i} and h_{a_i} are the x-position, y-position and height of module M_{a_i} respectively. Figure 5.5

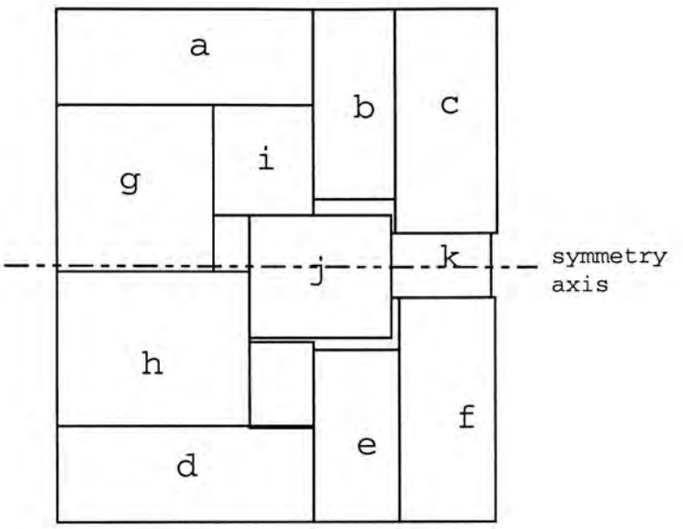


Figure 5.5: The Symmetry Constraints

shows an example that illustrates the symmetry constraint. In the example, module (a, b), (c, d) and (e, f) is the symmetry group with symmetry constraint in the packing.

Symmetry Constraints is useful to handle the analog circuits. It is because analog circuits usually use differential architectures based on electrically symmetric networks. In addition, symmetry is commonly used balance thermal effects and to match interconnect parasitics and device parameters.

5.2 Timing Analysis Method

In the paper [SYTB95], the authors described the implementation of a floorplanner that considers the timing information. The algorithm is divided into two phases. Phase I focuses on timing optimization while phase II performs floorplan refinement to allow the aspect ratios of the modules to be changed. However, there are some disadvantages. The floorplanner uses the timing data from another timing analysis program and includes the results into a fitness function as follows:

$$Fitness(i) = \frac{A^*(i)}{A_{max}} \times W_A + \frac{T^*(i)}{T_{max}} \times W_T + \frac{L^*(i)}{L_{max}} \times W_L$$

where A^*, T^*, L^* are the area, clock speed and wirelength of the circuit respectively. A_{max}, T_{max}, $Lmax$ are the maximum values in the given population. W_A, W_T, W_L are the user specified relative weight of different aspects.

This equation including the area , timing and wirelength factors. However, using another timing analysis program is time consuming. The method is based on an iterative method and the timing value is included only in the cost function. It does not have any scheme that directly reduce the delay of the system during the process.

There is another paper [YSAF95] which described a similar approach. This paper combines the force directed approach and the constraint graph approach. It is also consisted of two phases. In Phase I, a timing and connectivity driven topological arrangement is acquired using a force directed approach. Then, the topological arrangement is transformed into a legal floorplan in Phase II. The floorplan is obtained from a greedy approach is such a way that the modules are put into the floorplan one by one. The growth of the chip will be controlled by a given aspect ratio. A gain function is used to an unplaced module into the floorplan in each step. The gain function of module i is computed as:

$$cost_i = \frac{CLOCK - u_i}{CLOCK}$$

$$G_i = \sum_{j \in F_k} c_{ij} + \beta \sum_{i \in B_j, j \in N_k} (1 - p_j) cost_j$$

where $CLOCK$ is the clock period and u_i is the delay bound on net $n_i \in N$ computed from the timing analyzer. c_{ij} is the connectivity between modules i and module j. The approach also use the timing analyzer to analyze circuit performance.

In the paper [VNLG95], another timing driven floorplanner is introduced. The timing analyzing process is also done by a timing analyzer. It is not

efficient since the timing information have to be analysis in each iteration and thus the total computational time for the floorplanning step is increased

5.3 Buffer Block Planning and Congestion Control

Most of the interconnect optimization like buffer insertion/sizing, wire sizing, etc., are done on the layout after placement. It will be more effective if interconnect optimization can be done in the earlier stage such as the floorplanning stage.

5.3.1 Buffer Block Planning

Buffer insertion is a very effective and useful method to minimize the delay of a wire. It is an active devices to break original long interconnects into shorter ones such that the overall delay can be reduced. The first publication about buffer block planning is [CKP99]. The concept of feasible region(FR) was introduced. The FR for a buffer B is defined to be the maximum region where B can be located such that by inserting B into any location in the region, the delay constraint can be satisfied assuming that other buffers are inserted correctly. Figure 5.6 shows the feasible region of k buffers on a wire. For a long interconnect with k buffers inserted, the feasible region for the i-th buffer $(i \leq k)$ is $x_i \in [x_{min_i}, x_{max_i}]$ where x_{min} and x_{max} can be computed using Elmore delay model:

$$x_{min_i} = MAX(0, \frac{K_2' - \sqrt{K_2'^2 - 4K_1'K_3'}}{2K_1'})$$

$$x_{max_i} = MIN(l, \frac{K_2' - \sqrt{K_2'^2 - 4K_1'K_3'}}{2K_1'})$$

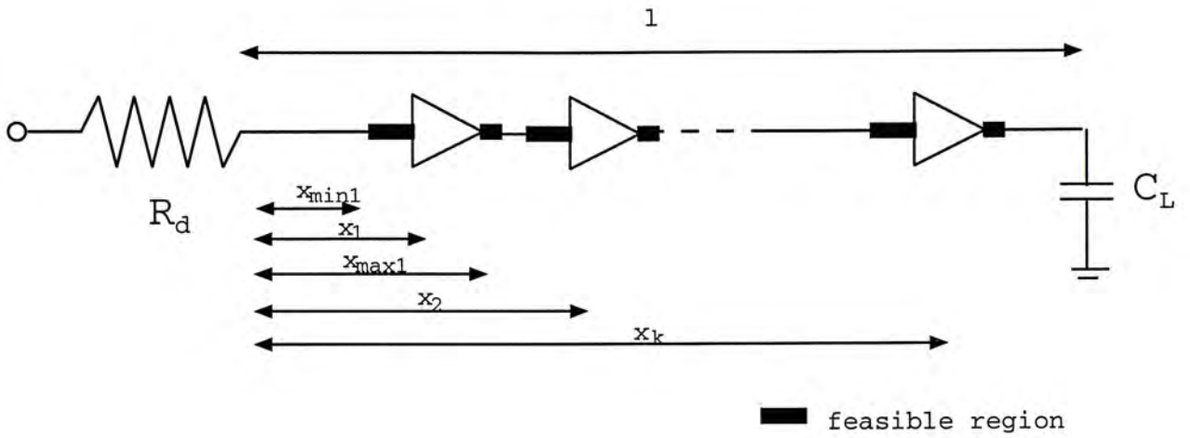


Figure 5.6: Feasible regions for inserting k buffers.

where

$$K_1'(k,i) = \frac{(k+1)rc}{2i(k-i+1)}$$

$$K_2'(k,i) = \frac{(R_b - R_d)c}{i} + \frac{r(C_L - C_b) + rcl}{k-i+1}$$

$$K_3'(k,i) = kT_b - T_{req} + [R_d + (i-1)R_b + \frac{(k-i)rl}{k-i+1}]C_b + R_b[(k-1)C_b + C_L + cl]$$
$$+ \frac{rcl^2}{2(k-i+1)} + rlC_L - \frac{(i-1)c(R_b - R_d)^2}{2ir} - \frac{(k-i)r(C_b - C_L)^2}{2(k-i+1)c}$$

r, c are the unit length wire resistance and capacitance respectively, T_b is the intrinsic delay of the buffer, C_b, R_b is the input capacitance and output resistance of the buffer, and R_d, C_L and l are defined in Figure 5.6.

Besides, the minimum number of buffers to meet the delay constraint T_{req} for an interconnect of length l is devised as

$$k_{min} = \lceil \frac{K_5 - \sqrt{K_5^2 - 4K_4K_6}}{2K_4} \rceil$$

where

$$K_4 = R_bC_b + T_b$$

$$K_5 = T_{req} + \frac{r}{c}(C_b - C_L)^2 + \frac{c}{r}(R_b - R_d)^2 - (rC_b + cR_b)l - T_b - R_dC_b - R_bC_L$$

$$K_6 = \frac{1}{2}rcl^2 + (rC_L + cR_d)l - T_{req}$$

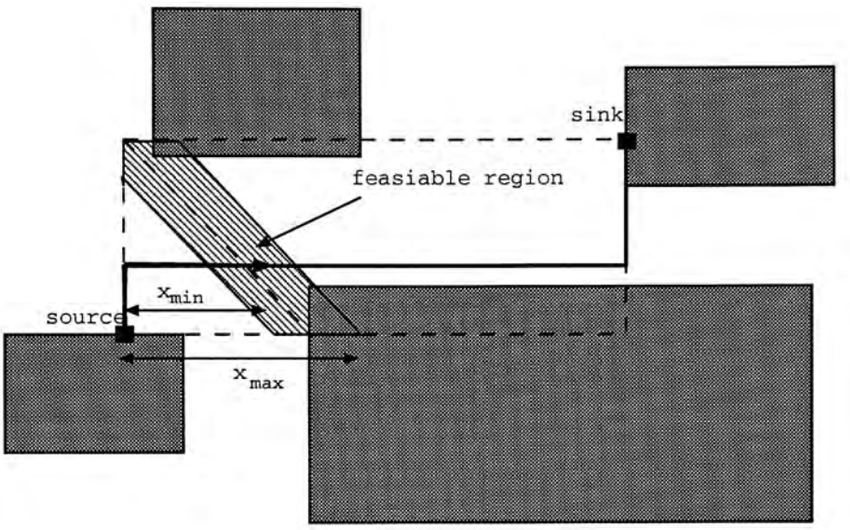


Figure 5.7: 2-D feasible regions with existing placed modules.

In a floorplan, a feasible region is bounded by two parallel lines with Manhattan distances from the source to be x_{min_i} and x_{max_i} respectively as shown in Figure 5.7. For those parts of a feasible region which overlap with a module, they have to be removed from the feasible region.

The buffer block planning is a hard problem in which the number, size and positions of the buffers have to be determined. Besides, if there is not enough space for the buffers to insert in order to satisfy the timing requirement, the packing have to be expanded. The buffer block planning algorithm in [CKP99] have several limitations. First, it only makes use of the deadspace in the packing. The formulae are simplified to reduce the runtime in handling thousands of nets. In addition, only one buffer is inserted for a single net.

Their buffer block planning algorithm first builds the horizontal and vertical polar graphs. Each channel is divided into a set of rectangular tiles for better manipulation and representation of buffer blocks. Finally, the algorithm will insert as many buffers as possible by selecting the most suitable tiles for them.

The buffer block planning algorithm in [CKP99]is a direct method to address the problem. The paper [TW00] have presented a polynomial time optimal algorithm based on network flows for solving the problem of inserting the

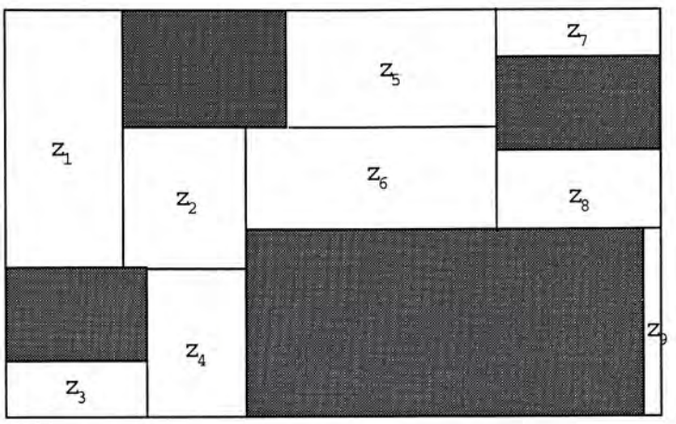


Figure 5.8: Buffer zones of a packing

maximum number of buffers into the free space between the modules.

Since the feasible region can overlap with the modules, buffer zones are defined such that they are the dead spaces in which the buffer can be inserted. Given a placement, the buffer zones are obtained by cutting the free space between the modules into rectangular zones with different costs according to the routing congestion and other factors. Figure 5.8 shows buffer zones $z_1, z_2, \ldots, z_9$ obtained by cutting the deadspace of the placement.

Given a set of buffers $B = \{b_1, b_2, \ldots, b_n\}$ and the corresponding feasible regions $F = \{f_1, f_2, \ldots, f_n\}$, and buffer zone $Z = \{z_1, z_2, \ldots, z_m\}$, a network flow graph is constructed. By finding the min-cut of the network flow graph, the maximum number of buffers with the minimum total insertion cost can be obtained.

A term buffer room is introduced. Given a set of FRs $F = \{f_1, f_2, \ldots, f_n\}$, buffer rooms are disjoint regions bounded by the boundaries of FRs. Let $R = \{r_1, r_2, \ldots, r_w\}$ denote the buffer rooms, then $r_i \cap r_j = \varnothing$ for $i \neq j$.

In Figure 5.9, it shows a problem containing two buffers, two nets and seven buffer zones. The corresponding network flow formulation is given in Figure 5.10. The vertices of the network flow graph consist of a source node, a destination node, B, R and Z. If a buffer room belongs to a certain buffer,

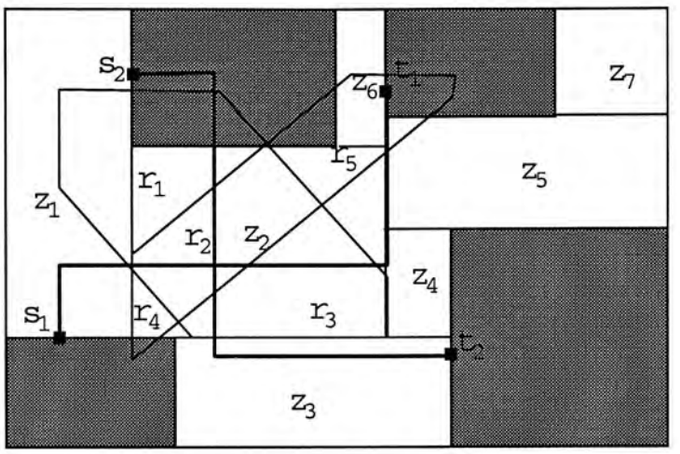


Figure 5.9: A BP problem containing two buffers, two nets and seven buffer zones.

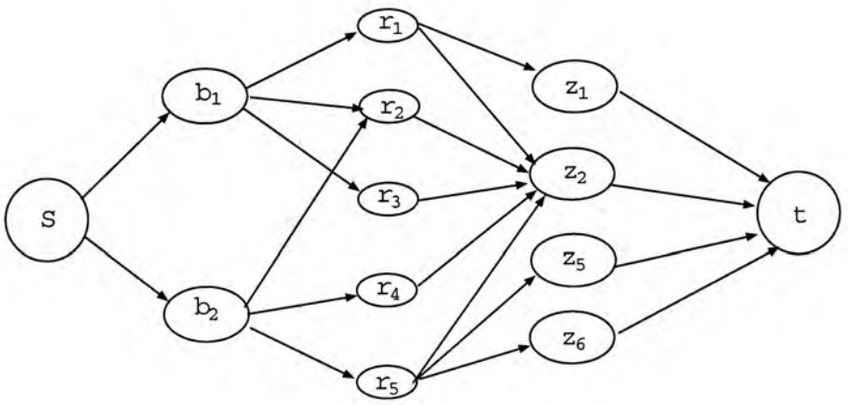


Figure 5.10: The corresponding network flow graph of problem in 5.9

there will be a directed edge from the buffer to that buffer room. If a buffer room is overlapped with a buffer zone, there will be a directed edge from that buffer room to that buffer zone. A min-cost maximum flow in the graph corresponds to a buffer insertion solution to the buffer planning problem with the maximum number of buffers and minimum total insertion cost. If the size of the max-flow is n, then the buffer planning problem is feasible that all the buffers can be inserted into the deadspace of the packing.

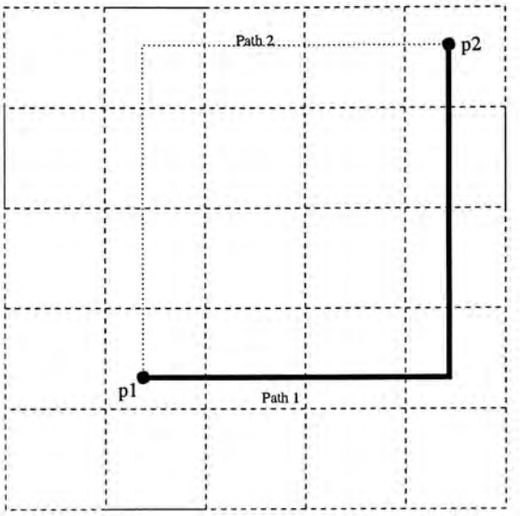


Figure 5.11: L-shaped routing

5.3.2 Congestion Control

A new wirelength estimation method is used in [CZY+99]. Simple geometry routing is used to estimate the routing length between two pins. There are two types of routing to obtain the minimum wire length: L-shaped and Z-shaped(shown in Figure 5.11 and 5.12). Z-shaped routing gives a more accurate estimation than L-shaped routing. Also, the cost function is $\alpha A + \beta W + \gamma OF$ where OF is the sum of the square of overflow in each grid. There are three stages in estimating the wirelength W. (1) half- perimeter; (2) L-shaped global routing and (3) Z-shaped global routing. In each stage, the value of W will be estimated by different methods.

The cost function used is $\alpha A+\beta W+\gamma OF$ where OF is the sum of the square of overflow in routings. The values of OF is zero in stage 1 but it is computed by applying simple geometry routing to estimate the congestion/routablity of bin boundaries. This paper described a more efficient approach in estimating the timing information and considering the congestion during the floorplanning stage. Since it does not use any external timing analyzer and the computational time is reduced.

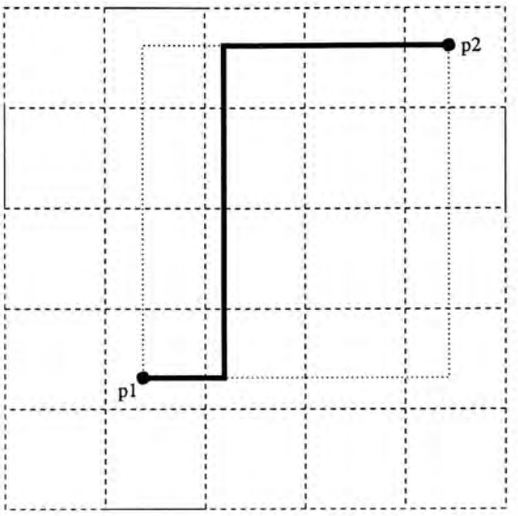


Figure 5.12: Z-shaped routing

The paper [SSK00] takes routing congestion into consideration in the buffer planning problem. The congestion model employed is a two dimensional rectangular grid based probabilistic map which assume two-bend routing for each segment. $C_h(i,j)$ and $C_v(i,j)$ is the expected number of horizontal and vertical routes passing through a routing tile $tile(i,j)$ respectively. $\delta C_h(i,j)$ and $\delta C_v(i,j)$ are defined as the probability of a horizontal and vertical route passing through $tile(i,j)$. The following equations compute the probability matrix for a route from $(0,0)$ to (m,n).

Tile	$\delta C_h(i,j)$	$\delta C_v(i,j)$
$0 < i < m, 0 < j < n$	$\frac{1}{n+m+2}$	$\frac{1}{n+m+2}$
$0 < i \leq m, j = 0$	$\frac{n-i}{n+m+2}$	$\frac{1}{n+m+2}$
$i = 0, 0 < j \leq n$	$\frac{1}{n+m+2}$	$\frac{m-j}{n+m+2}$
$i = 0, j = 0$	1	1
$i = m, j = n$	1	1

A subnet is defined as the segment of a net between two consecutive buffers.

For each $tile(i, j)$, the congestion matrices C_h and C_v are computed as follows:

$$C_h(i,j) = \sum_{\forall subnets} \delta C_h(i,j)$$

$$C_v(i,j) = \sum_{\forall subnets} \delta C_v(i,j)$$

Therefore, the congestion can be considered by introducing an element into the cost function using the congestion matrices. The congestion matrices will be updated after each buffer planning.

Chapter 6

Clustering Constraint in Floorplan Design

This chapter presents the work on clustering constraints in floorplan design. Clustering Constraints is a constraint in which some modules are required to be placed geometrically adjacent to each other to form a cluster in the packing. In this chapter, a linear time algorithm is presented to locate all the neighbors of a module in the packing by just scanning the Polish expression once.

6.1 Problem Definition

Clustering Constraint is considered in floorplan design. Given a set of modules Φ and a subset of modules $\Delta \subseteq \Phi$, we want to pack the modules in Φ such that the modules in Δ will be geometrically adjacent to each other. Figure 6.1 shows an example of the clustering constraint. Modules E, F and H are the subset of modules to be clustered and they have to be placed adjacent to each other in the final packing. The floorplanning problem with clustering constraints is defined as follows:

> ***Problem FP/CC*** *Given a set of* n *modules* $\Phi = \{m_1, m_2, \ldots, m_n\}$ *and* $m_i = (A_i, r_i, s_i)$ *for* $i = 1, \ldots n$ *where* A_i *is the area of modules,*

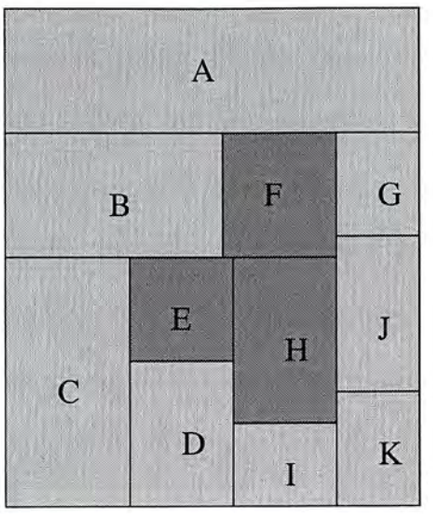


Figure 6.1: An illustration of the clustering constraint

and r_i and s_i are the minimum and maximum aspect ratio of modules i respectively. Let Δ be a subset of modules in Φ. The goal is to pack the modules in Φ to minimize the total area and interconnect cost such that the following three conditions are satisfied.

1. *Every M_i in Δ should be geometrically adjacent to at least one $M_k \in \Delta$ where $k \neq i$.*
2. *Each module satisfies its area and aspect ratio constraint.*
3. *The aspect ratio of the whole packing is within a give range [r,s].*

6.2 Overview

We consider clustering constraint in slicing floorplan. One method to solve this problem is by adding a optimizing term which is the center-to-center distance between the cluster modules to the cost function of the annealing process. Experimental is done, however the result is poor and the constraints will usually be violated in the final packing. A better approach will be introduced in which

clusters are maintained throughout the annealing process. In each iteration, we try to give the feasible packing by fixing the violation as much as possible.

A method is devised to locate all neighbors of a target module. A target module M_t is picked from the constraint set Δ. A set of M_t's neighboring modules Π_t is obtained by this method. For each $M_i \in \Pi_t$, if $M_i \notin \Delta$, we will swap M_i with M_j where $M_j \in \Delta \setminus \Pi_t$. This algorithm is able to maintain the clustering constraint throughout the annealing process.

An overview of the algorithm is given as follows:

Main Program

Begin

While $T \geq$ *threshold do*

Begin

Move by either M1, M2 or M3

Call procedure Clustering

Compute Cost

If Cost is reduced

Accept the move

Else

$Prob = min(1, e^{-\Delta_c/T})$

where Δ_c = *change of cost.*

If $random(0,1) \leq Prob$ *then*

Accept the move

Else

Reject the move

Update T

End

End

We have used simulated annealing to optimize the packing. After a move of the solution, the procedure Clustering is called. The procedure Clustering consists of two main steps. The first step is to find the neighboring modules of a target module. The second step is to perform swapping such that the packing will satisfy the clustering constraints as much as possible. The packing is then evaluated. A packing with better cost is accepted, while the acceptance of a worse one is dependent on the current temperature of the annealing processing.

6.3 Locating Neighboring Modules

An algorithm is devised to locate all neighboring modules of a target module in a normalized Polish expression. Note that we can locate the neighbors in linear time by just looking at the Polish expression once and no real packing is needed.

In each iteration, a target module M_t is selected randomly from Δ. A neighboring set Π_t is found such that M_t is surrounded by the modules in Π_t in the packing. An example is shown in Figure 6.2. In this example $M_t = F$ and $\Pi_t = \{A, B, C, D, E, G, H\}$. Note that the modules found (e.g. D and E) may not be adjacent to M_t.

For each module M_i in the slicing floorplan, M_i is surrounded by four cuts which correspond to four operators in the normalized Polish expression. If those four operators are found in the Polish expression, the neighboring structure can be located and Π_t can be found.

For a Polish expression $\alpha = \alpha_1\alpha_2 \ldots \alpha_n$, we define a *valid sub-expression* $\beta = \alpha_k\alpha_{k+1} \ldots \alpha_{k+m}$ where $k \geq 1$ and $n \geq k + m$ as a sub-expression in α such that α_k must be an operand and the number of operands in β is equal to the number of operators plus one. A valid sub-expression indeed represents a sub-tree in the whole slicing tree and also represents a supermodule in the packing.

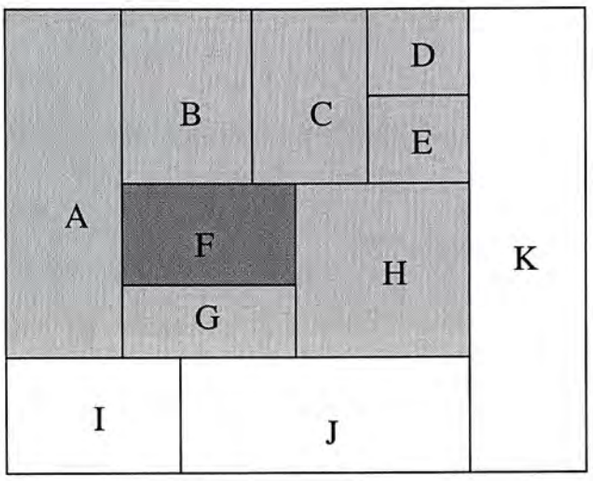


Figure 6.2: The Illustration of the neighboring structure

The two operators correspond to cuts of different orientations. Let ζ, δ and γ be valid sub-expressions in the Polish expression. Some terms are defined as follows:

Below : If $\gamma = \zeta\delta+$, $Below(\delta, \zeta)$

Above : If $\gamma = \delta\zeta+$, $Above(\delta, \zeta)$

Left : If $\gamma = \zeta\delta*$, $Left(\delta, \zeta)$

Right : If $\gamma = \delta\zeta*$, $Right(\delta, \zeta)$

Given a target module M_t, the algorithm *Find_Surrounding* finds four valid sub-expressions a, b, c and d such that $Below(\delta_1, a)$, $Above(\delta_2, b)$, $Left(\delta_3, c)$ and $Right(\delta_4, d)$ where $\delta_i's$ are some valid sub-expressions containing M_t.

Algorithm : Find_Surrounding(M_t, α)

Input : $\alpha = \alpha_1\alpha_2 \ldots \alpha_n$ *is a Polish expression of the original packing.*

t *is the index of the target module, i.e.,* $\alpha_t = M_t$

Output : a *is a valid sub-expression such that* $Below(\delta_1, a)$

b *is a valid sub-expression such that* $Above(\delta_2, b)$

c *is a valid sub-expression such that* $Left(\delta_3, c)$

d *is a valid sub-expression such that* $Right(\delta_4, d)$

where δ_i *for* $i = 1 \ldots 4$ *is the shortest valid sub-expression containing* M_t

such that the a, b, c *and* d *above can be found.*

1 $first = end = t$

2 *While* a, b, c, d *are not found and first*≥ 1 *and end*$\leq 2n-1$

3 *Begin*

4 *If* α_{end+1} *is an operator*

5 *Begin*

6 *Find* k *such that*

7 $e = \alpha_{first-k}\alpha_{first-k+1}\cdots\alpha_{first-1}$

8 *is the shortest valid sub-expression*

9 *If* α_{end+1} *is* $+$ *and* a *is not found yet*

10 $a = e$

11 *Else if* α_{end+1} *is* $*$ *and* c *is not found yet*

12 $c = e$

13 $first = first - k; end = end + 1$

14 *End*

15 *Else*

16 *Begin*

17 *Find* k *such that*

18 $e = \alpha_{end+1}\alpha_{end+2}\cdots\alpha_{end+k}$

19 *is the shortest valid sub-expression*

20 *If* $\alpha_{end+k+1}$ *is* $+$ *and* b *is not found yet*

21 $b = e$

22 *Else if* $\alpha_{end+k+1}$ *is* $*$ *and* d *is not found yet*

23 $d = e$

24 $end = end + k + 1$

25 *End*

26 *End*

The search starts from the position of the target module M_t in the Polish expression. We have two position pointers $first$ and end initialized as t (line 1). For each iteration, we will first check whether the character at the position $end + 1$ is an operator or an operand in the Polish expression. If it is an operator, the supermodule that shared the same slicing cut with the target module is located before the position of $first$ in the Polish expression (line 4-14). The supermodule is either on the left or below the target module. We will then search for the shortest valid sub-expression starting from the position $first - 1$ to the left. If the character at the position $end + 1$ is an operand, the supermodule that shared the same slicing cut with the target module is located after the position of end in the Polish expression (line 16-25). The supermodule is either on the right or above the target module. We will then search for the shortest valid sub-expression followed by an operator starting from the position $end+1$ to the right. The position pointer $first$ and end will then updated accordingly and the process will be repeated until a, b, c and d are all found.

The complexity of this algorithm is O(n). Figure 6.3 illustrates the steps of the algorithm. Sub-expression a, b, c and d are valid sub-expressions representing sub-trees in the slicing tree. For the example in Figure 6.2, $a = G, b = BC*ED+*, c = A, d = H$ and $M_t = F$. The shortest valid sub-expression can be obtained by counting the number of operators and operands. Note that not all the basic modules in a, b, c and d belong to the neighboring set Π_t of M_t. If the supermodule of a sub-expression b is above M_t, only the modules lying at the bottom of the supermodule belong to Π_t. Figure 6.4 shows an example that $b = BC * ED + *$ but D does not belong to Π_t in this case. A recursive procedure can be used to find Π_t efficiently given a, b, c and d. The procedure shown in the following is for sub-expression below the target module only, i.e., sub-expression a. Procedures for sub-expressions above, to the left and to the right of the target module can be done similarly.

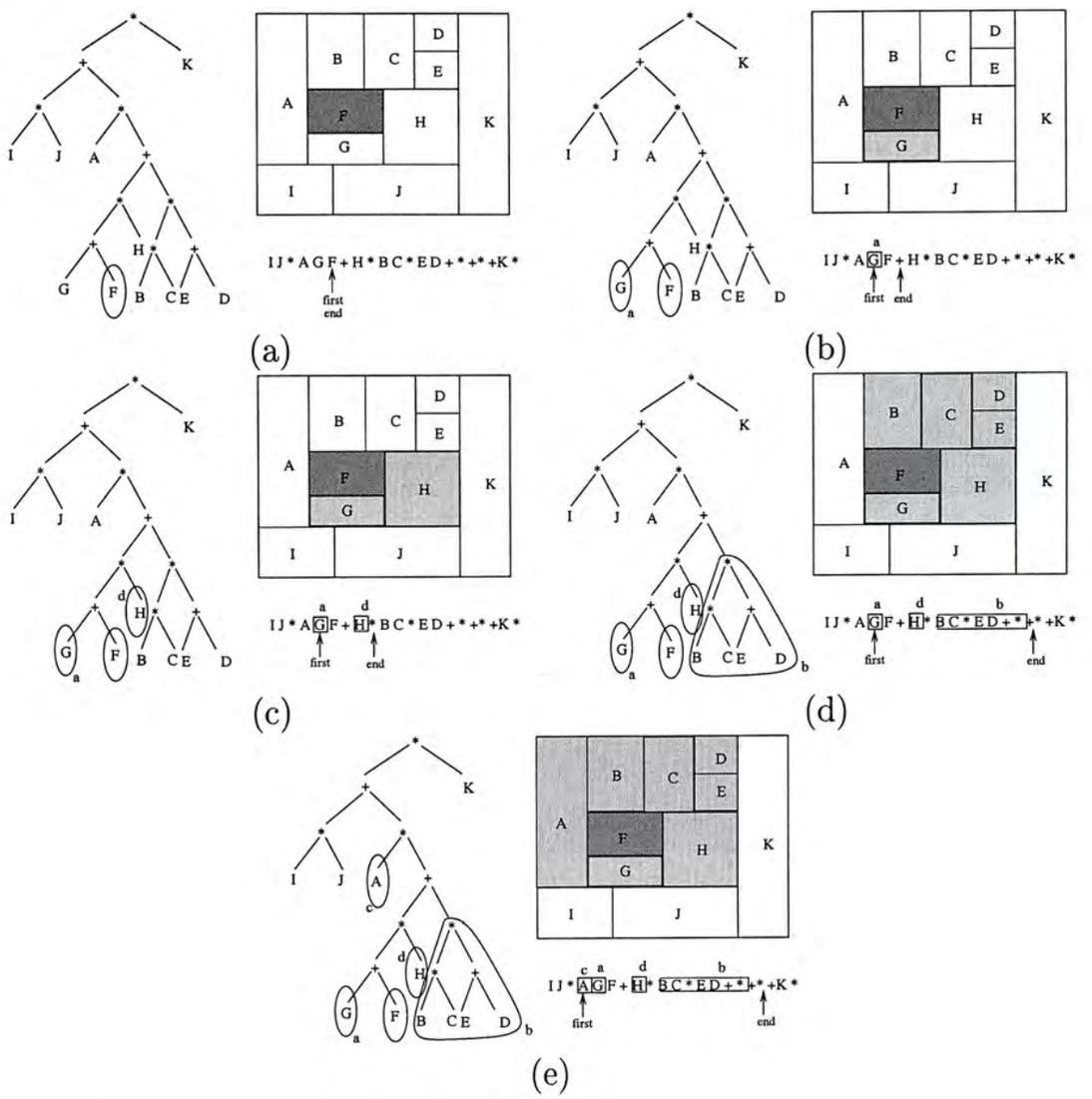


Figure 6.3: An illustration of the algorithm Find_Surrounding

Procedure : Marking_Neighbor_Below$(first, end)$

Input : $first$ *is the first index of the valid sub-expression a*

end is the last index of a where $first \leq end$

Output: Π *is the set of module at the bottom of the supermodule represented by a*

1 *If* $first = end$

2 $\Pi = \Pi \cup \alpha_{first}$

3 *Else*

4 *Begin*

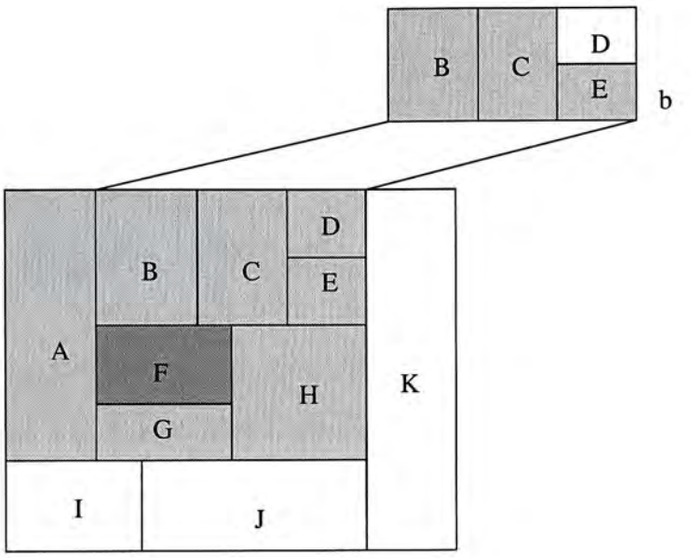


Figure 6.4: D does not belong to Π_t where M_t is F

```
5       Find k such that e = α_{end-k} α_{end-2} ... α_{end-1}
6           where e is the shortest valid sub-expression
7       If α_{end} is *
8           Marking_Neighbor_Below(first, end − k − 1))
9           Marking_Neighbor_Below(end − k, end − 1)
10      Else if α_{end} is +
11          Marking_Neighbor_Below(end − k, end − 1)
12  End
```

The above procedure is a recursive procedure locating all the modules at the top boundary of a given supermodule. The input of the procedure are two position pointers corresponding to a supermodule in the Polish expression. Therefore, the character at the position *end* should be an operator. We check whether this operator is corresponding to a horzontial cut or a vertical cut in the packing. Since we want to locating the modules at the top boundary of the supermodule, the supermodules on the both sides of the cut have to be considered if the cut is a vertical cut. Both supermodules will contain modules

at the top boundary. However, if the cut is horzontial, only the one at the top will be considered. All the modules at the top boundary will be marked as neighboring modules of the target module.

6.4 Constraint Satisfaction

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

In the first iteration, M_t is randomly selected from Δ. An intersect set Υ is defined to be $\Delta \cap \Pi_t$. If $|\Delta| > |\Upsilon| + 1$, swapping is needed to satisfy the clustering constraints. If $|\Delta| > |\Pi_t| + 1$, there is not enough space for swapping, the whole process will be repeated recursively by selecting another module which is already in the cluster as the new target module until all the constraints are satisfied. All three moves in the simulated annealing can affect the neighboring structure and give an infeasible packing. However we will swap operands in the Polish expression to maintain a feasible one.

There is only one case in move M1 that does not affect the neighboring structure of the packing, i.e., if two adjacent operands to be swapped are both in Δ or both in $\Phi - \Delta$. Modification is not required in this case and the clustering constraint will not be violated after the move.

The following algorithm describes the swapping strategy such that the clustering constraints are satisfy throughout the annealing process.

Algorithm : Clustering(α, Δ)

Input:$\alpha = \alpha_1\alpha_2 \ldots \alpha_n$ *is a Polish expression of the problem.*

Δ *is the set of modules having clustering constraint.*

1 *For each* $M_i \in \Delta$

2 *Begin*

3 *Call Find_Surrounding*(M_i, α)

```
4       Call Marking_Neighbor_Below(first(a), end(a))
5       Call Marking_Neighbor_Above(first(b), end(b))
6       Call Marking_Neighbor_Left(first(c), end(c))
7       Call Marking_Neighbor_Right(first(d), end(d))
8       Υ_i = Δ ∩ Π_i
9       If |Υ| + 1 == |Δ|
10          All clustering constraints satisfied
11          Return
12  End
13  count = 0
14  While count < |Δ| − 1
15  Begin
16      Take i where |Υ_i| is maximum and M_i is not marked
17      If |Π_i| ≥ |Δ| − 1
18      Begin
19          For each M_k ∈ Δ ∩ Π̄_i find M_j ∈ Π_i ∩ Δ̄_i
20              swap(M_j, M_k)
21          count = |Δ| − 1
22      End
23      Else
24      Begin
25          For each M_j ∈ Δ ∩ Π̄_i find M_k ∈ Π_i ∩ Δ̄_i
26              swap(M_j, M_k)
27          count = count + |Π_i|
28      End
29      Mark M_i
30  End
```

If $|\Pi_i| < |\Delta| - 1$ (lines 24-28), the number of positions in Π_i is not enough for swapping all the constrained modules into the neighboring positions. The other target module will be selected from Π_i and the process is repeated until all the constraints are satisfied. The algorithm can handle even very large cluster size.

6.5 Multi-clustering Extension

Multi-clustering constraint allows us to have more than one cluster in the final packing. The algorithm described above handles only one cluster. Multi-clustering constraints can be resolved by invoking the above algorithm several times. However, the major problem of addressing multi-clustering constraint is that the neighboring sets can overlap. Infeasible packing will be resulted if modules are swapped randomly. For example, given two clustering sets Δ_1 and Δ_2. A target module M_{t_1} and M_{t_2} is found from each clustering set. Let Π_{t_1} and Π_{t_2} be the neighboring sets of M_{t_1} and M_{t_2} respectively. If a module M_k, where $M_k \in \Pi_{t_1}$ and $M_k \in \Pi_{t_2}$ exists, the module M_k should be removed from either Π_{t_1} or Π_{t_2}.

Besides, while locating the neighboring modules, the module found earlier is probably nearer to the target module. It is thus better to swap into those positions first. This property make sure that the modules under clustering constraints will be placed as close to each other as possible.

6.6 Cost Function

The cost function is defined as $A + \lambda W + \beta C$ where A is the total area of the packing, W is the half perimeter estimation of the wirelength, and C is a penalty for the clustering constraint. The penalty term C is the sum of center to center distances between the modules within the same cluster.

The penalty term helps to give packings in which the modules with clustering constraints will be packed closely together. λ and β are constants that control the weighting between the importance of the three terms.

6.7 Experimental Results

Our method is tested with three MCNC building blocks examples (ami33, ami49 and playout). Ami33 has 33 modules and 123 nets. Ami49 has 49 modules and 408 nets. Playout has 62 modules and 1161 nets. In the first set of experiment, 20% of the modules in each benchmark are selected randomly to have clustering constraint, i.e., ami33, ami49 and playout have 7, 10 and 12 modules respectively. For each benchmark, we repeat the experiment three times by selecting different modules into the constraint set. The results are given in Table 6.1.

In the second set of experiment, we tested our method with multi-clustering constraints. In each benchmark problem, we picked 3, 4 and 5 clusters and each cluster has 2 to 7 modules. The results are given in Table 6.2. All the data are shown in the appendix A. A control experiment is performed without clustering constraint for each data set and the results are shown in Table 6.3. The temperature decreases with a constant rate (0.9), and the number of iterations at one temperature step is one hundred times the number of modules. All experiments were done on a UltraSPARC-II 400MHz processor.

Figure 6.5 and 6.6 shows a result packing of ami33 with three clusters and a result packing of ami49 with four clusters respectively. Figure 6.7 and 6.8 shows the improvement in interconnection by imposing clustering constraints. In Figure 6.7, we observed from the data set that modules 15, 18, 19, 20, 21, 24 and 25 are heavily connected with each other, so we impose clustering constraint between them. Figure 6.8 shows the result packing without imposing any clustering constraint. One can see that the interconnect cost in Figure 6.7

is smaller than that in Figure 6.8.

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

Table 6.1: Results of testing with one cluster for the MCNC examples

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

Table 6.2: Results of testing with multi-clusters for the MCNC examples

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

Table 6.3: Results of the control experiments

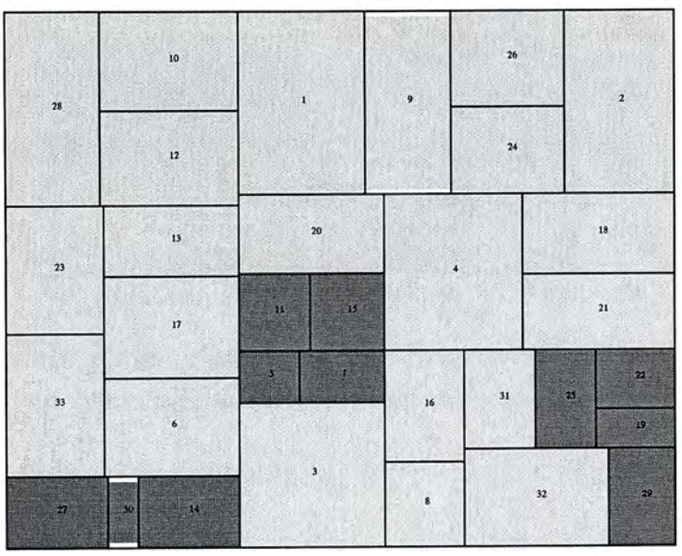


Figure 6.5: A result packing of ami33 with three clusters (C_1:5,7,11,13; C_2:14,27,30; C_3:19,22,25,29)

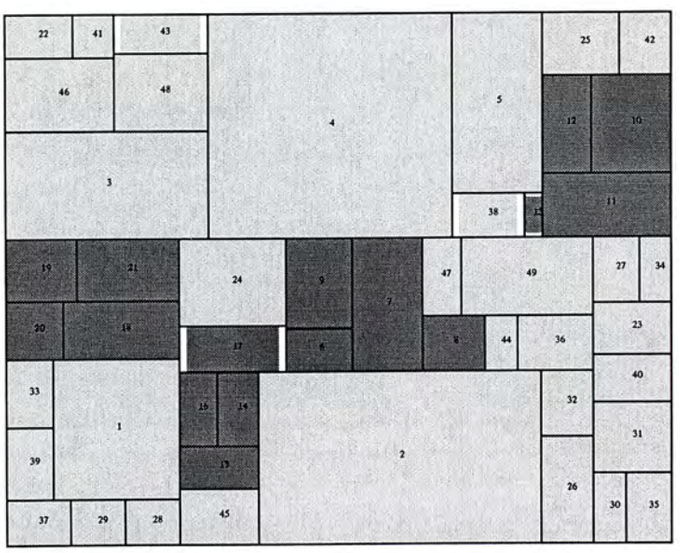


Figure 6.6: A result packing of ami49 with four clusters (C_1:6,7,8,9; C_2:10,11,12,13; C_3:15,16,17,18; C_4:18,19,20,21)

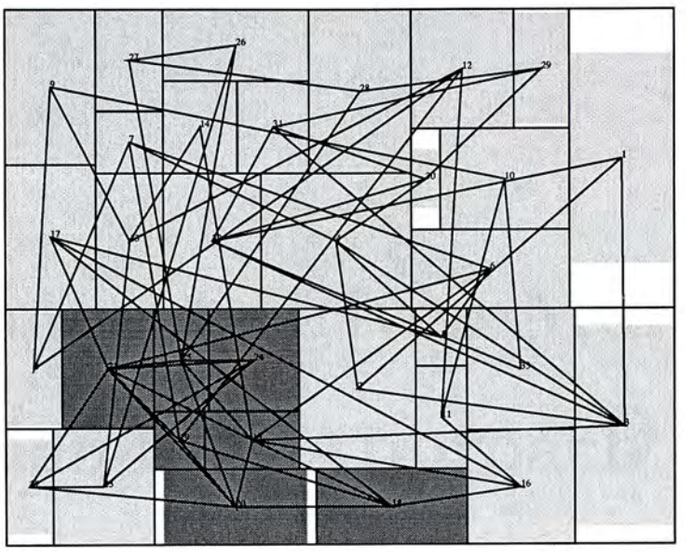


Figure 6.7: A result packing showing the improvement in interconnection by imposing clustering constraints (wirelength = 0.1472×10^6units).

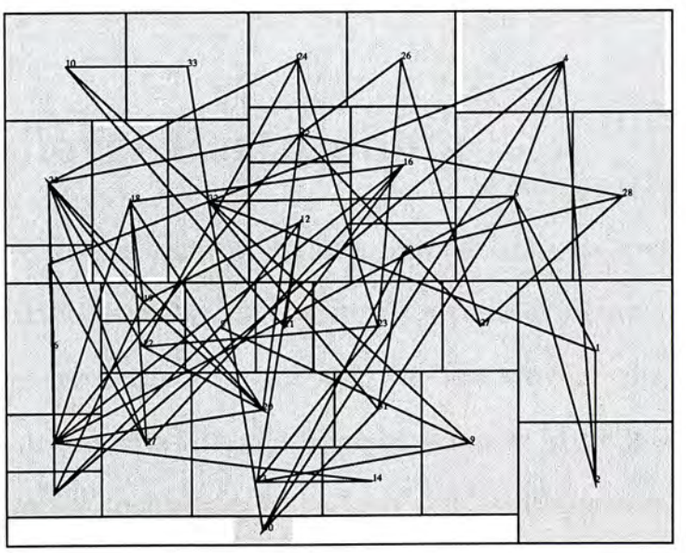


Figure 6.8: A result packing of the same problem in Figure 6.7 without clustering constraints (wirelength = 0.1596×10^6units).

Chapter 7

Interconnect Driven Multilevel Floorplanning Approach

In this chapter, a multilevel framework for floorplanning is presented. The Clustering and Refinement methods in the multilevel approach are described in details. The experimental results of the multilevel floorplanner are compared with those without applying the multilevel technique. Experimental results are shown and compared with some results recently published. The runtime of the implementation and the wirelength of the resultant packings are improved.

7.1 Multilevel Partitioning

Multilevel is a technique used in circuit partitioning to speed up the runtime [KK95, AHK98, WA98, KAKS99]. Circuit size is growing rapidly to millions of gates nowadays and the runtime will be too slow if use traditional partitioning methods are used to handle huge problems. Multilevel partitioning use a divide and conquer technique to reduce the problem size. Figure 7.1 illustrates the flow in multilevel partitioning. It consists of two phases: coarsening (clustering) and refinement (uncoarsening).

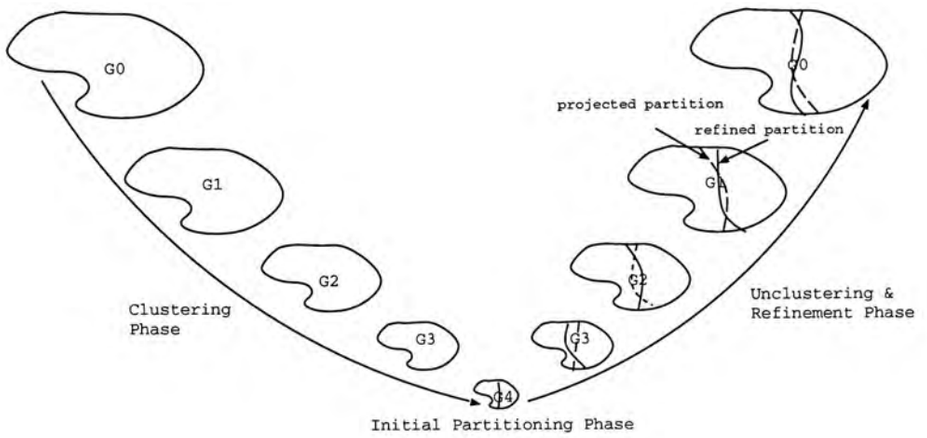


Figure 7.1: Multilevel Partitioning

7.1.1 Coarsening Phase

In the Coarsening Phase, the nodes (gates) are recursively grouped until the number of node (gates) is smaller than a certain threshold. This Coarsening step is illustrated in Figure 7.2.

Several nodes are grouped together to form a new node in the next level of the Coarsening Phase. The nets within the group will be removed. The net between two groups will be combined to form one net and the weight of the net will be updated. Therefore, the number of nodes and nets will be reduced in the next level of the Coarsening Phase. There are many different ways to group the nodes and these method are usually different in the way they consider the netlist information. These methods included Heavy Edge Matching [KK95], Random Matching, Hyper Edge Coarsening [KAKS99], etc. Most of the multilevel partitioners are using more than one methods.

7.1.2 Refinement Phase

After the Coarsening Phase, the problem size will be significantly reduced. Traditional partitioning methods are the applied to perform partitioning on this smaller problem instance. The solution will then be projected to the

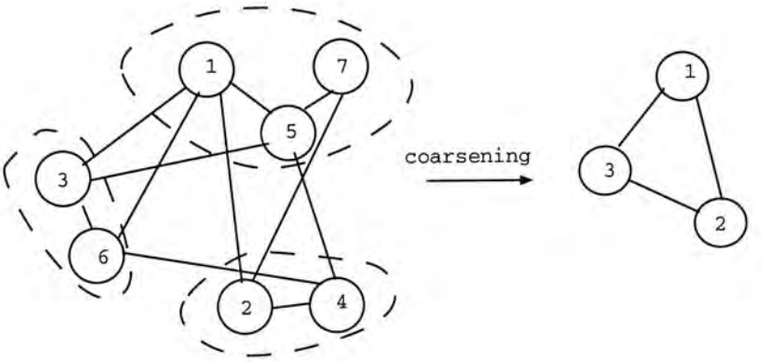


Figure 7.2: Coarsening Step

next level by ungrouping the nodes and adding the nets back to the graph. The partitioning algorithms will then be applied to this projected solution again. These steps are repeated in each level until all the nodes and nets are uncoarsened. Since the initial solution at the beginning of each level is a projected solution from the previous level and it is thus already a pretty good solution and the number of iterations needed to reach a good solution in each level can be reduced. Therefore, the total runtime will be reduced and can be used to handle larger problems.

Many research works have been done on applying and comparing different the kinds of partitioning methods in the Refinement Phase [KAKS99, AHK98, WA98, KAKS99]

However, using multilevel frameworks on placement and floorplanning is new and there are only a few publications on this topic. [CCKS00] presented a pioneer work of applying this multilevel framework on the circuit placement problem based on the constrained nonconvex nonlinear programming method to solve the problem.

The major difference between the work in [CCKS00] and ours' is that our multilevel framework is based on simulated annealing. Some techniques are used to determine the initial temperature in each stage and also, we will handle the sizing problem by the Lagrangian Relaxation technique.

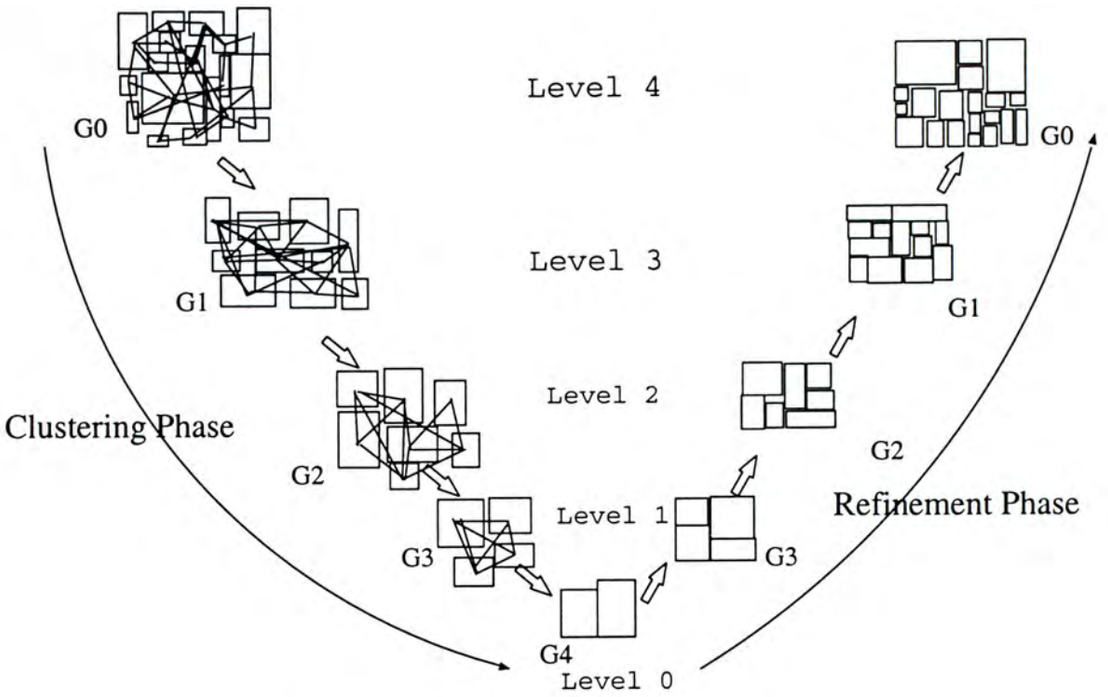


Figure 7.3: Illustration of a Multilevel Floorplanning

7.2 Overview of Multilevel Floorplanner

Multilevel approach can also be applied in the floorplanning problem. Figure 7.3 illustrates the steps of a multilevel floorplanner. There are two Phases in the multilevel floorplanner: Clustering and Refinement. In each level of the Clustering Phase, modules which are heavily connected with each other will be clustered together to form a new module. These new modules, each indeed is a collection of modules, will go through the same clustering process in the next level and this clustering process will repeat recursively until the number of modules remaining is small enough to be handled efficiently. Afterward, the Refinement Phase will perform refinement and packing. In each level of the Refinement Phase, the modules (each may be a cluster of modules itself) in a cluster will be unclustered and packed using some basic floorplanning algorithm. This refinement and packing step will be repeated in the next level using the results obtained from the current level as the initial solution.

The process of refinement and packing will be repeated recursively in each successive level until all the basic modules are unclustered.

In the following, we will discuss the clustering and Refinement Phase in more details.

7.3 Clustering Phase

The Clustering Phase performs grouping between the modules recursively in order to reduce the problem size and minimize the interconnect cost. In each level of the Clustering Phase, modules which are heavily connected with each other will be clustered together to form a new module for the next level. The area of the new modules will be the sum of the module areas in that cluster. The netlist information should also be reconstructed. The nets connecting modules in the same cluster can be removed, while those connecting modules in different clusters or connecting to an I/O pin will remain. The size of the circuit will thus be reduced successively during this Clustering Phase.

7.3.1 Clustering Methods

Given the netlist information, we consider two clustering methods, the Hyperedge Clustering Method and the Heavy Edge Matching Method:

Hyperedge Clustering [KAKS99]: In this clustering method, the hyperedges representing the nets are first sorted according to their weights. They are then scanned in descending order of their weights and modules belonging to the same *independent* net will be clustered together. A net is independent if and only if all the modules in it are still unclustered. Using this method, we can take care of the heavily weighted nets.

We restrict the number of modules that can be put into one cluster

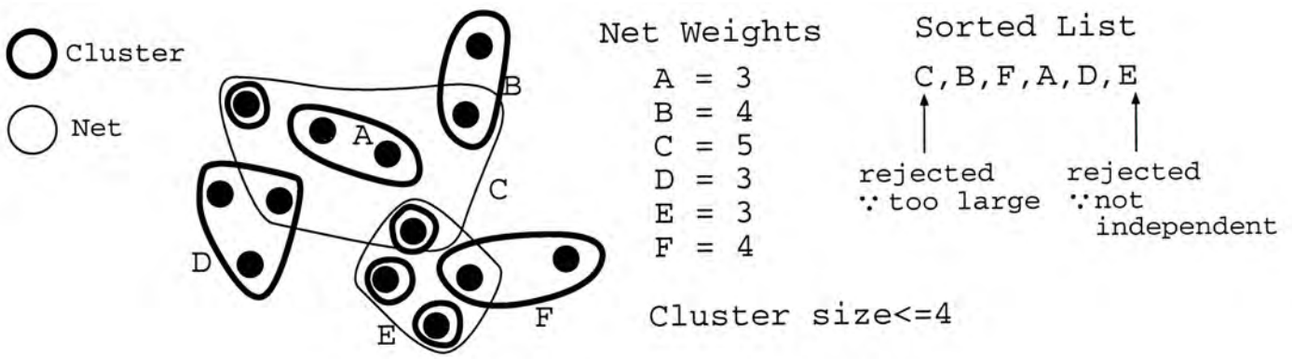


Figure 7.4: Illustration of The Hyperedge Clustering Method

to be within an upper bound (four in our case) because large cluster size is not good for the Refinement Phase. In the Refinement Phase, we will directly expand a cluster in the sequence pair representation by replacing a cluster name by a list of the modules in the cluster. This corresponds to packing the modules inside the cluster horizontally from left to right. This simple expansion will create deadspace in the packing and small cluster size can minimize this undesirable effect. Those modules which are not selected to be in any cluster will each be a cluster on its own. Figure 7.4 illustrates this method.

Heavy Edge Clustering: In this clustering method, modules will be clustered in pairs. From the netlist information, a simple graph G is built. In G, the vertices represent the modules and the edges represent the interconnection. A weight on an edge $e(i, j)$ represents the total number of nets connecting between module i and j. The edges are sorted in descending order of their weights. We will then scan the sorted list of edges. A pair of modules connected by an edge will form one cluster if both of them are not clustered yet. Figure 7.5 illustrates this heavy edge clustering

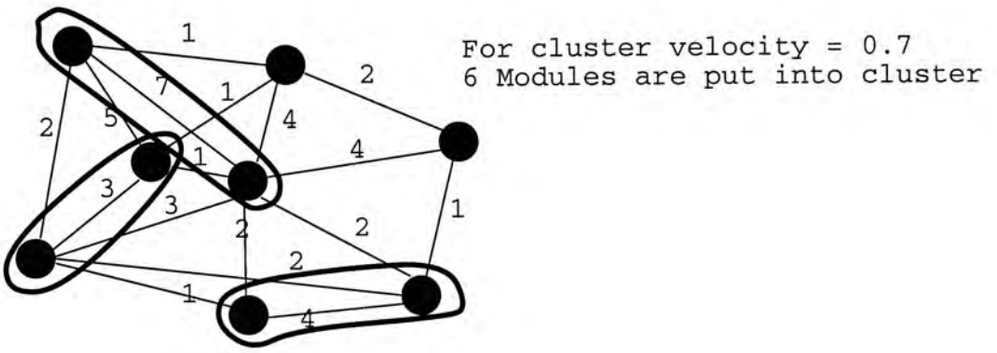


Figure 7.5: Illustration of The Heavy Edge Clustering Method

method.

7.3.2 Area Ratio Constraints

In the Clustering Phase, we need to control the area ratios between the modules in a cluster in order to obtain a tight packing at the end. For example, we should prevent very large modules from clustering with very small ones because packing modules of very different size is difficult due to the modules' aspect ratio bound. We impose two constraints on the area ratios during the Clustering Phase. The first constraint ensures that the clusters are formed progressively in the Clustering Phase:

$$\frac{\sum_{M_i \in C} Area(M_i)}{AREA} \leq r \times level \leq 1 \qquad \text{for each clusters } C$$

where $Area(M_i)$ denotes the area of module M_i, $AREA$ is the total area of all the modules, r is a constant and *level* is the current level in the Clustering Phase. By imposing this constraint, small modules can be clustered together to form larger ones and clusters of similar size will be formed progressively. This is good since packing of modules with very different sized is difficult and should be avoided. The second constraint prevents very large modules from clustering with very small ones:

$$\alpha_{min} \leq \frac{Area(M_i)}{Area(M_j)} \leq \alpha_{max} \quad \text{for all pairs of } M_i \text{ and } M_j \text{ which belong to the same cluster}$$

where α_{min} and α_{max} are the minimum and maximum area ratio allowed. These ratios are set to r and s in our implementation where r and s are the aspect ratio bounds of the modules. By imposing this constraint, large modules are prevented from clustering with small ones and packing within a cluster will thus be made easier.

7.3.3 Clustering Velocity

Clustering velocity is the percentage of modules that are being grouped into clusters at each level. If the clustering velocity is high (the number of levels will be small), the packing quality will be lower but the algorithm will be more efficient. On the other hand, if the clustering velocity is low (the number of levels will be large), the packing quality will be higher but the algorithm will be less efficient. It is important to control the velocity in order to yield a good result in a short runtime.

We will use a mixture of the Hyperedge Clustering method and the Heavy Edge Clustering method because each method has its own pros and cons. The Heavy Edge Clustering method gives clusters of size two only and a simple graph has to be built. However, it can control the clustering velocity accurately. On the other hand, the Hyperedge Clustering method can only give very few number of clusters at one level especially at those later stages. However, we can take cares of several nets with heavier weights by putting all the member of the nets into a cluster.

In each level of the Clustering Phase, we will first apply the Hyperedge Clustering method. If the percentage of modules being grouped into clusters is less than the required clustering velocity, we will apply the Heavy Edge Clustering method to increase the number of clusters to the required threshold.

The Clustering Phase is summarized as follows:

Algorithm: Clustering

Input: *A set* $C = \{M_1, M_2, \ldots, M_n\}$ *of* n *modules*
A set $N = \{N_1, N_2, \ldots, N_m\}$ *of* m *nets*
Clustering velocity v
Maximum number of clusters at the highest level K

Output: *A set of clusters* C^k *at each level* $1 \leq k \leq L$ *where* L *is the highest level number*
A set of coarsened nets N^k *at each level* $1 \leq k \leq L$

1 $C^0 = C$
2 $N^0 = N$
3 $k=0$
4 *while* $|C^k| \geq K$
5 *Perform Hyperedge Clustering on* (C^k, N^k) *to give* C^{k+1}
6 *If* $|C^{k+1}| \leq |C^k| \times v$
7 *Augment* C^{k+1} *with clusters obtained by performing Heavy Edge Clustering on the remaining unclustered modules.*
8 *Merging nets for the new set of clusters* C^{k+1} *to get* N^{k+1}
9 $k = k + 1$

7.4 Refinement Phase

After the Clustering Phase, we have all the information about the clusters and their interconnections at every level. We will then perform packing successively at each level starting from the coarsest one, i.e., the one with the fewest number of clusters. The packing at each level is done by simulated annealing using the sequence pair representation. The solution will be passed from one level to another by using the result packing at one level as the initial solution for the

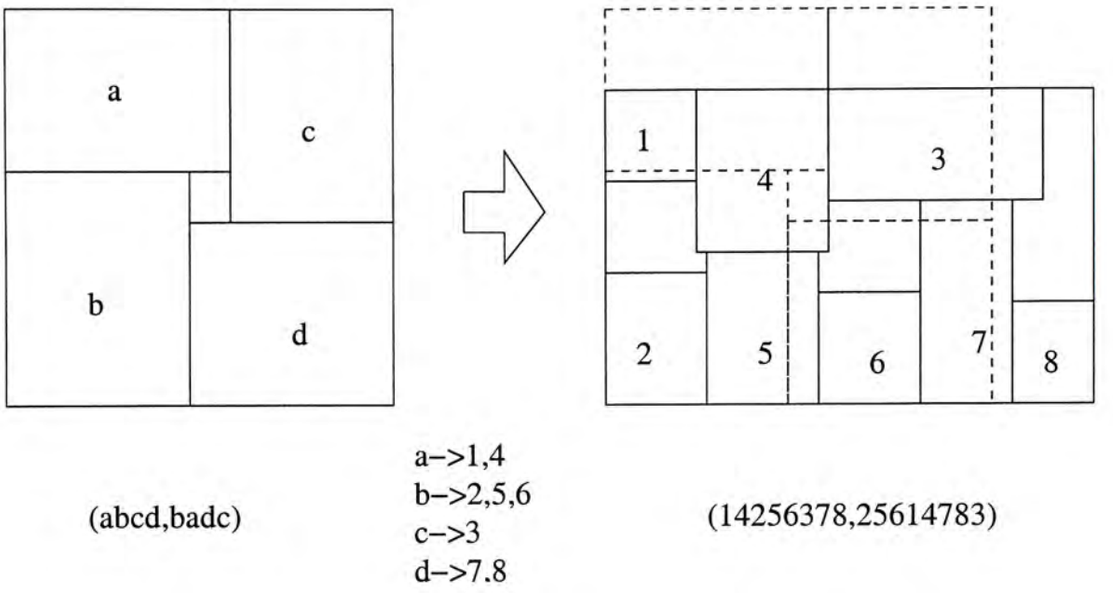


Figure 7.6: Illustration of the Sequence Pair Refinement

next. One advantage of using sequence pair is that refinement can be done directly on the representation. The sequence pair of the result packing at level i can be unclustered naturally by expanding each cluster into a list of modules it contains (as illustrated in Figure 7.6) which is then used as the initial solution for the next level.

Since the initial solution of the annealing process at one level is obtained from the result packing at the previous level, we expect that the number of iterations at each level to obtain a good packing will be reduced and the whole process can be sped up.

Algorithm: Refinement

Input: *A set of clusters* C^k *at each level* $1 \leq k \leq L$ *where* L *is the highest level*

A set of coarsened nets N^k *at each level* $1 \leq k \leq L$

Output: *A sequence pair* s^0 *for the modules in* C^0

1 $k = L$

2 *while* $k \geq 0$

3 *If* $k == L$

4 $s^k = (C_1^k C_2^k \ldots C_p^k, C_1^k C_2^k \ldots C_p^k)$ *where* $\{C_1^k, C_2^k, \ldots, C_p^k\}$ *is the set of clusters in* C^k

5 *else*

6 *Expand* s^{k+1} *to* s^k *by replacing each cluster* C_i^{k+1} *in* s^{k+1} *by a sequence of clusters from* C^k *contained in* C_i^{k+1}

7 *Perform simulated annealing on* C^k *and* N^k *using* s^k *as the initial solution*

8 $k = k - 1$

7.4.1 Temperature Control

In most simulated annealing process, the initial temperature is very high to allow random moves in the solution space and avoid being trapped in a local minimum. In our multilevel floorplanner the initial solution at each level of the Refinement Phase is already pretty good because it is obtained from the previous level by unclustering. A high initial temperature will, on the contrary, ruin the initial packing and waste the effort spent in the previous levels. In order to determine an appropriate initial temperature for the annealing process at each level, we will perform a certain number (proportional to the number of modules at the beginning of the annealing process) of random moves and an average change in cost is obtained from those iterations. The initial temperature is then computed according to the following equation:

$$T = -\frac{\delta C}{\ln r}$$

where T is the initial temperature, δC the average change in cost in the set of random moves and r is the probability to accept worse solution. This approach allows us to well control the starting temperature according initial probability of accepting a worse solution we want to have at the beginning of each level.

The initial probability of accepting a worse solution should decreasing during the refinement process since the packing is expected to be converging towards a good solution. In our implementation, it will be decreased linearly throughout the whole Refinement Phase.

7.4.2 Cost Function

Traditional floorplanners usually assume a die area with an unlimited size and pack the modules as tightly as possible. However, in fact, the die size is already chosen before floorplanning, so the packing should be performed in a fixed die regime. We use a better cost function that focuses more on optimizing the interconnect cost while constraining the final packing to within the fixed die regime. We compute the cost function in the annealing process as follows:

$$Cost = Wirelength + \lambda([W - W']^+ + [H - H']^+)$$

where $[x]^+ = x$ if x is positive and 0 otherwise, $Wirelength$ is the half perimeter estimation of the interconnect cost, W and H are the width and the height of the current packing solution, W' and H' are the width and height of the fixed die regime and λ is the weight. Notice that we are trying to minimize the interconnect cost as long as the modules are packed within the fixed die regime. Usually, λ is set to a large value to ensure that the constraint of fixed die regime can be satisfied. Packing outside the die regime will lead to a high cost and the solution will be rejected.

7.4.3 Handling Shape Flexibility

Many modules still have large flexibility in shape in the floorplanning stage and we can make use of this flexibility to improve the packing quality. We apply the Lagrangian relaxation technique in [YCLW00] to handle soft modules by invoking the shaping procedure once after the Refinement Phase. The soft

modules have fixed areas but their widths and heights can be changed as long as their aspect ratio are lying within a given bound. The shaping procedure is time consuming but there is no need to invoke it in every iteration of the annealing process to minimize the area. The runtime will be extremely long for large problem size if the shaping procedure is invoked in every iteration of the annealing process.

We should minimize the interconnect cost as much as possible while maintaining the packing within the fixed die regime. Applying the shaping procedure once to the last result packing of the Refinement Phase (which should already be quite good) can further reduce the deadspace by 4% to 8% to give a very tight packing. Note that the interconnect cost will not change much after the shaping procedure.

7.5 Experimental Results

The algorithms are written in C language. All experiments were done using Sun Enterprise E4500 with twelve UltraSPARC-II 400MHz processor and 8GB Memory running in Solaris 7 operating system. It is tested with three MCNC building blocks examples (ami33, ami49 and playout) and some randomly generated data set. Ami33 has 33 modules and 123 nets. Ami49 has 49 modules and 408 nets. Playout has 62 modules and 1161 nets. Data sets with 100, 200, 300, 400 and 500 modules and 888, 2388, 2888, 3388 and 3888 nets are generated randomly for testing purpose (for detail in appendix B). For those soft modules their aspect ratios are bounded to [0.5, 2.0] for problems with less than 400 modules. The soft modules in the data set with 500 modules has an aspect ratio bound of [0.1, 10.0].

7.5.1 Data Set Generation

In order to test the scalability of our floorplanner, we have randomly generated some data sets with up to hundreds of modules (data_100, data_200, data_300, data_400, data_500). The generator is given the number of modules, number of pins and number of nets and it will generate the area of each module, pin positions and netlist information automatically. For each data set, we define several ranges of area. The areas of each module is generated randomly so that the distribution of areas in each range is uniform. The pin positions are assigned uniformly at the boundary of the chip. The net information is also generated randomly. We will first determine whether a net is connected to a pin. Then, we will decide randomly the number of modules a net is connected to according to the distribution of nets and these modules will be picked randomly. All the module's number are resulting from a uniformly random number times some constants which according to number of modules of the problem set.

7.5.2 Temperature Control

As we mention before the initial temperature (T) is calculated using the change in cost. Figure 7.7, 7.8 and 7.9 plot of T at different level of the Refinement Phase for ami33, ami49 and playout respectively. The graphs also plot the number of nets that are unclustered at that level. The acceptance rate of a worse solution will change uniformly from 0.9 to 0.1 in the Refinement Phase. We can see from Figure 7.7, 7.8 and 7.9 that T is low at the first few levels because there are only a small number of clusters to be packed at the beginning of the Refinement Phase. T increases to its highest at some intermediate stages which is dependent on the interconnect structure of the individual data set. T is computed from the change in cost and the acceptance rate. Notice that the total areas of the modules will remain unchanged in the refinement process.

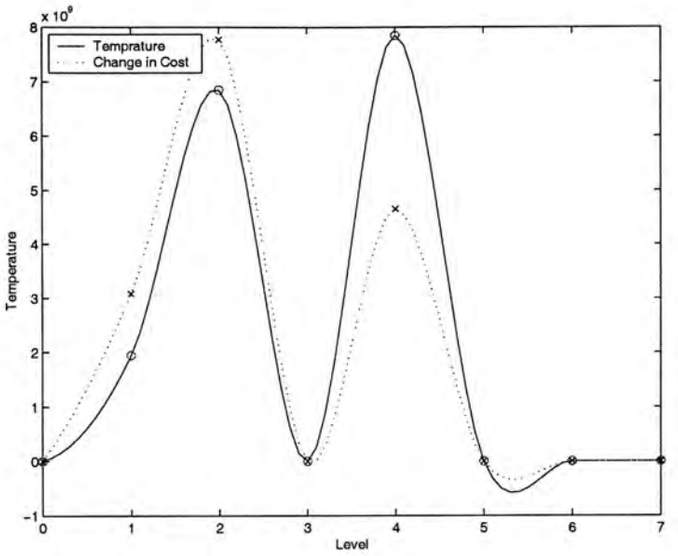


Figure 7.7: Initial temperature in the Refinement Phase for ami33

The cost of the packing will increase due to the uncoarsening of nets. T is lowest at the last level because the solution is already good.

7.5.3 Packing Results

Table 7.1 shows the packing results of our multilevel floorplanner for all the data set. Table 7.2 shows the results of the experiments using the original simulated annealing method using sequence pair representation without multilevel approach and shape flexibility. All experiments are done using the same set of parameters. The original algorithm can only handle up to two hundreds modules in practical time and the results for the problems with more than 200 modules is thus not shown (cannot be obtained in practical runtime).

The best result of ami33 and ami49 are compared with the result in [MK98]. It is shown in table 7.3. [MK98] used a quadratic programming method to perform area and wirelength optimization. Their results are nearly optimal but the runtime is very long and is unable to solve problem with large size in practical time.

Figure 7.10, 7.11, 7.12, 7.13 and 7.14 shows some result packings of the

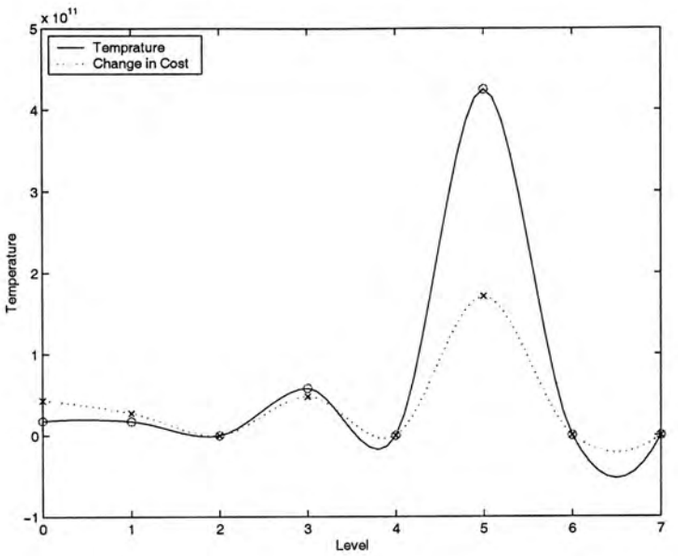


Figure 7.8: Initial temperature in the Refinement Phase for ami49

multilevel floorplanner. The deadspace of the packings are very small, from 0% to 4%.

Data set	#Mod	#Net	% Dead-space	Wlen $(\mu m)\times 10^6$	# Iter	Time\Iter (sec)$\times 10^{-3}$	Time (sec)
ami33	33	123	0.70	0.0587	441987	0.3373	149.1
ami49	49	408	2.05	0.8573	579307	0.7975	462.0
playout	62	1161	8.51	0.5216	774369	1.9220	1488.4
data_100	100	1888	1.84	2.8909	1020917	3.6225	3698.3
data_200	200	2388	4.28	4.5768	2350453	8.5513	20100.7
data_300	300	2888	4.31	6.7701	3804513	15.8206	60189.7
data_400	400	3388	5.44	8.8815	609360	38.9537	23740.5
data_500	500	3888	3.57	15.8380	922474	39.3843	36331.0

Table 7.1: Results of the multilevel floorplanner

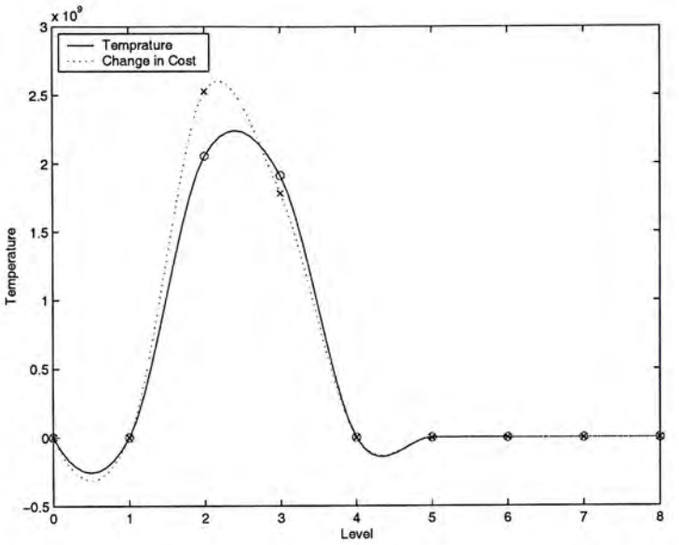


Figure 7.9: Initial temperature in the Refinement Phase for playout

Data set	#Mod	#Net	% Dead-space	Wlen $(\mu m)\times 10^6$	# Iter	Time\Iter (sec)$\times 10^{-3}$	Time (sec)
ami33	33	123	5.22	0.0680	529322	0.4682	247.9
ami49	49	408	4.58	1.0673	878082	1.0495	921.6
playout	62	1161	9.29	0.5855	1061642	2.6347	2797.2
data_100	100	1888	3.55	3.1716	1948002	4.5363	8836.9
data_200	200	2388	7.76	5.7849	4744002	13.0356	61845.2
data_300	300	2888	–	–	–	–	–
data_400	400	3388	–	–	–	–	–
data_500	500	3888	–	–	–	–	–

Table 7.2: Results of original algorithm without multilevel

Data set	Area (μm^2)	Wlen (μm)	Time (sec)	Area in [MK98] (μm^2)	Wlen in [MK98] (μm)	Time in [MK98] (sec)
ami33	1143430	56479	156.2	1159929	53393	75684
ami49	36709200	775486	447.1	35581225	775104	612103

Table 7.3: Results of comparison with [MK98]

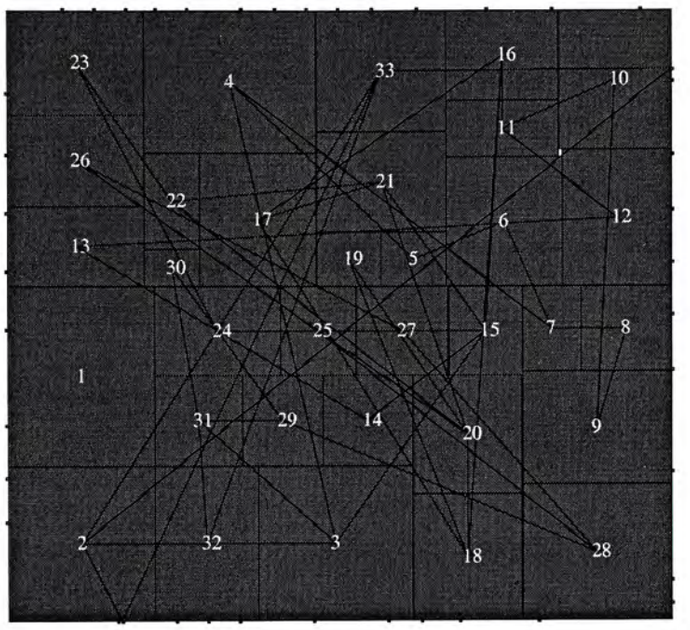


Figure 7.10: A result packing of ami33 (area = $1143430\mu m^2$, wlen = $56479\mu m$)

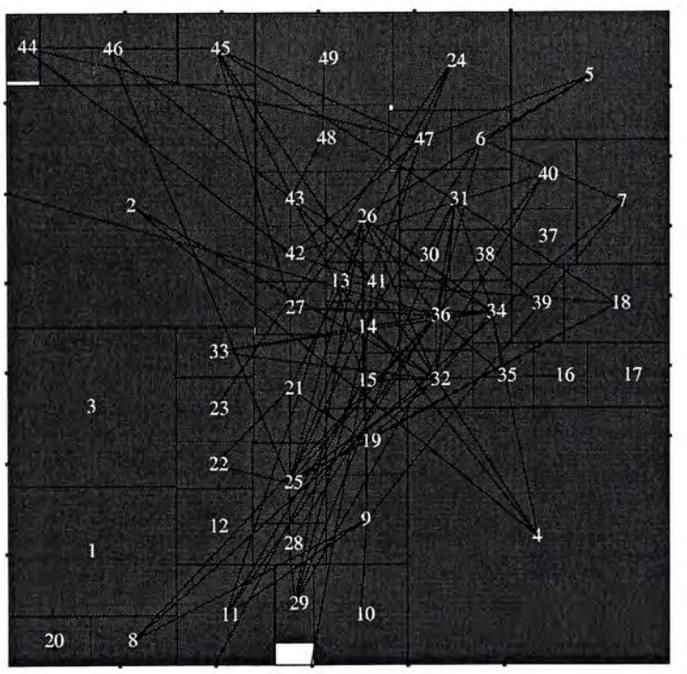


Figure 7.11: A result packing of ami49 (area = $35543600\mu m^2$, wlen = $907515\mu m$)

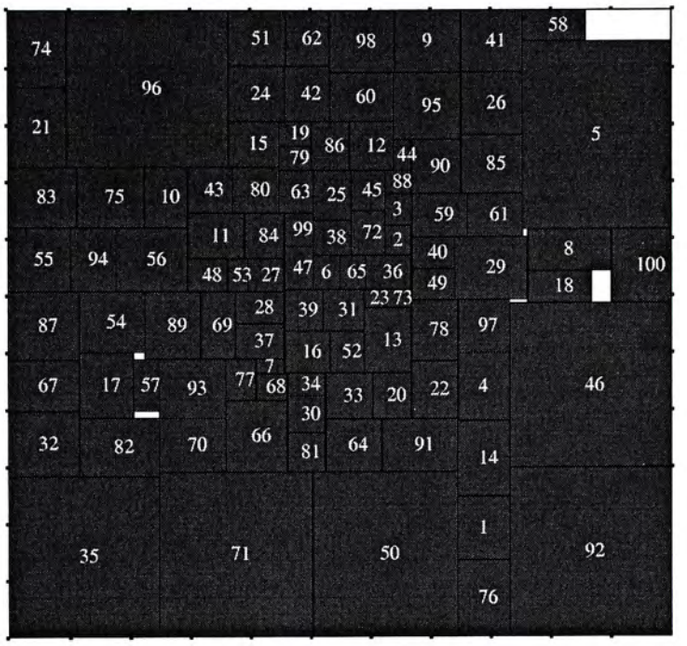


Figure 7.12: A result packing of data_100 (area = 8733540μm^2, wlen = 2811010μm)

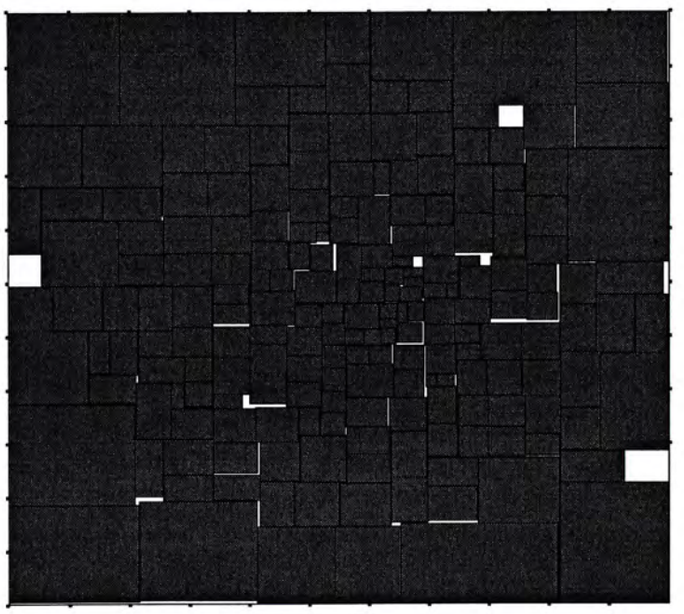


Figure 7.13: A result packing of data_200 (area = 17567900μm^2, wlen = 4506900μm)

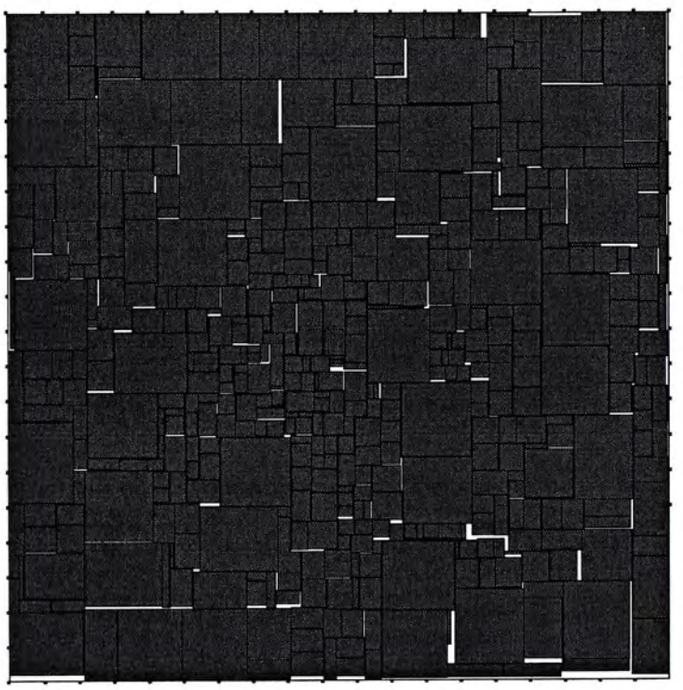


Figure 7.14: A result packing of data_500 (area = 49626700μm^2, wlen = 14559800μm)

Chapter 8

Study of Non-slicing Floorplan Representations

This chapter will analyze the complexity of different floorplan representations. We proved that if a floorplan is non-slicing there will be at least one super-module in the packing with its four T-junctions at the corners in four different orientations, i.e., a wheel. A new representation for mosaic floorplan called twin binary tree is proposed by [YCCG01] and we have derived an efficient algorithm to generate pairs of valid twin binary trees and to convert this representation to its corresponding packing efficiently.

8.1 Analysis of Different Floorplan Representations

In this section, we will discuss five popular floorplan representations: Polish expression(PE), Bounded-Slice-Grid(BSG), Sequence Pair(SP), O-tree(OT) and Corner Block List(CBL).

8.1.1 Complexity

Before we discuss the complexity of each representation, we need to know the enumeration of trees, because most of the representations make use of the trees.

An ordered binary tree is a binary tree in which the two children of each node are ordered. Let b_n be the number of different ordered binary tree with n nodes. For a binary tree with n nodes, each parent have at most two children. Therefore,

$$b_n = b_0 b_{n-1} + b_1 b_{n-2} + \ldots + b_{n-1} b_0$$

where $n \geq 1$. It is derived in [Knu93] that:

$$b_n = \frac{1}{n+1}\binom{2n}{n}$$

By Stirling's approximation,

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Therefore

$$\begin{aligned} b_n &= \frac{(2n)!}{(n+1)(n!)^2} \\ &= \frac{\sqrt{4\pi n}\left(\frac{2n}{e}\right)^{2n}}{(n+1)2\pi n\left(\frac{n}{e}\right)^{2n}} \\ &= \frac{4^n}{(n+1)\sqrt{\pi n}} \\ &= O\left(\frac{2^{2n}}{n^{1.5}}\right) \end{aligned}$$

1. **Polish Expression**

   Each Polish expression corresponds to a slicing tree in which every internal nodes has two children. The n leaf nodes are operands and the $n-1$ internal nodes are operators and the number of such slicing trees with different structures is equal to b_{n-1}. There are two types of operators and n leaf nodes. Therefore, there are 2^{n-1} possible assignments of operators

to the internal nodes, and $n!$ possible assignments of operands to the leaf nodes for each tree structure. The total number of combination of Polish expression is thus $(2^{2n-2}/n^{1.5}) \times 2_{n-1} \times n!$, i.e., $O(n!2^{3n-3}/n^{1.5})$.

2. **Baseline-Slice-Grid(BSG)**

   For a BSG with size $n \times n$, n room are selected for putting the modules. The total number of selection is $C(n^2, n)$. The total number of combination using BSG is thus $C(n^2, n) \times n!$, i.e., $O(n!C(n^2, n))$.

3. **Sequence Pair**

   There are two sequences of module names in a sequence pair. The number of possible assignments for each sequence is $n!$. The complexity is thus $O((n!)^2)$.

4. **O-tree**

   An ordered tree is a rooted tree in which the children of each node are ordered. Since every ordered tree can be converted to an ordered binary tree with a single child root uniquely, the number of ordered trees with n vertices is equal to the number of order binary tree with $n-1$ vertices, i.e., b_{n-1} [Knu93]. O-tree is an order tree with n vertices. The number of such trees is $2^{2n-2}/n^{1.5}$. The total number of combinations using O-tree representation is $(2^{2n-2}/n^{1.5}) \times n!$, i.e., $O(n!2^{2n-2}/n^{1.5})$.

5. **Corner Block List**

   S is a sequence of n modules and L is a sequence of $n-1$ bits. The total number of '1's in the list T should be less than or equal to $n-2$ and the number of '0's should equal to $n-1$. The paper [ZDH+01] has shown that the total number of T is $2^{2n-2}/n^{1.5}$. The total number of combination using corner block list representation is thus $n! \times 2^{n-1} \times 2^{2n-2}/n^{1.5}$, i.e., $O(n!2^{3n-3}/n^{1.5})$.

Table 8.1 summarizes the complexity of different floorplan representations and

Figure 8.1 plots these complexity measures (normalized by $n!$) in logarithmic scale. We can see that BSG has the largest solution space.

	Solution Space	Construction Time
Polish Expression(PE)	$\frac{n!2^{(3n-3)}}{n^{1.5}}$	n
Sequence Pair(SP)	$(n!)^2$	n^2
Baseline-Sliceline-Grid(BSG)	$n!(C(n^2,n))$	n^2
O-tree and B*-tree	$\frac{n!2^{(2n-2)}}{n^{1.5}}$	n
Corner Block List(CBL)	$\frac{n!2^{(3n-3)}}{n^{1.5}}$	n

Table 8.1: Complexity of different floorplan representations

8.1.2 Types of Floorplans

A mosaic floorplan is a floorplan without any empty room such that each room is acquired by one and only one module. Mosaic floorplan is introduced by [ZDH+01] and slicing floorplan is a subset of it. In the paper [YCCG01] the complexity of slicing floorplan and mosaic floorplan are derived. The exact number of slicing floorplan is a Schröder Numbers A_n given as below:

$$A_0 = 1$$

$$A_1 = 1$$

$$A_n = (3(2n-3)A_{n-1} - (n-3)A_{n-2})/n$$

The exact number of mosaic floorplan is a Baxter number $B(n)$ as given below:

$$B(n) = \binom{n+1}{1}^{-1}\binom{n+1}{2}^{-1}\sum_{k=1}^{n}\binom{n+1}{k-1}\binom{n+1}{k}\binom{n+1}{k+1}$$

The redundancies of a representation can be shown by comparing the exact number of packings and the complexity of the representation. Figure 8.2 and 8.3 plots the complexity of CBL with the exact number of mosaic floorplan. According to the plots, we can conclude that the redundancies in PE is greater

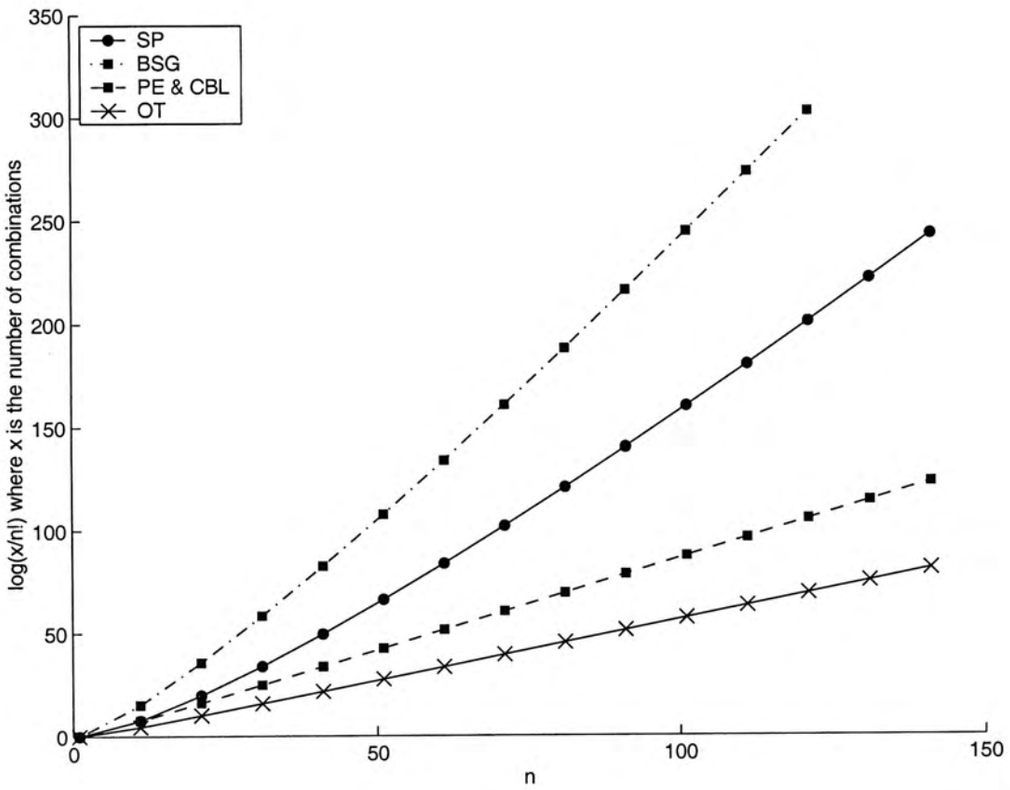


Figure 8.1: Complexity of different floorplan representations

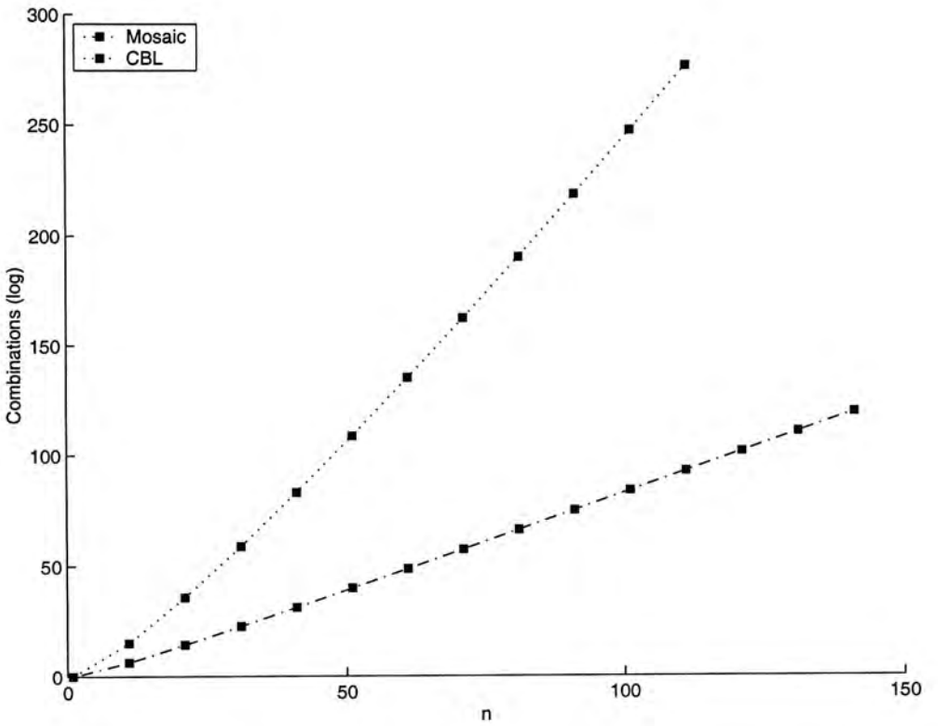


Figure 8.2: Complexity of CBL comparing with exact number of mosaic floorplan.

than that in CBL. The redundancies in Polish expression can be reduced by using normalized Polish expression. A normalized Polish expression is an expression in which all consecutive operators are different, i.e., no "++" nor "**" in the expression. Besides, redundancies CBL may also give infeasible representations (representation that does not correspond to a packing) during the moves.

The paper [LW01] introduced the concept of maximally compact placement. A packing with set of modules is maximally compact if no module in the packing can be moved horizontally to its left nor vertically downward without moving any other modules. An illustration is shown in Figure 8.4. Theorem 1 in the paper [LW01] have proved that for any maximally compact placement P, there will exist a slicing tree T such that performing compaction of the slicing placement P_T of T can generate P.

The complexity of maximally compact floorplan is difficult to obtain. A

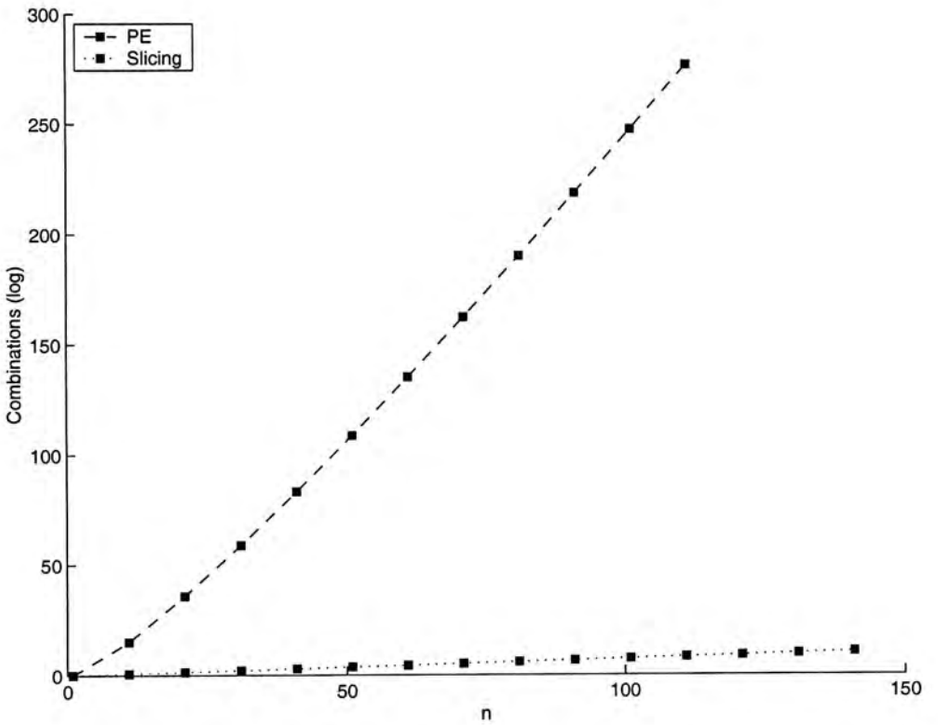


Figure 8.3: Complexity of PE comparing with exact number of slicing floorplan.

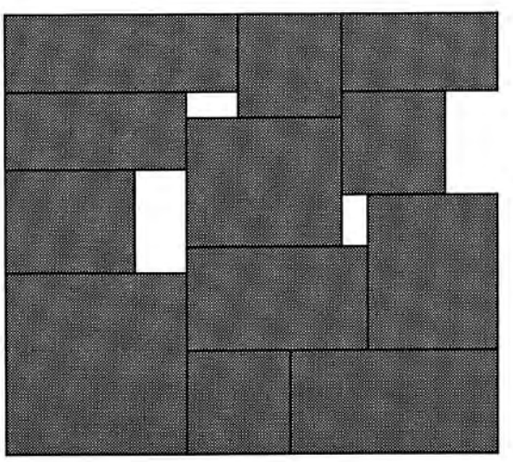


Figure 8.4: The illustration of Maximally Compact Placement

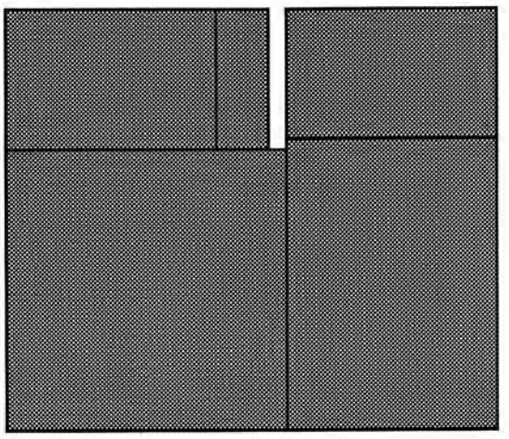


Figure 8.5: The illustration of slicing floorplan is not necessarily to be maximally compact

floorplan which is maximally compact depends on the dimensions of the module. Slicing floorplan may not be maximally compact. For example, Figure 8.5 give a slicing floorplan which is maximally compact even we force all the modules placed along the left and bottom boundaries of its room.

In the paper [ZDH+01], the authors consider a representation called Extend Corner Block List to represent maximally compact floorplans using CBL by introduce some dummy blocks into the list. λn of dummy blocks are introduced for a problem with n modules. The complexity of ECBL will become $(\lambda n + n)!2^{3(\lambda n+n)-3}/(\lambda n + n)^{1.5}$ which is larger than that of CBL.

According to above factors, slicing floorplans with smaller complexity but it may not be maximally compact. For those non-slicing floorplans representation which able to give maximally compact floorplans but they with larger complexity than slicing floorplan. Therefore, it is difficult to draw a conclusion about the complexity of maximally compact floorplan.

We can categorize four types of floorplan: slicing, mosaic, maximally compact and general. Table 8.2 shows the relationships between these popular representations and different kinds of floorplans.

	Slicing	Mosaic	Max. Compact	General
Expression(PE)	√			
Sequence Pair(SP)	√	√	√	√
Baseline-Sliceline-Grid(BSG)	√	√	√	√
O-tree and B*-tree	√	√	√	
Corner Block List(CBL)	√	√		

√ : can represent

Table 8.2: Relationship between representations and different kinds of floorplans

8.2 T-junction Orientation Property

At each corner of a module, a T-junction is formed between the modules.

Definition 8.1 $\Upsilon = \{\vdash, \dashv, \top, \perp\}$ represents the set of all four possible orientations of a T-junction at the corner of a module. For each module i in a packing, there is a corner sequence $s = (\alpha_{i_{LB}}, \alpha_{i_{LT}}, \alpha_{i_{RT}}, \alpha_{i_{RB}})$ where $\alpha_{i_{LB}}, \alpha_{i_{LT}}, \alpha_{i_{RT}}, \alpha_{i_{RB}} \in \Upsilon$ and

- $\alpha_{i_{LB}}$ corresponds to the lower left corner of the module.
- $\alpha_{i_{LT}}$ corresponds to the upper left corner of the module.
- $\alpha_{i_{RT}}$ corresponds to the upper right corner of the module.
- $\alpha_{i_{RB}}$ corresponds to the lower right corner of the module.

Definition 8.2 If a module is surrounded by four T-junctions in different orientations, it is said to be '4T'.

Definition 8.3 For a T-junction $\vdash$, the '|' is said to be the 'edge' and the '-' is said to be the 'dash'.

Definition 8.4 Given an edge of a module, if the orientation of two T-junctions at its two ends are opposite, i.e., $\top$ and $\perp$ or $\vdash$ and $\dashv$. the module is said to have an 'I' structure.

Lemma 8.5 In a *corner sequence* $s = \{\alpha_{i_{LB}}, \alpha_{i_{LT}}, \alpha_{i_{RT}}, \alpha_{i_{RB}}\}$ of a module i:

- $\vdash$ should not be assigned to $\alpha_{i_{RT}}$ or $\alpha_{i_{RB}}$.
- $\dashv$ should not be assigned to $\alpha_{i_{LB}}$ or $\alpha_{i_{LT}}$.
- $\top$ should not be assigned to $\alpha_{i_{LB}}$ or $\alpha_{i_{RB}}$.
- $\perp$ should not be assigned to $\alpha_{i_{LT}}$ or $\alpha_{i_{RT}}$.

Proof: For the T-junction $\vdash$, the edge is on the left hand side. Therefore, the module should be on the right hand side of the T-junction and $\vdash$ can only be assigned to $\alpha_{i_{LB}}$ or $\alpha_{i_{LT}}$ but not to $\alpha_{i_{RT}}$ nor $\alpha_{i_{RB}}$ for module i. For the T-junction of the other three orientations, similar argument follows. □

From Lemma 8.5, we have the following Corollary:

Corollary 8.6 For module i,

- $\alpha_{i_{LB}}$ is either $\vdash$ or $\perp$.
- $\alpha_{i_{LT}}$ is either $\vdash$ or $\top$.
- $\alpha_{i_{RT}}$ is either $\dashv$ or $\top$.
- $\alpha_{i_{RB}}$ is either $\dashv$ or $\perp$.

Lemma 8.7 The T-junction at the diagonal corners of a module should not have the same orientation.

Proof: Consider $\alpha_{i_{LB}}$ and $\alpha_{i_{RT}}$ which are in diagonal positions. According to Corollary 8.6, $\alpha_{i_{LB}}$ is either $\vdash$ or $\perp$ and $\alpha_{i_{RT}}$ is either $\dashv$ or $\top$. So, the orientation of $\alpha_{i_{LB}}$ should not be the same as that of $\alpha_{i_{RT}}$. It can be similarly proved for all other cases. □

Lemma 8.8 If a module is not '4T', there is at least one 'I' structure on one side of the module.

Proof: From Lemma 8.7, the T-junctions at the diagonal corners of module i should not have the same orientation. Therefore, if module i is not '4T', there is at least two T-junctions with the same orientation in its corner sequence s. The two T-junctions with the same orientation can only be at the two end points of one of its edges. Without lost of generality, we can assume that these T-junctions are at the two ends of the upper edge, we have $\alpha_{i_{LT}} = \alpha_{i_{RT}} = \top$. From Corollary 8.6, $\alpha_{i_{LB}}$ is either $\vdash$ or $\perp$ and $\alpha_{i_{RB}}$ is either $\dashv$ or $\perp$. Therefore, there will exist two T-junctions on one side of the module with opposite orientations. Hence, an 'I' structure exists. □

Lemma 8.9 If a module has an 'I' structure, the module is either obtained by a slicing cut or there exists a '4T' module (or supermodule) in the packing.

Proof: Without lost of generality, we consider the case that module M_j has an 'I' structure at its right edge, i.e., $\alpha_{j_{RT}} = \top$ and $\alpha_{j_{RB}} = \perp$. Then, there exists a set of modules $N = \{M_0, M_1, \ldots, M_k\}$ on the right hand side of M_j such that one of their $\alpha_{i_{LT}}$'s and $\alpha_{i_{LB}}$'s where $i = 0, \ldots, k$ share the T-junction $\top$ and $\perp$ with M_j. There are two cases for the placement of the modules in N:

- $|N| = 1$, i.e., $k = 0$. We have $\alpha_{0_{LT}} = \alpha_{j_{RT}} = \top$ and $\alpha_{0_{LB}} = \alpha_{j_{RB}} = \perp$. The 'I' structure is a slicing cut between module M_k and M_0.

- $|N| > 1$, Since $|N| > 1$, there exists at least one T-junction $\vdash$ such that its edge is on the right edge of M_j, i.e., along the 'I' structure. From the horizontal segment at the top and bottom of the 'I' structure, there should be a falling edge and a rising edge (Figure 8.6 and 8.7). There are two cases:

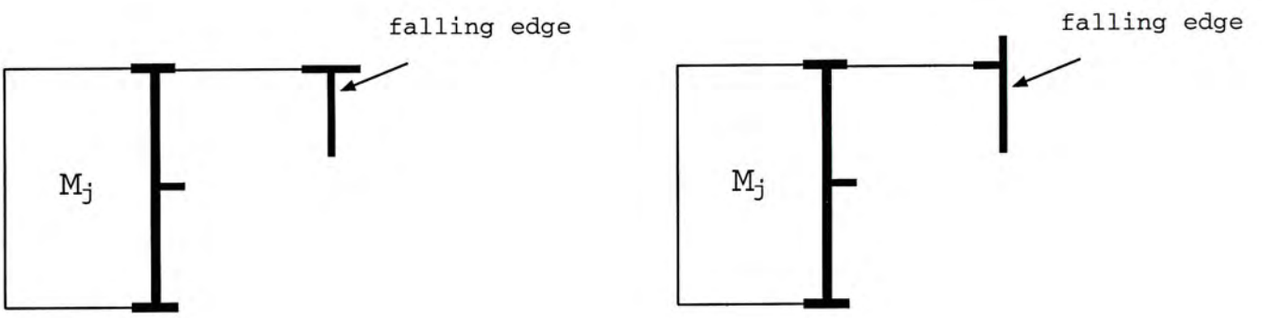


Figure 8.6: A falling edge from the top of the 'I' structure.

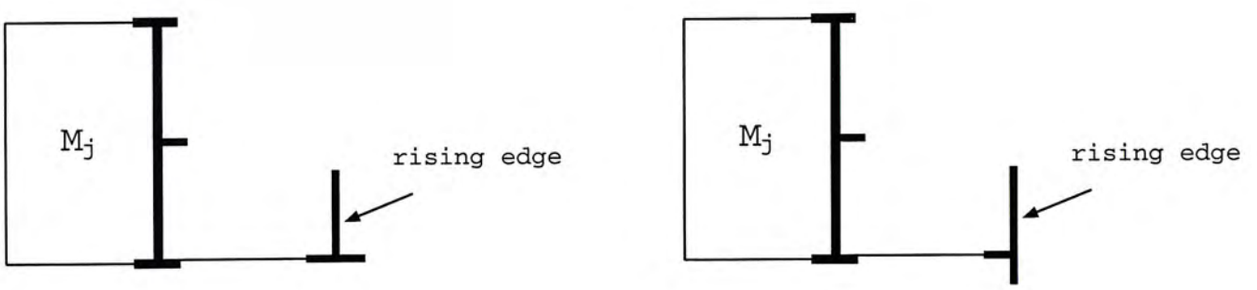


Figure 8.7: A rising edge from the bottom of the 'I' structure.

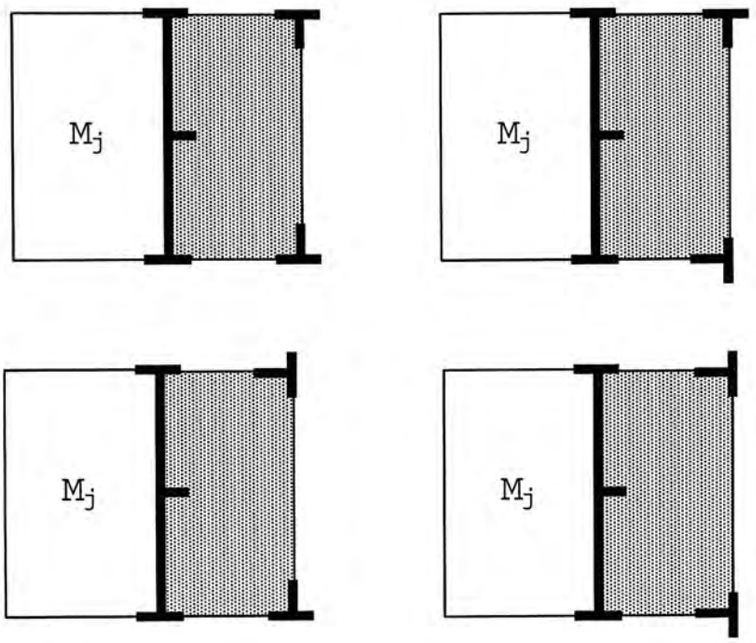


Figure 8.8: The four possible cases when the rising and falling edges are one vertical segment.

- The falling edge and the rising edge is the same vertical segment. There will be four possible cases as shown in Figure 8.8. Note that the modules in the shaded area will form one supermodule and this case will thus be the same as that when $|N| = 1$, i.e., the 'I' structure is a slicing cut between M_k and the supermodule in the shaded region.

- The falling edge and rising edge are not the same vertical segment. Consider the top segment of the 'I' structure, it should terminate in one of the two possible ways as shown in Figure 8.9. These two cases are further elaborated in Figure 8.10. For case 1 in Figure 8.10, the top segment of the 'I' structure terminates at a corner of the floorplan. The bottom segment from the 'I' structure (edge *d*) must not meet edge *b*. Therefore, it must meet with another vertical edge *e*. Edge *e* must not meet edge *a*, so it must meet with

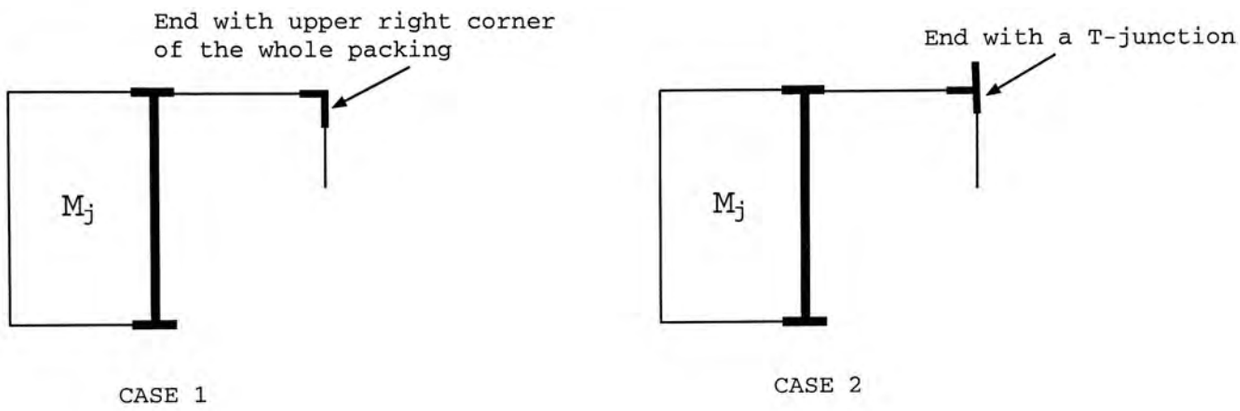


Figure 8.9: Two possible cases for the top segment of the 'I' structure

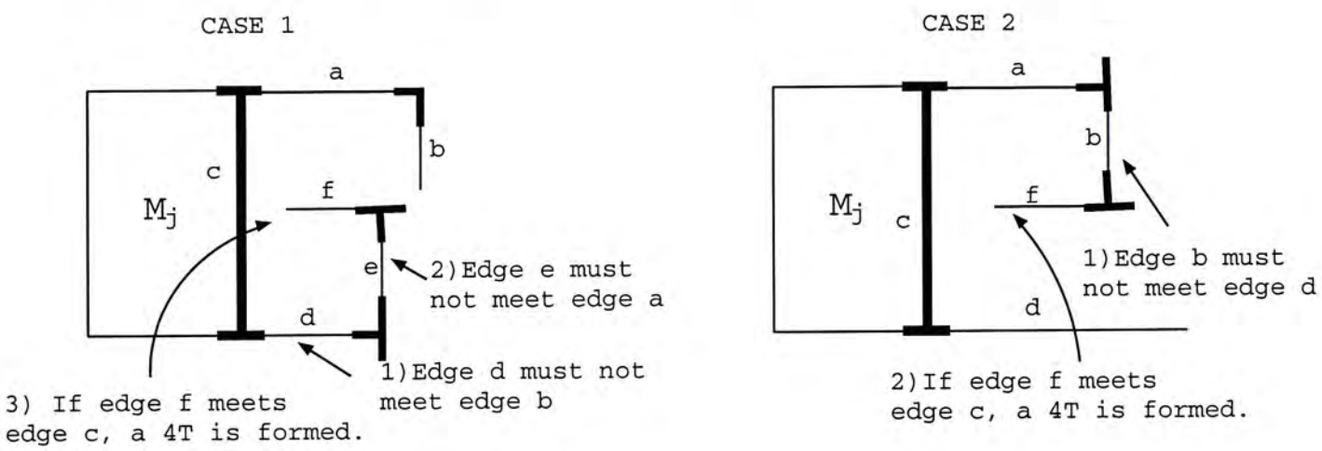


Figure 8.10: The illustration of formation of 4T modules

another horizontal edge f. If edge f meets edge c, a '4T' supermodule is formed; otherwise, the argument will repeat until a '4T' supermodule is formed finally. For case 2 in Figure 8.10, the top segment of the 'I' structure terminates at a $\dashv$ T-junction. Edge b must not meet the bottom segment of the 'I' structure, so it must meet with another horizontal edge f. If edge f meets edge c, a '4T' supermodule is formed; otherwise, the argument will repeat until a '4T' supermodule is formed. □

Theorem 8.10 A floorplan which is not slicing, will have at least one '4T'

module.

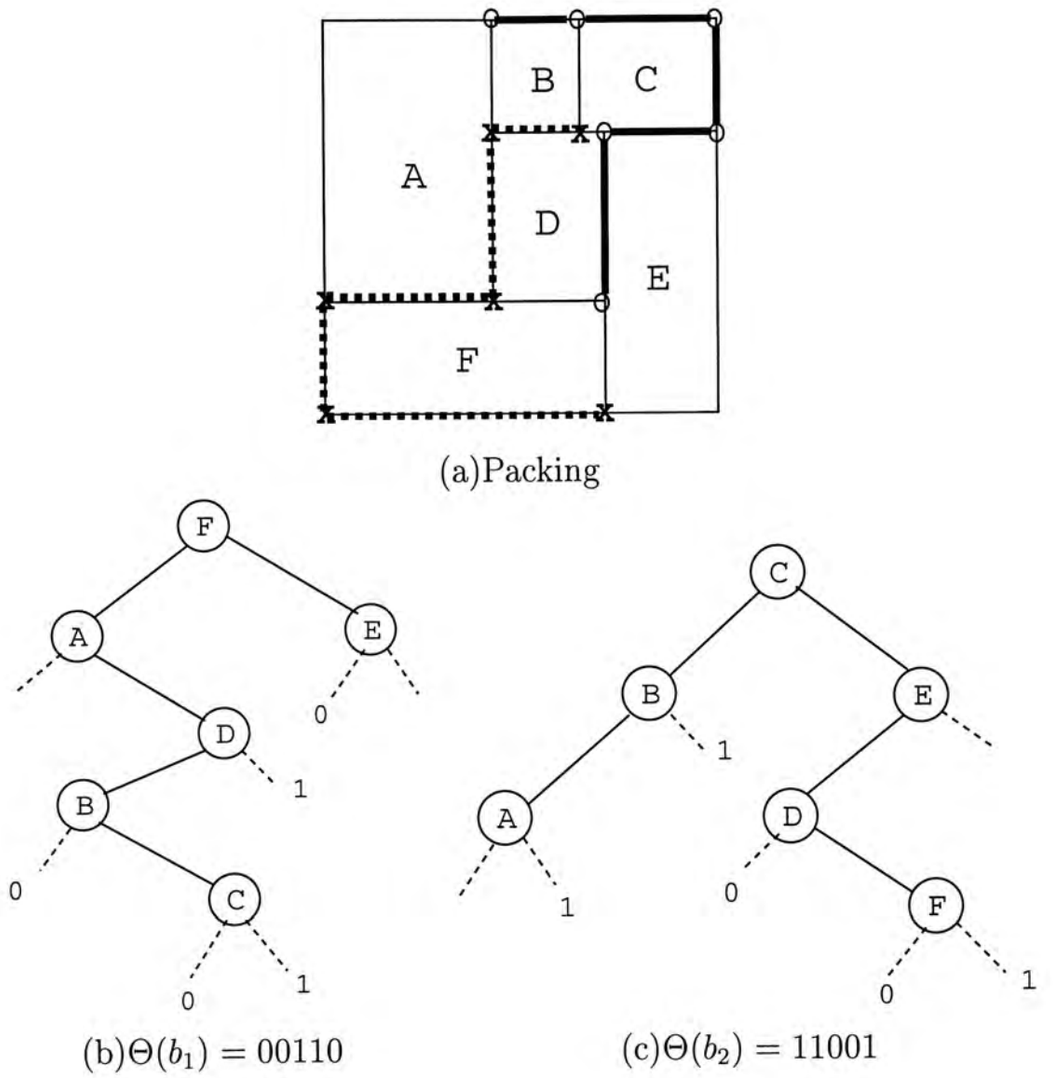
Proof: Assume that a floorplan which is not slicing has no '4T' module. From Lemma 8.8, there exist at least one 'I' structure in every module of the floorplan. From Lemma 8.9, if a module has an 'I' structure, the module is either obtained by a slicing cut or there exists a '4T' module within the floorplan. The second case will contradict with the assumption that the packing has no '4T' module. That means, all the modules are obtained by slicing cuts and this contradicts with the fact that the packing is not slicing. Therefore, we can conclude that a floorplan which is not slicing should have at least one '4T' module. □

8.3 Twin Binary Tree Representation for Mosaic Floorplan

8.3.1 Previous work

The paper [YCCG01] have proposed a twin binary tree representation for mosaic floorplan. However, they only devised the algorithm to convert a packing to its twin binary tree representation and proved that for any given mosaic floorplan, there exists a unique twin binary tree representation. In this section, we will describe an efficient algorithm to construct the packing from a twin binary tree representation.

Twin binary tree $TBT_n = \{(b_1, b_2) \mid b_1, b_2 \in Tree_n \text{ and } \Theta(b_1) = \Theta^c(b_2)\}$ where $Tree_n$ is the set of binary trees with n nodes and $\Theta(b)$ is the labeling of a binary tree b. The labeling of a binary tree is a sequence of '0' and '1' bits obtained by an in-order traversal of the tree in such a way that a bit '0' is appended to the sequence when a node with no left child is visited and a bit '1' is appended when a node with no right child is visited. The first and the



(a)Packing

(b)$\Theta(b_1) = 00110$

(c)$\Theta(b_2) = 11001$

Figure 8.11: A packing and its twin binary tree representation

last bit can be omitted since they are the same for all binary trees. $\Theta^c(b)$ is the complement of $\Theta(b)$ by changing all the bits '1' to '0' and '0' to '1'.

The paper [YCCG01] have proved that for a mosaic floorplan there exists a unique twin binary tree representation. Figure 8.11 shows an example of the representation. The roots of the twin binary tree will correspond to the top right corner block and the bottom left corner block of the packing, and these two blocks will be leaf nodes in the other tree of the pair.

In the binary tree b_1, the root node corresponds to the the module at the

bottom left corner of the packing. The two branches from this root will travel upward and to the right until they reach another T-junctions which are the bottom left corners of some other modules, then new nodes are formed corresponding to the modules reached. Note that the bottom left corner of all the modules can be reached in this way. The construction of b_2 is similar, except that the root node now corresponds to the module at the top right corner of the packing and the branches of the tree travel downwards and towards the left.

The method [YCCG01] described above shows the conversion from a packing to the twin binary tree representation. However, another important step is to generate valid twin binary trees and to convert an arbitrary twin binary tree representation to its packing efficiently. In the following section, we will describe an algorithm to achieve this purpose.

8.3.2 Twin Binary Tree Construction

From the twin binary tree in Figure 8.11, we observe that the in-order walk of the two twin trees are the same, while their labelling are complements of each other.

Lemma 8.11 For the in-order traversal of b_1 and b_2, where b_1 and b_2 are the twin binary tree constructed from a packing by the method described in Section 8.3.1, the first node in the two traversals are the same and it corresponds to the module at the top left corner of the packing.

Proof: In b_1, the first node visited in an in-order traversal is the left most child which is reached by traveling upward from the bottom left corner until there is no more module on the top. Therefore, it is the module at the top left corner of the packing. For b_2, the first node visited in an in-order traversal is the left most child which is reached by traveling towards the left from the top

right corner until there is no more module on the left. Therefore, it is also the module at the top left corner of the packing. □

Theorem 8.12 Given a packing and its twin binary tree representation b_1 and b_2, the sequences obtained by in-order traversals of the trees including the bits labeled at the nodes with either no right child or no left child (we call this sequence an *in-order sequence*) will be the same except that, the bits in the two sequences are complemented, for example, A1B1C0D0F1E1 and A0B0C1D1F0E0 is a pair of twin binary tree of a feasible packing.

Proof: The proof is done by induction. The proof for the base case with only one module is trivial and is not shown here. Assume that the statement is true for any packing with k modules where $k \geq 1$. Consider a packing with $k+1$ modules. Let module n_i be the top left corner of the packing, and b_1 and b_2 are the corresponding twin binary tree. From lemma 8.11, node n_i is the left most child in both b_1 and b_2. There are two possible cases:

- Case 1: n_i has a right child in b_1.
  If n_i has a right child in b_1, then n_i in b_2 will have no right child. Figure 8.12 illustrates this case. The in-order sequences of b_1 and b_2 are $n_i 0 T_2 T_1$ and $n_i 1 T_3$ respectively. However, $T_2 T_1$ and T_3 are the in-order sequences of the twin binary tree b_1' and b_2' for the packing P' obtained from P by sliding the edge a (Figure 8.12) to the left boundary of the whole packing to remove n_i from P. By the induction hypothesis, $T_2 T_1$ and T_3 are the same except that the bits are toppled, so $n_i 0 T_2 T_1$ and $n_i 1 T_3$ also have the same property.

- Case 2: n_i has no right child in b_1
  If n_i has no right child in b_1, then n_i in b_2 will have a right child. Figure 8.13 illustrates this case. The in-order sequences of b_1 and b_2 are $n_i 1 T_1$ and $n_i 0 T_3 T_2$. T_1 and $T_3 T_2$ are the in-order sequences of the twin

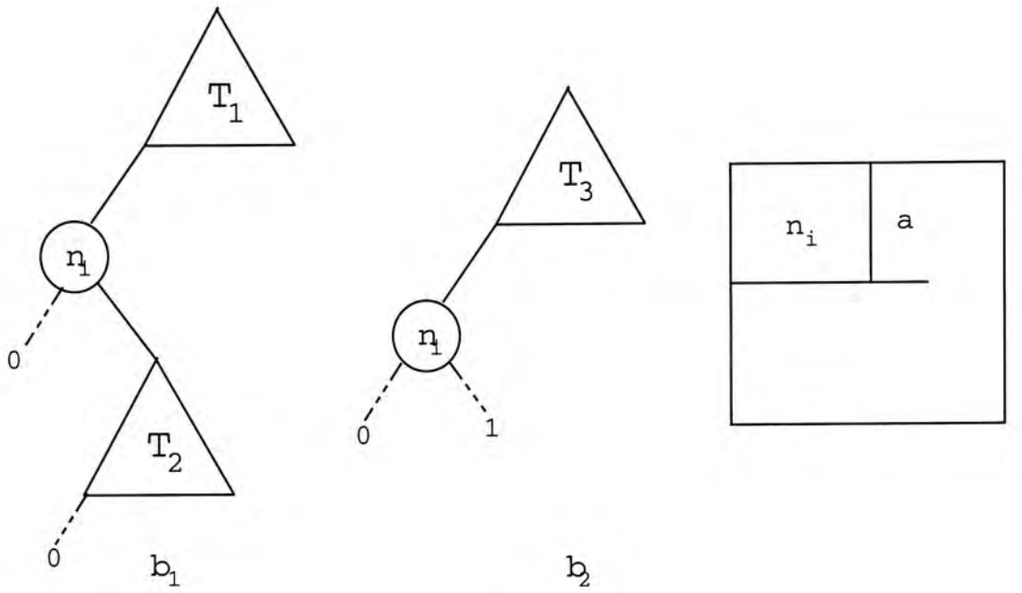


Figure 8.12: The illustration of case 1 in the proof of Theorem 8.12

binary tree b'_1 and b'_2 for the packing P' obtained from P by sliding edge b (Figure 8.13) to the top boundary of the whole packing to remove n_i from P. By the induction hypothesis, T_1 and T_3T_2 are the same except that that the bits are toppled, so n_i1T_1 and $n_i0T_3T_2$ also have the same property. □

A binary tree b_1 is given in Figure 8.14 and its in-order sequence is C0D1B1A0E. Using Theorem 8.12, we can obtain the in-order sequence of its twin binary tree by complementing all the bits in the sequence, i.e., C1D0B0A1E. Then, we can construct the twin binary tree according to this in-order sequence. Note that there may be more than one trees with that in-order sequence (Figure 8.14), and these are the possible twin binary trees of b_1. Note that a leaf node of b_1 will be the root node of b_2, and a leaf node of b_2 will be the root node of b_1. Each b_2 can pair up with b_1 to generate a packing using the method described in Section 8.3.3.

All possible b_2 of a given b_1 can be obtained from the in-order sequence of b_2 since the positions of all the internal nodes and leaf nodes are known.

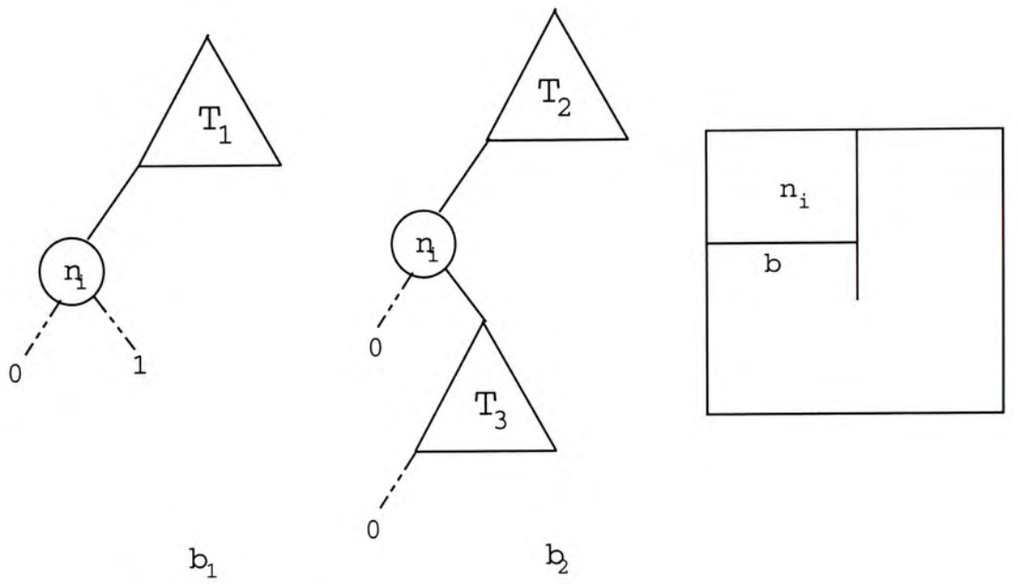


Figure 8.13: The illustration of case 2 in proof of Theorem 8.12

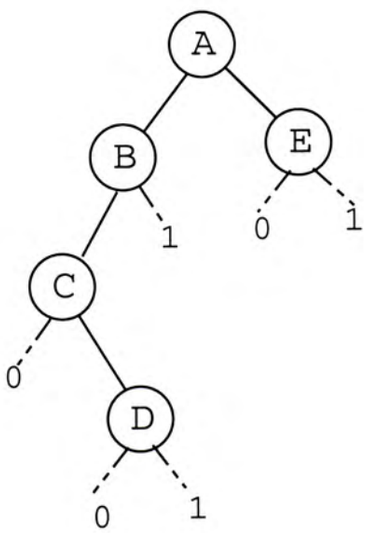


Figure 8.14: A example of binary tree b_1

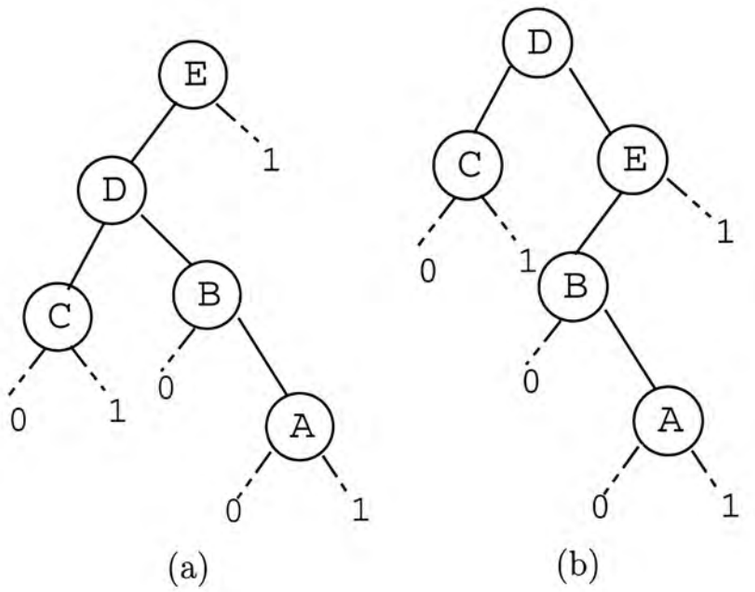


Figure 8.15: The possible b_2 for the b_1 in Figure 8.14

8.3.3 Floorplan Construction

When we have a pair of twin binary tree, we can construct the packing from the two trees. The binary trees b_1 and b_2 in Figure 8.14 and Figure 8.15(a) are used as an illustration.

Consider a node A in a binary tree b_1. The right child E of node A corresponds to the module on the right of A. The left child B of node A corresponds to the module on top of A. Therefore, we have the following equations from the binary tree b_1:

$$x_i = x_j + w_j$$

where module i is the right child of module j, and

$$y_i = y_j + h_j$$

where module i is the left child of module j, w_j and h_j are the width and height of module j, and x_i and y_i are the x-coordinate and y-coordinate of the bottom left corner of module i respectively.

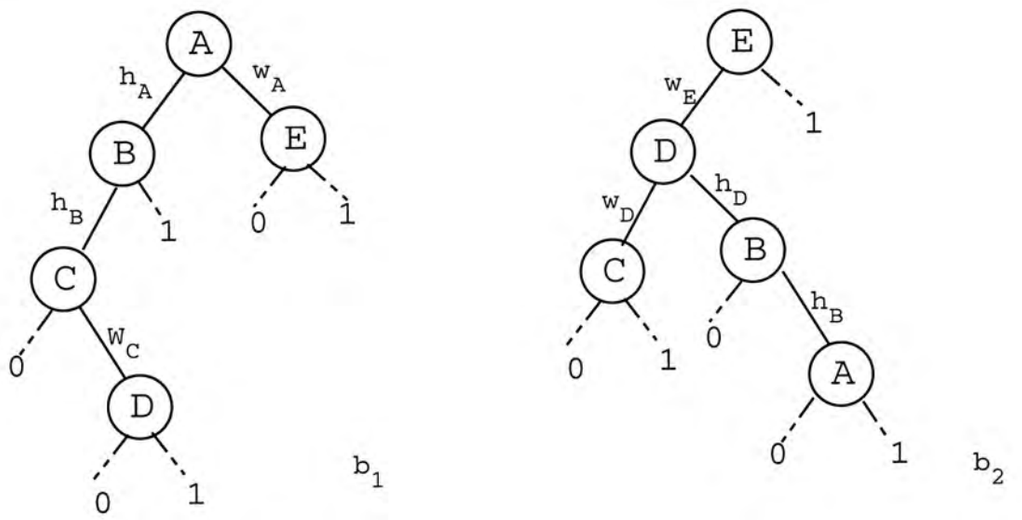


Figure 8.16: The twin binary tree labeled with the corresponding width and height.

Similarly, consider a node D in a binary tree b_2. The right child B of node D corresponds to the module below D. The left child C of node D corresponds to the module on the left of D. Therefore, we have following equations from the binary tree b_2:

$$x'_i = x'_j - w_j$$

where module i is the left child of module j, and

$$y'_i = y'_j - h_j$$

where module i is the right child of module j, x'_i, y'_i is the x-coordinate and y-coordinate of the top right corner of module i respectively. Also, we have the following constraints:

$$x'_i - x_i \geq w_i$$

and

$$y'_i - y_i \geq h_i$$

for module i. Figure 8.16 illustrates the twin binary tree labeled with correct widths and heights.

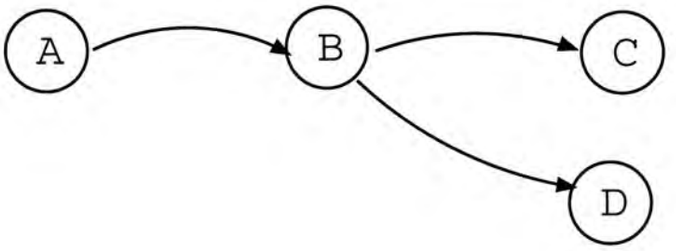


Figure 8.17: The vertical constraint graph from the example in Figure 8.16

The vertical constraint graph and horizontal constraint graph can be constructed from the twin binary tree. In b_1, if node A is the left child of its parent B, there is an edge from B to A in the vertical constraint graph. If A is not the left child of its parent, then we have to search for a node along the path from A to the root which is a left child of its parent. For example, in b_1 of Figure 8.16, node D is not the left child of its parent, and we will search along the path towards the root until we find node C which is the left child of its parent and an edge will then be added from B to D in the vertical constraint graph. A similar method is applied to b_2 except that the direction is reversed. If a node A is the right child of its parent B, we will add an edge from A to B in the vertical constraint graph. Figure 8.17 shows the vertical constraint graph constructed from the twin binary tree in Figure 8.16.

For the horizontal constraint graph, the graph can be constructed using the same method. If node A is the right child of its parent B in b_1, there is an edge from B to A in the horizontal constraint graph. For b_2, if node A is the left child of its parent B, there is an edge from A to B in the horizontal constraint graph. The weight of each edge is the corresponding width or height of its source node module. Figure 8.18 shows the horizontal constraint graph obtained from the twin binary tree in Figure 8.16.

Finally, we will add a source node s and a destination node t to the vertical constraint graph and the horizontal constraint graph. Edges of weight '0' are inserted into the constraint graphs from s to every other vertices with no

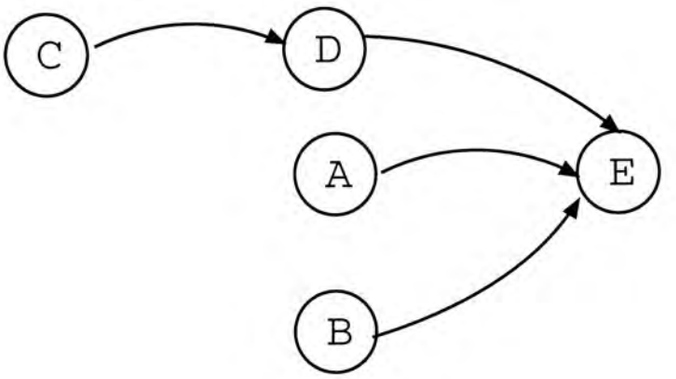


Figure 8.18: The horizontal constraint graph from the example in Figure 8.16

incoming edge; and edges of weight equal to the width or height of its source node module are also inserted from every vertex with no outgoing edge to t. Node s and t correspond to the boundaries of the packing. The coordinates of the lower left corner of module n_i can be obtained from the longest path length from node s to node n_i in the constraint graphs.

The resultant floorplan corresponding to the twin binary tree in Figure 8.16 is shown in Figure 8.20.

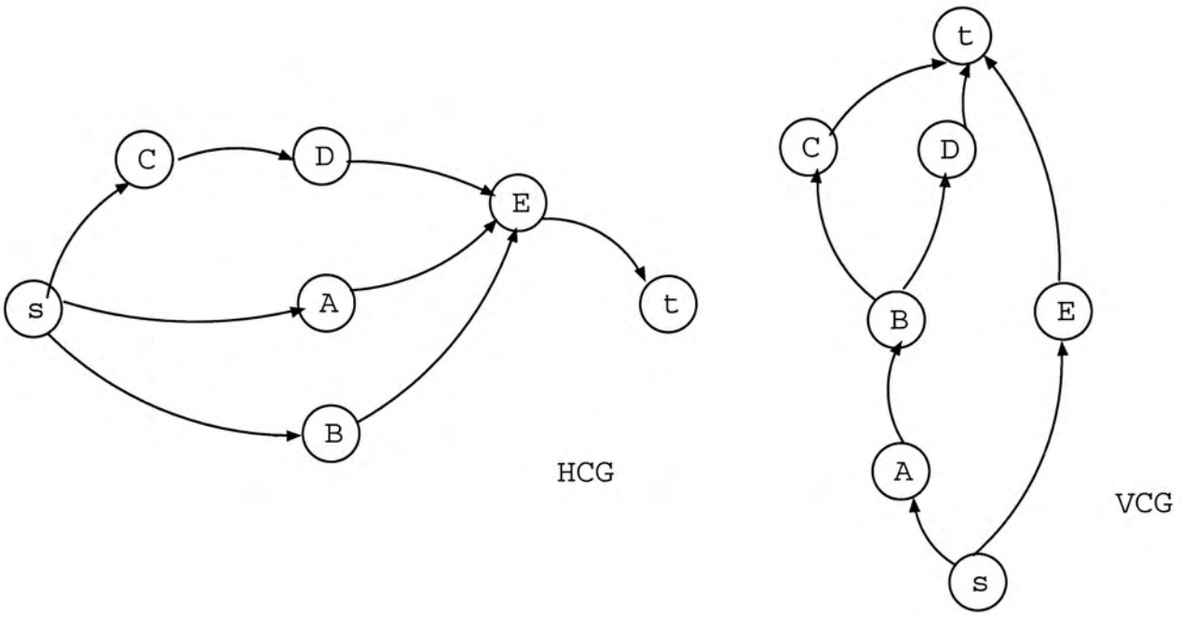


Figure 8.19: The final vertical and horizontal constraint graph for the example in Figure 8.16

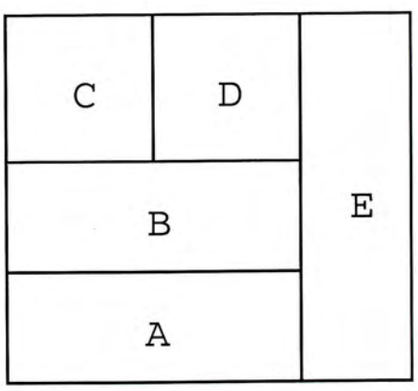


Figure 8.20: The resultant mosaic floorplan of the twin binary tree shown in the Figure 8.16

Chapter 9

Conclusion

9.1 Summary

This thesis gives an overview of floorplanning in CAD of VLSI physical design. Important problems in floorplanning are described. These problems are due to the advance in the deep sub-micron technology and the scaling down of the feature size in the IC technology. Two important issues in floorplanning are scalability and interconnection optimization. There have been many studies and research on these problems. We have studied and investigated the methods of imposing clustering constraints and applying multilevel approach in floorplanning to address these two issues. In additions, we have studied the properties and different representations of non-slicing floorplans in order to achieve a better understanding of this important problem.

A technique to handle clustering constraints is implemented on top of a slicing floorplanner. An algorithm is devised to find all the neighboring modules of a target module from the Polish expression. All the result packings satisfy the given clustering constraints and the deadspace is small on average.

Multilevel approach is introduced to floorplan design. We have implemented an interconnect driven floorplanner by applying the multilevel technique. The interconnect cost is optimized in the clustering phase of the multilevel process and the placement of the modules is performed in the refinement

phase. The applicability of this approach is supported by the promising experimental results in which the interconnect cost is reduced significantly and the runtime is shortened.

A study on non-slicing floorplan is given in the last chapter. We have analyzed different floorplan representations in terms of complexity and representation power. A proof is given to show that for a non-slicing floorplan there exists at least one module (or supermodule) with four T-junctions of different orientations at its corners. A mosaic floorplan representation call twin binary tree has been proposed recently. We have devised an efficient algorithm to generate pairs of valid twin binary trees and to convert a pair of such trees to its corresponding packing.

Bibliography

[AHK98] Charles J. Alpert, Jen-Hsin Huang, and Andrew B. Kahng. Multilevel circuit partitioning. *IEEE Transactions on Computer-Aided Design*, 17(8), August 1998.

[CCKS00] Tony F. Chan, Jason Cong, Tianming Kong, and Joseph R. Shinnerl. Multilevel optimization for large-scale circuit placement. In *Processing of the International Conference on Computer-Aided Design*, pages 171 –176, 2000.

[CCWW00] Y. C. Chang, Y. W. Chang, G. M. Wu, and S. W. Wu. B*-trees: A new representation for non-slicing floorplans. In *Proceedings of the 37th ACM/IEEE Design Automation Conference*, pages 458–463, 2000.

[CHMR91] J. P. Cohoon, S. U. Hegde, W. N. Martin, and D.S. Richards. Distributed genetic algorithms for the floorplan design problem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(4):483–492, April 1991.

[CKP99] Jason Cong, Tianming Kong, and David Zhigang Pan. Buffer block planning for interconnect-driven floorplanning. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 358–363, 1999.

[CZY+99] H. M. Chen, H. Zhou, F. Y. Young, D. F. Wong, Hannah H. Yang, and Naveed Sherwani. Integrated floorplanning and interconnect planning. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, 1999.

[FM82] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th ACM/IEEE Design Automation Conference*, pages 175–181, 1982.

[FMK97] K. Fujiyoushi, H. Murata, and M. Kaneko. Vlsi/pcb placement with obstacles based on sequence-pair. In *Proceedings of International Symposium on Physical Design*, pages 26–31, 1997.

[GCY99] P. N. Guo, C. K. Cheng, and T. Yoshimura. An O-tree representation of non-slicing floorplan and its applications. In *Proceedings of the 36th ACM/IEEE Design Automation Conference*, pages 268–273, 1999.

[HHC+00] Xianlong Hong, Gang Huang, Yici Cai, Jiangchun Gu, Sheqin Dong, Chung-Kuan Cheng, and Jun Gu. Corner block list: An effective and efficient topological representation of non-slicing floorplan. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 8–12, 2000.

[Kah00] Andrew B. Kahng. Classical floorplanning harmful? In *Processing of International Symposium on Physical Design*, pages 207 – 213, 2000.

[KAKS99] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: Applications in vlsi domain. *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, 7(1), March 1999.

[KD98] M. Z. Kang and W. W. M. Dai. Arbitrary rectilnear block packing based on sequence pair. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 259–266. IEEE Computer Society Press, 1998.

[KK95] G. Karypis and V. Kumar. Multilevel graph partitioning schemes. In *Processing of International Symposium on Physical Design*, pages 113 – 122, 1995.

[KL70] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J.*, 49(2):291–307, Feb 1970.

[Knu93] Danald E. Knuth. *The Art of Computer Programming Vol 1: Fundamental Algorithms.* Addison-Wesley, 2 edition, 1993.

[LLWW01] Jianbang Lai, Ming Shiun Lin, Ting-Chi Wang, and Li-C. Wang. Module placement with boundary constraints using the sequence-pair representation. In *Proceedings of IEEE Asia South Pacific Design Automation Conference*, pages 515–520, 2001.

[LW01] Minghorng Lai and D. F. Wong. Slicing tree is a complete floorplan representation. In *Processing of Design Automation and Test Conference in Europe*, pages 228–232, 2001.

[MDH+01] Yuchun Ma, Sheqin Dong, Xianlong Hong, Yici Cai, Chung-Kuan Cheng, and Jun Gu. Vlsi floorplanning with boundary constraints based on corner block list. In *Proceedings of IEEE Asia South Pacific Design Automation Conference*, pages 509–514, 2001.

[MFK98] H. Murata, K. Fujiyoshi, and M. Kaneko. VLSI/PCB placement with obstacles based on sequence-pair. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(1):60–68, January 1998.

[MFNK95] Hiroshi Murata, Kunihiro Fujiyoshi, Shigetoshi Nakatake, and Yoji Kajitani. Rectangle-packing-based module placement. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 472–479, 1995.

[MK98] H. Murata and Ernest. S. Kuh. Sequence-pair based placement method for hard/soft/pre-placed modules. In *Proceedings of International Symposium on Physical Design*, pages 162–172, 1998.

[NFMK96] S. Nakatake, K. Fujiyoshi, H. Murata, and Y. Kajitani. Module placement on BSG-structure and IC layout applications. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 484–493, Washington, November 10–14 1996. IEEE Computer Society Press.

[Ott82] R. H. J. M. Otten. Automatic floorplan design. In *Proceedings of the 19th IEEE/ACM International Conference on Computer-Aided Design*, pages 261 – 267, 1982.

[PBLC00] Yingxin Pang, Florin Balasa, Koen Lampaert, and Chung-Kuan Cheng. Block placement with symmetry constraints based on the o-tree non-slicing representation. In *Proceedings of the 37th ACM/IEEE Design Automation Conference*, pages 464–467, 2000.

[She99] N. Sherwani. *Algorithms for VLSI Physical Design Automation.* Kluwer Academic, 3 edition, 1999.

[SSK00] Probir Sarkar, Vivek Sundararaman, and Chaeng-Kok Koh. Routability-driven repeater block planning for interconnect-centric floorplanning. In *Proceedings of the International Symposium on Physical Design*, pages 186–191, 2000.

[SSR91] S. Sutanthavibul, E. Shargowitz, and J. Rosen. An analytical approach to floorplan design and optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(6):761–769, June 1991.

[SYTB95] S. M. Sait, H. Youssef, S. Tanvir, and M. S. T. Benten. Timing influenced general-cell genetic floorplanner. In *Proceedings of the ASP-DAC '95/CHDL '95/VLSI.*, pages 135–140, 1995.

[TTW00] Xiaoping Tang, Ruiqi Tian, and D. F. Wong. Fast evaluation of sequence pair in block placement by longest common subsequence computation. In *Processing of Design Automation and Test Conference in Europe*, 2000.

[TW00] Xiaoping Tang and D. F. Wong. Planning buffer locations by network flows. In *Proceedings of the International Symposium on Physical Design*, pages 180–185, 2000.

[TW01] Xiaoping Tang and D. F. Wong. Fast-sp: A fast algorithm for block placement bsed on sequence pair. In *Proceedings of IEEE Asia South Pacific Design Automation Conference*, pages 521–526, 2001.

[VNLG95] G. Vijayan, V. Narayananan, D. LaPotin, and R. Gupta. PEPPER: A timing driven early floorplanner. In *International Conference on Computer Design*, pages 230–235, Los Alamitos, Ca., USA, October 1995. IEEE Computer Society Press.

[WA98] S. Wichlund and E. J. Aas. On multilevel circuit partitioning. In *Processing of the International Conference on Computer-Aided Design*, pages 505 – 511, 1998.

[WL86] D. F. Wong and C. L. Liu. A new algorithm for floorplan design. In ACM/IEEE, editor, *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pages 101–107, Las Vegas, NV, June 1986. IEEE Computer Society Press.

[WLL88] D. F. Wong, H. W. Leong, and C. L. Liu. *Simulated Annealing for VLSI Design.* Kluwer Academic, 1988.

[YCCG01] Bo Yao, Hongyu Chen, Chung-Kuan Cheng, and Ronald Graham. Revisiting floorplan representations. In *Processing of International Symposium on Physical Design*, pages 138–143, 2001.

[YCLW00] F. Y. Young, Chris C. N. Chu, W. S. Luk, and Y. C. Wong. Floorplan area minimization using lagrangian relaxation. In *Proceedings of International Symposium on Physical Design*, 2000.

[YSAF95] H. Youssef, S. M. Sqit, and K. J. Al-Farra. Timing influenced force directed floorplanning. In *Proceedings of the EURO-VHDL, Proceedings EURO-DAC '95*, pages 156–161, 1995.

[YW98] F. Y. Young and D. F. Wong. Slicing floorplans with preplaced modules. In *Proceedings IEEE International Conference on Computer-Aided Design*, pages 252–258, 1998.

[YW99a] F. Y. Young and D. F. Wong. Slicing floorplans with boundary constraints. In *Proceedings of IEEE Asia South Pacific Design Automation Conference*, pages 17–20, 1999.

[YW99b] F. Y. Young and D. F. Wong. Slicing floorplans with range constraints. In *Proceedings of the International Symposium on Physical Design*, pages 97–102, 1999.

[ZDH+01] S. Zhou, S. Q. Dong, X. L. Hong, Y. C. Cai, C. K. Cheng, and J. Gu. ECBL: An extended corner block list with solution space

including optimum placement. In *Proceedings of International Symposium on Physical Design*, 2001.

Appendix A

Clustering Constraint Data Set

A.1 ami33

A.1.1 One cluster

Data Set	Cluster (module's number)
ami33-cc1	5 7 8 11 15 19 22
ami33-cc2	5 7 8 26 29 30 31
ami33-cc3	11 19 20 26 29 30 31

A.1.2 Multi-cluster

Data Set	Number of Clusters	Cluster (module's number)	
ami33-mc1	3	1) 5 7 11 15	2) 19 22 25 29
		3) 14 27 30	
ami33-mc2	4	1) 5 7 8	2) 11 14 18
		3) 21 24 27	4) 29 30
ami33-mc3	5	1) 5 7 11	2) 8 14
		3) 21 27	4) 24 29
		5) 18 30	

A.2 ami49

A.2.1 One cluster

Data Set	Cluster (module's number)
ami49-cc1	6 7 8 9 10 11 12 13 14 15
ami49-cc2	16 17 18 19 20 21 22 23 24 25
ami49-cc3	26 27 28 29 30 31 32 33 34 35

A.2.2 Multi-cluster

Data Set	Number of Clusters	Cluster (module's number)	
ami49-mc1	3	1) 6 7 8 9 10 11	2)12 13 14 15 16
		3) 17 18 19 20 21	
ami49-mc2	4	1) 6 7 8 9	2) 10 11 12 13
		3) 14 15 16 17	4) 18 19 20 21
ami49-mc3	5	1) 6 7 8 9	2) 11 12 13
		3) 14 15 16	4) 17 18 19
		5) 20 21 22	

A.3 playout

A.3.1 One cluster

Data Set	Cluster (module's number)
playout-cc1	1 2 5 6 7 10 11 12 13 16 18 19
playout-cc2	21 23 24 25 26 27 28 29 31 32 33 34
playout-cc3	40 42 43 44 45 46 48 49 50 51 52 53

A.3.2 Multi-cluster

Data Set	Number of Clusters	Cluster (module's number)	
playout-mc1	3	1) 1 2 5 6 7 10 11	2) 13 16 18 19 21 22 26
		3) 27 28 31 33 36 37	
playout-mc2	4	1) 1 2 5 6 7	2) 10 11 13 16 18
		3) 19 21 22 26 27	4) 28 31 33 36 37
playout-mc3	5	1) 1 2 5 6	2) 7 10 11 13
		3) 16 18 19 21	4) 22 26 27 28
		5) 31 33 36 37	

Appendix B

Multilevel Data Set

B.1 data_100

Number of Modules = 100

Minimum Aspect Ratio = 0.5

Maximum Aspect Ratio = 2.0

Size of modules:

49849	98183	75071	95600	61093	85791	94347	85135
78336	94340	62177	61581	41382	99532	94960	47531
17784	41769	81607	43630	92959	30494	32491	13738
86207	66031	42056	7716	20099	79035	86148	96014
38409	35077	37906	56472	5299	69761	76492	54874
56713	8817	77678	70900	81432	73361	74732	97500
72091	33468	62268	27015	30187	83107	11759	99749
39775	52890	65317	70480	96060	6462	26557	45240
37057	88023	92981	42521	80755	39158	91122	16340
22721	82037	59121	21774	76069	36677	29755	11156
8744	54602	66333	77751	44148	76537	93282	46083
77292	63238	99290	78738	27599	75777	13260	89496
53376	9258	34519	20102				

Number of Pins = 48

Distribution of Pins: Evenly distributed on the boundaries of the chip.

Number of Net = 1888

Distribution of Nets:

2 pins net	80%
3 pins net	10%
4 pins net	5%
5 pins net	2%
6 pins net	2%
7 pins net	1%

B.2 data_200

Number of Modules = 200

Minimum Aspect Ratio = 0.5

Maximum Aspect Ratio = 2.0

Size of modules:

49512	227393	37432	14075	31679	263100	70330	99589
563849	68630	7696	45066	36529	55383	67717	24267
61930	86274	56472	71641	66955	71334	70564	60843
71862	55045	253651	31469	51235	89834	80542	68640
47478	61358	67609	525336	8679	55534	11501	65751
52472	34681	39490	99456	515465	220744	14244	70758
33803	67233	73427	65442	14616	38343	97917	14938
45814	298641	99873	49070	6216	48042	7296	14793
57940	18350	37940	9219	27676	96835	34779	74304
96375	37068	27264	25307	31642	75425	21993	48884
22151	64927	56091	62132	37254	11159	9282	44031

18633	88938	39397	243075	86149	47893	592217	50642
43424	56213	21036	69840	97930	19091	555659	91701
39575	229577	98443	83352	25528	15693	53856	25686
22381	61616	31077	57560	60267	99223	38337	264629
99181	27568	99968	77475	46112	33769	45347	90047
39805	52184	23737	61923	76325	565236	31892	573411
583072	18901	61245	61484	98607	89696	74416	9399
87174	8532	54656	5445	24014	64099	64306	18360
67196	53917	68570	74507	50801	91963	42561	35542
89985	44949	94307	43378	290768	41243	67751	67688
54875	24031	56680	73997	263831	41454	27027	53531
47481	10228	90674	59992	12481	32352	45799	5563
204886	40683	48937	78682	507568	298473	12149	59532
28026	80425	25021	25664	15040	18361	85714	59836

Number of Pins = 42

Distribution of Pins: Evenly distributed on the boundaries of the chip.

Number of Net = 2388

Distribution of Nets:

2 pins net	80%
3 pins net	10%
4 pins net	5%
5 pins net	2%
6 pins net	2%
7 pins net	1%

B.3 data_300

Number of modules = 300

Minimum aspect ratio = 0.5

Maximum aspect ratio = 2.0

Size of modules:

597414	42925	79706	38579	54275	19606	5713	68993
62969	92758	57253	23444	35047	59669	15719	11233
65746	83647	90266	45558	7135	25518	50242	38166
44181	53308	41576	32945	41132	8975	50861	94472
30015	65421	515948	28767	85220	63182	5930	33658
66813	6309	51035	31224	93275	88451	278662	63951
224079	11311	15293	73073	48201	58508	556329	267487
68567	91902	76272	34718	44344	44539	50239	41597
94646	84437	21610	63363	18501	8024	48360	13346
36455	61679	50103	78333	86074	29257	229049	68987
79076	98038	10905	45842	24643	65113	52936	30680
85148	70849	26719	90154	92351	14476	87445	39924
65999	54065	74159	71082	99893	69418	53029	97009
28640	56625	93609	58748	70650	94271	598820	73482
86588	41824	268538	84172	44864	86723	252851	34368
59685	78665	8260	40730	13400	89708	30279	12438
74167	17000	99336	71500	72423	43311	60833	5212
69032	89748	88047	96094	17810	53467	42025	50202
56919	81230	36676	8935	82033	54082	81624	26849
11282	30152	53178	64499	34441	95918	40547	15313
30352	82345	72061	61981	89056	93799	284082	88473
34533	92260	581304	57420	81593	77140	51327	99468

76157	38636	51123	62021	72277	553566	20668	89173
79545	90360	57964	28829	29472	555051	53203	88561
44792	58746	92183	227273	32500	14742	51196	54215
28403	88984	23337	58847	253697	292499	29846	59080
44260	71750	85043	74265	42367	548255	51484	26289
72658	82357	543143	47204	59430	87671	8766	25083
80732	502550	54952	51089	96214	32803	49125	40272
230333	15927	76203	86166	240266	21295	9631	74996
48751	93018	97356	29152	76092	62595	56401	71566
62686	7034	15892	33694	547689	560142	20836	46305
76241	86996	22817	90198	564816	70010	55513	14415
33989	20639	15780	80055	63232	58668	34054	25990
63252	66786	44780	55643	75434	72715	95607	53044
16905	80612	65898	93022	73484	21544	16489	6758
528737	68551	75741	76492	64854	32128	97258	21982
54513	90135	80381	18392				

Number of Pins = 64

Distribution of Pins: Evenly distributed on the boundaries of the chip.

Number of Net = 2888

Distribution of Nets:

2 pins net	80%
3 pins net	10%
4 pins net	5%
5 pins net	2%
6 pins net	2%
7 pins net	1%

B.4 data_400

Number of modules = 400

Minimum aspect ratio = 0.5

Maximum aspect ratio = 2.0

Size of modules:

83841	94654	82060	8914	23925	27070	31236	296471
60089	73214	542710	12909	57208	38003	54217	225471
49445	86582	19044	279003	74646	87587	12570	84632
19404	73132	517704	81661	71442	9959	62882	77301
49511	47385	11392	54169	82660	31978	23798	40505
56368	38159	95949	574061	40114	26773	28136	12714
24312	16620	17520	277075	98361	62851	98207	9746
67325	549357	11011	96448	11330	17007	19834	48464
84679	11778	59105	44058	51646	93056	10079	81271
58364	48696	20304	77430	95808	47947	61734	17529
25897	53413	46452	37637	33863	23611	529530	49472
8958	269130	276831	73456	13074	509347	55772	563808
66583	21630	43769	78154	55320	6241	14688	41450
20722	578033	14416	16259	52489	88389	72704	25064
290743	28661	62579	57168	10342	17176	10924	32614
525948	84226	88679	60303	22029	69460	64924	67291
47954	93429	93185	47621	89013	38491	43133	94299
14063	74248	33129	47004	97004	62904	81323	31254
56029	81660	13446	18164	564292	63849	81791	57066
20454	275410	78915	93984	98475	8661	54464	28533
70941	73320	96300	99992	88822	53476	72094	25173
13901	29253	54020	54639	526986	5813	81832	98922

67908	48281	99136	31222	7456	67756	81666	503539
69761	35962	88277	54148	85670	98042	72834	8235
86147	17825	29825	45404	37342	99318	519684	50386
94628	41483	82629	28583	41532	27598	88367	70249
78419	47324	5306	92628	22625	5116	94523	94641
47048	90940	86718	78381	53831	49696	45082	89662
55540	39770	44350	95709	56872	52180	22994	64021
96135	534470	538864	71734	9728	75196	21194	81043
95246	47583	43313	224953	94181	11896	40841	47516
32742	8742	83408	12119	26225	92927	41286	36154
7083	97975	7945	21707	18915	48963	53596	45030
21850	44163	86385	77841	76919	89406	32914	63217
96039	53303	76441	63727	18959	15330	83839	53306
53913	23341	72963	82580	596925	78935	17274	97786
55444	97629	541108	8494	64620	60788	25785	23447
23956	22761	77790	96892	89299	42024	82130	58897
46078	34028	34091	217398	232381	11664	46423	42760
73766	65136	62228	543568	10066	507077	47489	74953
30267	547820	73848	52264	97210	41500	30441	29174
58795	62461	32641	94675	10435	68461	27634	66973
94706	82241	232058	13951	98444	76235	275279	9642
34992	76589	63409	44091	79857	212082	23763	39981
61740	21164	37254	93601	97700	96054	17064	249978
86663	45947	42444	584686	9115	41660	75406	25011
80248	5788	58446	65356	70314	83355	25542	26052
37882	43079	90828	51126	21435	39471	86937	15940
20420	54362	17252	20637	75564	88434	71993	23428
78379	17667	64949	31689	17986	75873	68150	15062

Number of Pins = 64

Distribution of Pins: Evenly distributed on the boundaries of the chip.

Number of Net = 3388

Distribution of Nets:

2 pins net	80%
3 pins net	10%
4 pins net	5%
5 pins net	2%
6 pins net	2%
7 pins net	1%

B.5 data_500

Number of modules = 500

Minimum aspect ratio = 0.5

Maximum aspect ratio = 2.0

Size of modules:

70268	46383	84415	79249	93575	34533	56758	23949
57208	53670	84034	68566	90972	40959	248764	35223
87656	7419	54440	67740	48375	29617	518006	43746
88565	8099	91227	10013	86973	84632	45626	48162
92769	42918	49372	35190	22763	84414	75080	79077
9506	85589	97086	49004	60272	67821	46766	84672
9162	523729	38856	23649	77501	37906	63163	267163
30147	30120	16623	87941	89670	95129	46688	527762
20416	23564	81681	22748	54519	268368	584046	93180
41513	17562	14316	94211	506009	582808	27750	94867
64369	521884	522579	58654	33832	64981	7581	68514

242654	55298	91975	21390	35265	18689	55814	47614
290308	56576	38324	558294	583008	272025	31824	48353
83170	86660	79479	87214	58501	51676	27748	78860
65815	37676	20364	21101	99046	34083	17673	25473
12346	287911	49212	99568	22144	21740	69858	87034
23741	5603	75667	42955	27765	22915	79464	76510
5960	10857	576198	72341	20621	12305	555138	82090
90215	27393	46552	573616	81958	87284	29625	20669
43892	53527	501033	76775	93616	26714	59537	74578
30619	10928	83844	22871	55715	15813	35543	77202
50621	27685	76832	60299	264197	21686	33969	257926
47150	42635	49730	14844	74160	27533	49162	13960
47725	78757	53244	18137	7523	57723	80112	8377
527151	76066	23488	44421	10964	11787	65918	550840
29270	72759	227321	23983	58763	83446	91690	266233
14471	46394	559129	58968	72591	27874	45903	42077
34666	94209	64671	31678	28543	5112	44440	39075
83110	78277	30416	63422	58367	75491	37542	18194
60219	24692	18411	17275	20189	84711	46112	90304
74300	79793	5684	18925	32479	79592	34013	85657
90098	76942	47343	5341	96648	20511	547996	65453
63037	292374	7493	36059	58654	81534	81902	22775
22050	93624	27136	26827	585032	54273	46780	45145
88896	9058	23169	590395	53719	96471	53562	235977
46842	72020	37637	94194	85067	91338	39924	17189
96610	18110	81379	569693	63444	8647	97999	60234
62476	39105	541195	33179	224807	83618	84550	17824
36412	22050	44705	510633	554582	20271	53613	73165
27015	21197	213614	25473	9845	92092	74153	82062

68245	19807	65489	544136	39397	35978	23081	298904
96420	82396	42412	12478	32987	23357	61168	37800
74224	94805	88250	69480	74788	21833	14857	75190
64411	89338	81193	13081	28577	81973	13925	52577
92621	19006	94077	51947	11193	97297	62864	10196
51866	21329	223798	43077	53431	33954	17816	56458
90221	76773	92157	91228	31256	69182	94029	29389
29962	13423	13861	23534	54403	579709	31246	31024
35446	65737	91996	90489	6768	55758	289916	57861
14621	39819	14030	88018	54209	299208	40079	95058
71724	37683	16290	49341	56294	39703	17743	72660
34490	66677	21080	74238	271568	60158	81909	29729
78193	6937	80891	42996	19217	35157	41375	46166
21905	52858	11334	587295	85026	10756	45460	26977
97378	69616	19443	15293	84521	579223	57793	48752
15923	30750	58986	13192	21188	27221	257305	83685
27821	69421	70529	60943	35568	596960	40703	68160
80148	502716	36728	84610	94098	24151	586685	59416
82151	50670	94363	8311	81137	543098	30416	67027
6105	265723	82345	81469	79569	38551	9195	21062
79838	75438	12758	5757	81449	5940	60974	79203
63632	585727	5324	509270	55494	525374	47877	523088
21112	53180	70783	86904				

Number of Pins = 72

Distribution of Pins: Evenly distributed on the boundaries of the chip.

Number of Net = 3888

Distribution of Nets:

2 pins net	80%
3 pins net	10%
4 pins net	5%
5 pins net	2%
6 pins net	2%
7 pins net	1%

Publications

Full Length Conference Papers

- W. S. Yuen and F. Y. Young, "Slicing Floorplan with Clustering Constraints" in *Proceeding of IEEE Asia South Pacific Design Automation Conference*, pp. 503-508, 2001.

- W. S. Yuen and F. Y. Young, "Scalable and Interconnect Driven Multilevel Floorplanner" in progress.