National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

UNIVERSITY OF ALBERTA

WIRING SPACE OPTIMIZATION IN INTEGRATED CIRCUIT FLOORPLANS

BY

YUEJIUN YANG ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science.

DEPARTMENT OF ELECTRICAL ENGINEERING

Edmonton, Alberta

FALL, 1993

National Library of Canada

Acquisitions and Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada

Direction des acquisitions et des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-88359-6

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Yuejiun Yang

TITLE OF THESIS: Wiring Space Optimization in IC Floorplans

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1993

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

[illegible]

#126, Royal Terrace
106 Street 27 Avenue
Edmonton, Alberta
Canada

Date: Oct. 7. 1993

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Wiring Space Optimization in Integrated Circuit Floorplans** submitted by **Yuejiun Yang** in partial fulfillment of the requirements for the degree of **Master of Science**.

[illegible]

Supervisor: Dr. J. T. Mowchenko

[illegible]

Dr. B. F. Cockburn

[illegible]

Dr. J. Culberson

Date: Oct. 7, 1993

Acknowledgements

I would like to express my special thanks to my supervisor, Dr. Jack Mowchenko, for his valuable guidance through out the project. Also, I would like to thank Mr. Norman Jantz for his patient help and technical support.

As well, I would like to acknowledge the financial support of the Natural Sciences and Engineering Research Council. The Canadian Microelectronics corporation is also acknowledged for providing the computers and the Cadence VLSI CAD package.

Also, I would like to thank my parents and my wife Jie for their understanding and support.

Abstract

Floorplanning a VLSI circuit is a very complicated problem. Trade-offs have to be made between the quality of the floorplan and the execution time required to produce the floorplan. One of the major concerns of floorplanning is to minimize the total chip area. In this thesis, two algorithms aimed at minimizing total chip area are presented. The first one is a branch-and-bound algorithm, and the other is a heuristic algorithm derived from the branch-and-bound algorithm. Although the branch-and-bound algorithm always gives the optimal solution to the problem, the time complexity of the algorithm is such that it is only practical for small circuits. On the other hand, the heuristic algorithm does not necessarily give an optimal solution to the problem, but it can give a reasonably good solution in a practical amount of time. One of the key features of the algorithms is that the wiring is taken into consideration during the floorplanning process. A new method for estimating the wiring space, called the *space profile* method, is also proposed in this thesis.

The effect of the algorithms is evaluated by applying them to some benchmark circuits. The floorplans, both before and after optimization, are routed to evaluate the effect of the optimizing algorithms and the results are promising.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the size and density of VLSI (Very Large Scale Integration) increases, the task of designing an IC (Integrated Circuit) becomes more and more complicated. An efficient way of handling the complexity of the circuit is hierarchical design. In hierarchical design the whole circuit is divided into several functional units, such as a multiplier, a divider, etc.. Then these functional units are in turn divided into simpler units. This process repeats until the functional units are simple enough to be handled easily; these functional units are referred to as *modules*.

The task of integrated circuit design can be divided into two parts: behavioral design and physical design. *Behavioral* design takes as its input the functional specifications of the circuit and realizes these functional specifications with functional units such as transistors, logic gates, etc.. The output of behavioral design is a description of the circuit, for example, a circuit schematic and a performance specification of the circuit. *Physical* design takes the circuit description produced by behavioral design as its input, and produces a physical realization of the circuit. The output of physical design is the mask data for fabricating the circuit.

The physical design of a VLSI circuit can also be referred to as the *layout design* of the circuit. The task of laying out an integrated circuit consists of placing the circuit modules in a two dimensional finite space and of interconnecting the module terminals according to the circuit schematic. The goal of this process is to complete the placement and interconnection of the design in the smallest possible area while satisfying a set of design constraints. One way to address the problem of laying out a VSLI is to break the problem into three phases: partitioning, floorplanning/placement, and routing.

Given a circuit, the first problem to be solved is partitioning. *Partitioning* is the task of decomposing the circuit into groups of modules such that a certain objective function is optimized. The objective function takes into account such factors as the amount of interconnection between the groups, the difference in the numbers of modules in each group, etc.. During partitioning, modules are grouped and assigned to blocks with fixed (or variable) dimensions. The main purpose of partitioning is to decrease the complexity of floorplanning and placement.

After a circuit has been partitioned, the next problem to be solved is floorplanning and/or placement. Floorplanning and placement are closely related problems and sometimes these two words are used interchangeably. In fact, floorplanning can be interpreted as a generalization of the placement problem. In order to define floor planning problem, we need the following definitions:

- A *terminal* of a module is a point on the module that is used to connect an input/output signal of the modules to other modules. The position of a terminal on a module can be either fixed or floating.
- A *net* is a set of terminals that are electrically connected together.
- A *netlist* of a circuit defines all the interconnections between the terminals.
- In most cases, a module is designed in a rectangular area. The rectangle can be either fixed or flexible. The *shape function* of a module gives all possible dimensions the rectangle can take.
- The *aspect ratio* of a module is the ratio of the module's width to its height.

Generally, floorplanning can be defined as follows:

Given:

- The netlist of the circuit;
- A set of design constraints;
- Possibly a circuit partition;
- The shape functions of the modules;
- Possible terminal positions on the modules.

Determine:

- Locations of the modules on the floorplan;
- Aspect ratios of the modules;
- Positions of the terminals on corresponding modules.

Minimizing: An objective function.

Satisfying: The design constraints.

The objective function gives a measure of the quality of the floorplan. It can take into consideration such factors as total area of the floorplan, total wire length of the interconnection, and the time delay along critical paths, etc..

As can be seen from the above definition, there can many kinds of floorplanning problems, depending on the given information. For example, the dimension of modules can be either fixed or flexible; the position of terminals on the modules can be either fixed or floating; the modules can be either partitioned or not, etc..

The last step in laying out a VLSI design is routing. *Routing* is the task of implementing the interconnections between the modules in the circuit. The goal of this step is to optimize an objective function while satisfying the interconnection requirements and design constraints. The objective function gives a measure of the quality of the routing. Specifically, it takes into consideration such factors as the total chip area, the total wire length of the interconnections, and the time delay along critical paths, etc.. It should be noted that the position of the modules could be moved during the routing process. For example, if there is a lot of empty space after the floorplan has been routed, the router may try to squeeze the modules together to reduce the total chip area. On the other hand, if the empty space left over during the floorplanning process is not enough to accommodate the wiring, the router may move the modules apart in order to make enough room for routing.

In this thesis, we are mainly concerned with integrated circuit floorplanning. In our problem, it is assumed that the dimension of the modules is fixed. Also it is assumed that the positions of the terminals on the modules are fixed.

Floorplanning is a very complicated task. Many algorithms have been proposed but few of them can give an optimal solution in a reasonable amount of time. Trade-offs have to be made between the computation time and the quality of the floorplan. As mentioned before, one of the objectives of floorplanning is to minimize the total chip area. In a floorplan there are two kinds of areas: module area and empty area. Module area is the area occupied by the modules of the circuit. This area will be constant once the layout of each module is finished. Some of the empty area left over by floorplanning will later be used for routing, and the rest of the empty area will be left empty after routing, and therefore, is wasted. Some early proposed algorithms took the minimization of the empty space in the floorplan as the only objective. This may not be a good choice because the quality of routing depends heavily on the quality of the floorplan. Fig 1.1 gives an example. In the figure, floorplan A was obtained by considering only the area of the floorplan, while floorplan B was obtained by considering both the floorplan area and the routing of the interconnections between the modules. It can be seen from the figure that although floorplan A is smaller than floorplan B, the total area of floorplan A after routing is larger than that of floorplan B. Also the total wire length of floorplan B is much smaller than that of floorplan A. In the last few years, more and more work has been done that takes routing into consideration during the floorplanning phase.

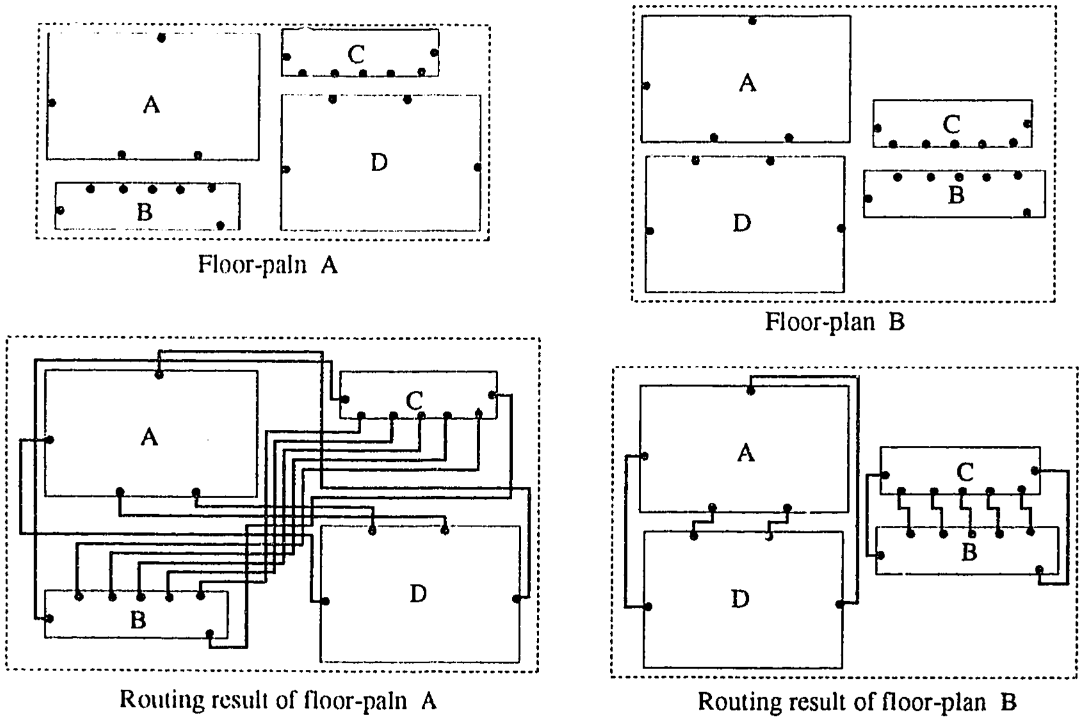

A
C
D
B
Floor-paln A
A
C
B
D
Floor-plan B
A
C
B
D
Routing result of floor-paln A
A
C
B
D
Routing result of floor-plan B

Figure 1.1: The routing results of two floorplans obtained with different strategy

In this thesis, we will present two closely related algorithms which re-arrange a given floorplan such that an estimate of wiring space is minimized. One of the algorithms is a branch and bound searching algorithm. This algorithm always gives the optimal solution to the problem, but the time complexity of the algorithm is such that it is only practical for small circuits. The other algorithm is a heuristic algorithm derived from the branch and bound algorithm. Although the heuristic algorithm does not necessarily give the optimal solution to the problem, it can give a reasonably good solution to the problem in a practical amount of time.

In chapter 2, we give some background knowledge of floorplanning. Some widely used data structures representing a floorplan are explained. Also, some existing algorithms for both floorplanning and wiring space estimating are discussed. In chapter 3, the two new algorithms aimed at minimizing wiring space are presented. The time performance and experimental results of applying the two proposed algorithms to some benchmark circuits are presented and discussed in chapter 4. Conclusions and some possible future work are included in chapter 5.

Chapter 2

Background

The purpose of this chapter is to introduce background knowledge of floorplanning and related topics. In section 1, data structures used to represent a floorplan are explained. Wiring space/length estimating techniques are introduced in section 2. In section 3, we describe some existing floorplanning algorithms. The last section evaluates the existing floorplanning algorithms and a brief introduction to the new algorithms proposed in this thesis.

2.1 Floorplan Data Structures

In order to manipulate floorplans, we need a data structure to represent them. In this section, we introduce two of the most popular data structures for floorplans: the *slicing tree* [24] and the *polar-digraph* [25].

A slicing floorplan is a partitioning of the IC surface into rectangular areas by recursively slicing the surface with vertical and horizontal lines. The partitioning procedure starts with a rectangle representing the whole chip area. This rectangle is split into two sub-rectangles with either a vertical or a horizontal slice. Then, these sub-rectangles are in turn sliced into smaller sub-rectangles. This process continues until there is one rectangle for each module in the circuit. The rectangles that are not further sliced are called *leaf rectangles.* The other rectangles which are further sliced are called *internal rectangles.* Each module in the circuit is assigned to a leaf rectangle.

A convenient way to represent a slicing floorplan is with a slicing tree. A slicing tree is an oriented binary tree in which each node, v, represents a rectangle in the floorplan. The leaf nodes correspond to leaf rectangles, and the internal nodes correspond to internal rectangles. For each internal node, v, its left and right child, v_l and v_r respectively, represent the two sub-rectangles produced by slicing v's rectangle. Node v_l represents v's left sub-rectangle in the case of a vertical slice and the bottom sub-rectangle in the case of a horizontal slice. Similarly v_r represents the right sub-rectangle in the case of vertical slice and the top sub-rectangle in the case of a horizontal slice.

In addition to representing a rectangle in the floorplan, a node v in a slicing tree also represents the set of modules that fall into v's rectangle. Such a set of modules are referred to as a *cluster.* If v is a leaf node, then the corresponding cluster consists of only one module.

A label $D(v)$ is attached to each internal node to describe the slice direction of the node. $D(v) = H$ if the slice is horizontal, and $D(v) = V$ if the slice is vertical. Each leaf node is labeled with the name of the module assigned to the node's rectangle.

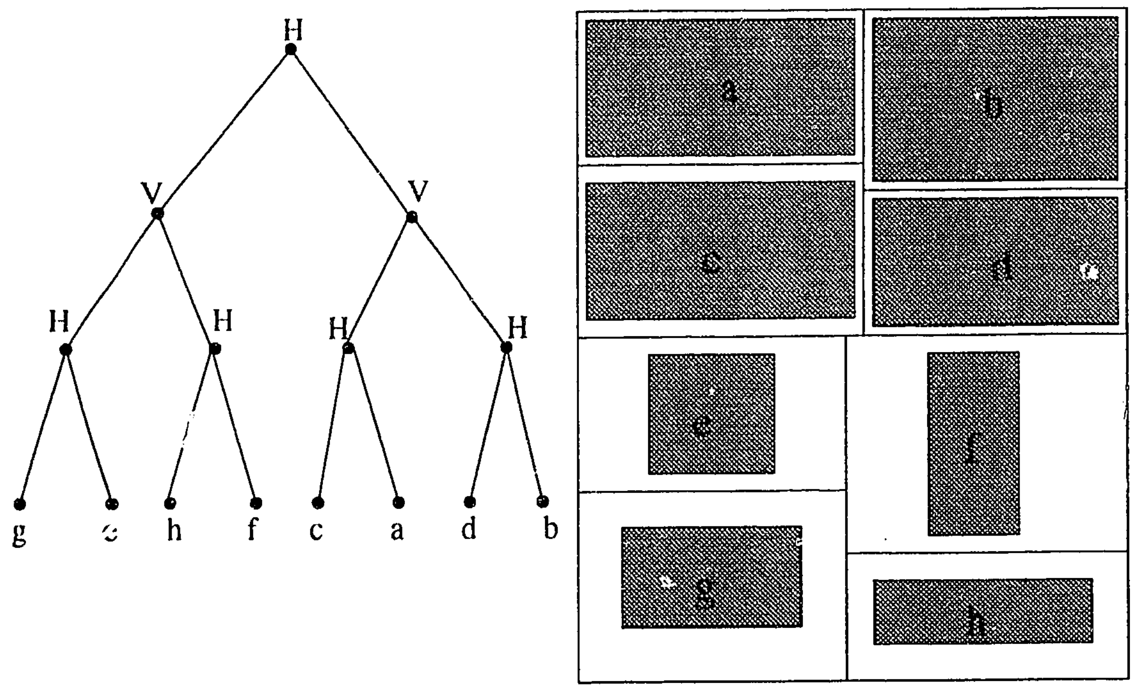


Figure 2.1: An example slicing floorplan and its slicing tree

Figure 2.1 gives an example of a slicing floorplan and the corresponding slicing tree. In the figure the shaded blocks represent the modules in the circuit. Unless otherwise specified, we assume that the modules are always placed at the center of the respective leaf rectangles.

There are several features about the slicing floorplan that make it attractive: (1) The representation of a slicing floorplan is simple and easy to manipulate. (2) It is ideally suited to the hierarchical design style. (3) There are efficient partitioning algorithms that are best suited for slicing floorplans [26, ?]. (4) There are algorithms that can find the optimal shape and the orientation of the modules for a slicing floorplan. (5) The structure of a slicing floorplan makes the problem of routing much easier [28]. One of the major limitations to the slicing structure is that not all floorplans can be represented by a slicing structure [24].

The other commonly used floorplan representation is the *polar-digraph* [25]. Whether the floorplan is a slicing structure or not, it can always be represented by a pair of dual directed graphs: the vertical polar-graph and the horizontal polar-graph. In a floorplan, the whole chip area is partitioned into rectangular areas and each rectangle is assigned a module in the circuit. The vertical polar-graph defines vertical relative positions of the modules while the horizontal polar-graph defines the horizontal relative positions of the modules. Consider the vertical polar-graph as an example. In this graph, the nodes correspond to with horizontal boundaries of the rectangles in a floorplan. In particular, a node corresponds to a single horizontal line which may bound several rectangles that share the same Y coordinate. The node corresponding to the top boundary of the whole chip area is called the source node, and the node corresponding to the bottom boundary of the chip is called the sink node. The rest of the nodes are called internal nodes. A directed edge from node i to node j represents the fact that there exists a rectangle in the floorplan with its top boundary at the location represented by node i and its bottom boundary at the location represented by node j. Each of the rectangles in the floorplan uniquely corresponds to an edge in the polar-graph. The edge is labelled with the name of the module assigned to the rectangle.

The horizontal polar graph is constructed in a similar manner except that the nodes correspond to the vertical boundaries of the floorplan, and the direction of the edges is from left to right. Figure 2.2 gives an example floorplan and its polar-graphs.

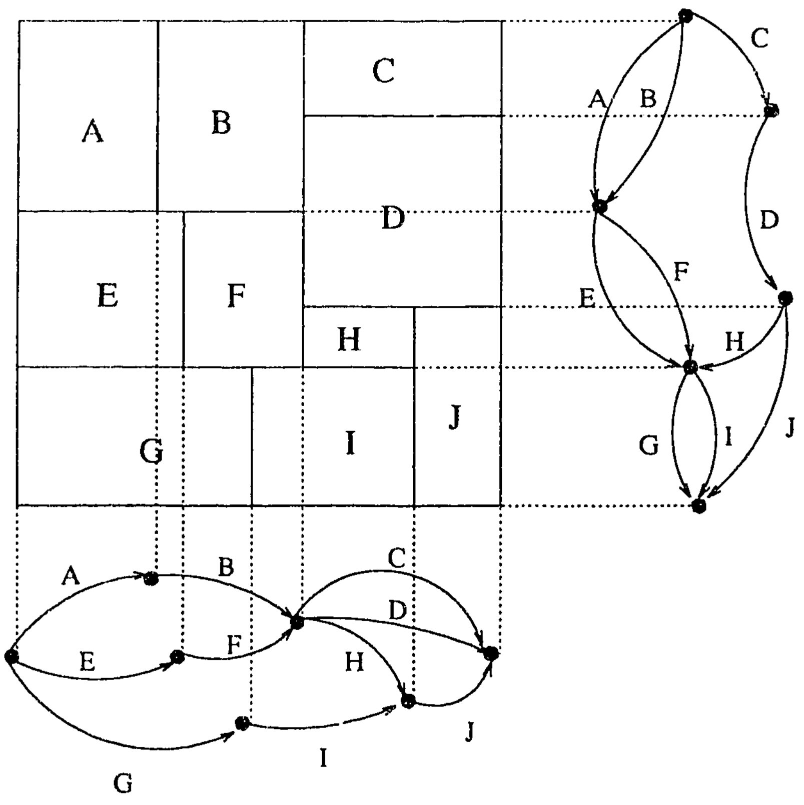


Figure 2.2: An example floorplan and its polar-graphs

The two polar graphs uniquely define a rectangular floorplan. If we label each edge in the pair of polar graphs with the vertical/horizontal dimension of the corresponding rectangle, then the longest path from the source node to the sink node determines the minimum Y/X dimension of the floorplan.

2.2 Wiring Space Estimation

In order to minimize the wiring space during floorplanning, we need to evaluate a floorplan from the routing point of view. It is definitely impractical to actually route each possible floorplan in order to evaluate it. Therefore, we need to analyze the interconnection of the circuit and the floorplan to estimate the wiring space.

Wiring analysis and area estimation is a very complicated task. A compromise has to be made between accuracy and computational complexity. Generally speaking, wiring space estimation can be classified into three categories: *empirical estimation* [29]; *stochastical estimation* [10, 13] and *procedural estimation* [8, 11, 23, 9].

2.2.1 Empirical estimation

Empirical estimation of wiring space is based on the experience of previous designs. Some empirical relationships and the related parameters are drawn from previous experience and are used to analyze the circuit under consideration. The initial empirical work on the subject was made in an unpublished study by E.F.Rent at IBM in the early 1960s [29]. His formulation of what is now called *Rent's Rule* was passed along to researchers both inside and outside IBM. Rent's rule is given by the formula $P = AS \times M^r$, where P is the number of pins in the circuit, AS is the average size of the modules, M is the number of modules in the circuit, and r is a mysterious quantity called the *Rent exponent*. The value of r is determined from previous design experience. Although the wiring space is not explicitly given in the formula, it can be calculated by the formula $WS = AS \times M - \sum_{i=1}^{M} m_i$ where WS is the estimated wiring space, and m_i is the space occupied by module i. Since empirical estimation is based on previous experience, its accuracy depends heavily on the circuits from which the parameters are drawn. Also, the characteristics of the circuit and its floorplan are not taken into consideration. Therefore its applicability is restricted. Usually this type of estimation is used before physical layout in order to evaluate the feasibility of the circuit under development.

2.2.2 Stochastic Estimation

Stochastic methods attempt to construct statistical models of various estimates such as chip area, wire length, wiring space, etc.. Some assumptions are made to simplify the complexity of the problem. Most of the work on this type of estimation has focused on regular structures such as standard cell and gate array design styles.

A typical stochastic method of estimating wiring space in a standard cell layout was proposed by Kurdahi and Parker [10]. In their method, it is assumed that each net has only two terminals. Also, it is assumed that the circuit is first arranged in a single row, and is then folded into multiple rows. Based on these assumptions, wiring area estimation is done first on the single-row structure. Terminals are assumed to be evenly distributed along the entire row. The length of a net is assumed to be a random variable with Poisson distribution. At each position of the row, the expected value of the number of wires crossing the position is calculated, so the channel density at the position is obtained. Then the result is modified based on the configuration of the folded multiple row structure.

The stochastic method is good at estimating layout characteristics of a circuit as a whole. The advantage of the method is that it can give a reasonable estimate prior to actual physical layout process. The disadvantage of the method is that it only gives a statistical estimate. Information about the floorplanning is not considered. Also, the accuracy of the result depends heavily on the value of the parameters determined for the various distribution models. Therefore, its applicability to the problem of evaluating a floorplan is questionable.

2.2.3 Procedural Estimation

Procedural estimation methods attempt to give a more accurate estimate by integrating the estimate with the floorplanning process. This is more difficult than the other two methods because we have to predict the behavior of the future floorplanning and routing in the layout process.

Most of the wiring space estimate methods use a bottom-up strategy along the hierarchical tree of the circuit [8, 11, 23]. The method used by Thomas and Miller [23] in their floorplanning algorithm traverses the hierarchical tree in a bottom-up manner. At each node during the traversal, a global router is invoked to rout the interconnections between the children of the current node. Channel area is calculated based on the result of the global routing. Then the dimension of the node is expanded accordingly to make enough room for the channels. This method can give a relatively accurate estimate at the price of increased computation time.

Pedram and Preas proposed a simpler method to estimate wiring space [11]. For each net, a bounding box is calculated which encloses all the terminals attached to the net. Then the channels that intersect with the bounding box or are included by the bounding box are marked with a probability. This probability represents the likelihood that the wiring of the net will actually go through the channel. The density of each channel is calculated by summing up all the probabilities assigned to the channel by each net. Then the width of the channel is calculated according to the estimated density. This method is faster than the previous one, but it is less accurate.

It is more difficult to estimate wiring space in a top-down manner. In order to estimate wiring area, we have to predict the future behavior of the floorplanning process. Dai and Kuh [9] proposed a method that recursively refines the estimate as the algorithm moves from the top of the hierarchical tree downward. At each level of the hierarchical tree, a connectivity matrix C is constructed with each entry $c(i, j)$ representing the number of interconnections between block i and block j at that level. All the channels on the shortest topological path between block i and block j are marked with a probability representing the likelihood of the wiring actually going through the channel. Thus, for each channel, a probability matrix P is constructed with each entry $p(i,j)$ denoting the probability that the interconnections between block i and block j will actually go through that channel. The width of each channel is estimated by the formula

$$w = T_r \times \sum_{i=1}^{k} \sum_{j=1}^{k} c(i,j) \times p(i,j)$$

where T_r is a factor representing track sharedness, and k is the number of blocks at that level. The main problem with this method is that as we proceed top-down in the hierarchical tree, the shape function and the terminal positions of the blocks at lower levels are not yet known. Some assumptions about these characteristics have to be made and this could seriously affect the accuracy of the estimate.

2.3 Existing Floorplanning Algorithms

Floorplanning is not an easy task. It is impractical, if not impossible, for a floorplanning system to produce the optimal solution to a problem of interesting size in a reasonable amount of time. Therefore, most algorithms proposed for the floorplanning problem are either heuristic or some assumptions and/or restrictions are made so that the problem is much simplified. Heuristic algorithms can give a solution to the problem in a practical amount of time, but the solution is not necessarily optimal. In this section, we introduce several floorplanning algorithms. Some of them restrict the problem to a certain kind of floorplan and give the optimal solution with respect to a specific cost function. Some of them use a heuristic method to approximately optimize a floorplan.

2.3.1 Stockmeyer's Algorithm

The algorithm proposed by Larry Stockmeyer [20] computes an optimal orientation of the modules in a slicing floorplan. While it has been proved that optimizing the orientations of the modules in a general floorplan is NP-complete [20], this algorithm can give an optimal solution to a slicing floorplan in a polynomial amount of time with respect to the number of modules in the floorplan.

Before discussing the algorithm, let us first introduce the concept of *shape function* of a node in a slicing structure. The shape function of a node v in a slicing tree consists of a list of pairs (h_i, w_i) which gives the lower bound on possible dimensions of v's rectangle in the floorplan. That is, for any pair (h_i, w_i), there is an arrangement π_i of the modules enclosed by v's rectangle such that the height of the rectangle equals h_i and the width of the rectangle equals w_i. For any other arrangement π, suppose that the height and the width of v's rectangle under the arrangement π are h and w, respectively, then $w_i \leq w$ or $h_i \leq h$ or both.

Figure 2.3(a) gives an illustration of the shape functions of a node with a single module. In the figure, the shaded area contains all possible dimensions of the node's rectangle. There are two pairs describing the shape function of M_1: (1,3) and (3,1). The shape functions for the nodes in a slicing tree are calculated in a bottom-up

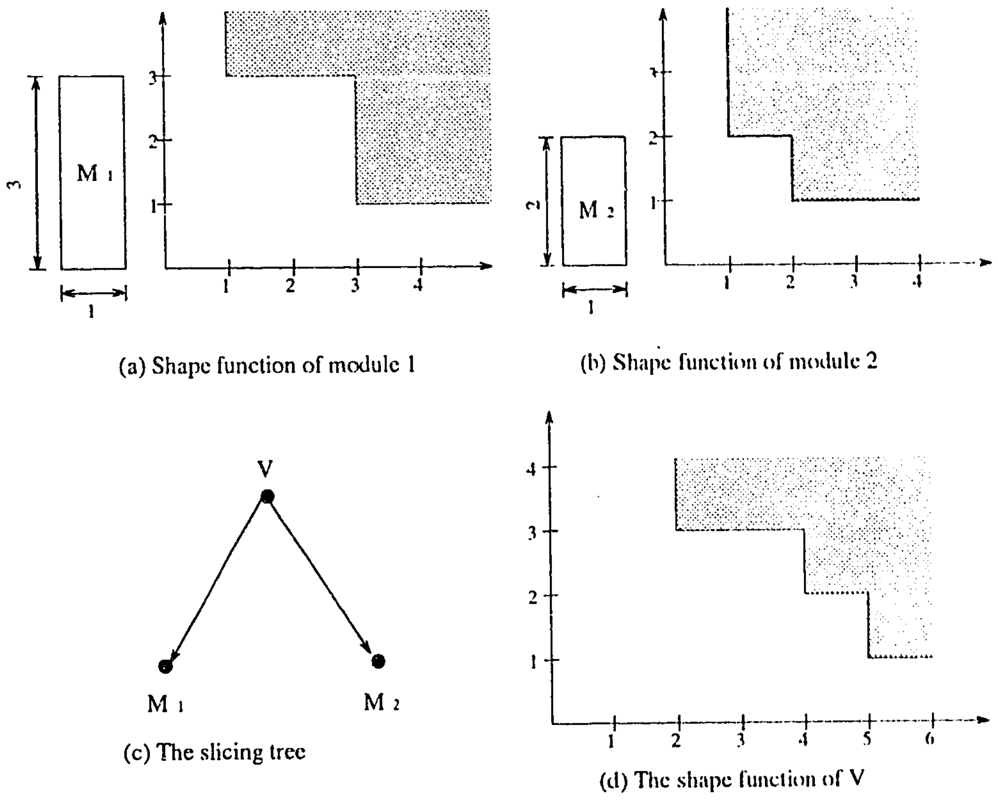

M 1
3
1
M 2
2
1
V
M 1
M 2
(a) Shape function of module 1
(b) Shape function of module 2
(c) The slicing tree
(d) The shape function of V

Figure 2.3: The shape function of a single module

manner. The calculation of shape function for a leaf node is straightforward, and therefore will not be discussed here. To calculate the shape function for an internal node v, consider the case where v has a vertical slice. Suppose the shape function of v's left child consists of m pairs (h'_i, w'_i) for $1 \leq i \leq m$, and the shape function of v's right child consists of k pairs (h''_j, w''_j) for $1 \leq j \leq k$. Then $m \times k$ pairs can be constructed for the shape function of v. Each pair is constructed by the equation $h_{i,j} = \max(h'_i, h''_j)$ and $w_{i,j} = w'_i + w''_j$. In the example of figure 2.3, the shape function of the node consisting of module 1 and module 2 has four pairs: $(3,3)$, $(3,2)$, $(2,4)$ and $(1,5)$. It should be noted that not all the new pairs are necessary for the corresponding shape function. In our example, it is obvious that the pair (3, 3) is not necessary because there is another pair (3, 2) which gives a better combination. The slicing tree and the shape function of the internal node are illustrated in figure 2.3(c) and 2.3(d). The shape function of the root of a slicing tree gives all the possible dimensions of the chip, each pair (h_i, w_i) corresponding to the chip dimension of an arrangement of modules. The one which gives the minimum chip area is chosen as the final floorplan.

As mentioned above, the algorithm constructs a list of pairs (h_i, w_i) for each node in the slicing tree in a bottom-up manner. Let v be a node in the slicing tree. The list of shape pairs constructed for v satisfies following conditions $h_i > h_{i+1}$ and $w_i < w_{i+1}$. The meaning of the conditions is that we do not keep the dimension $(h(\pi), w(\pi))$ of an arrangement π if there is another arrangement π' that is better than π in either h or w (or both) dimension and is not worse than π in the other dimension. The key feature of the algorithm is the way the shape pair lists are constructed for internal nodes. Consider an example of an internal node v with a vertical slice and its left and right child being v_l and v_r respectively. Assume there are k pairs in v_l's shape function and m pairs in v_r's shape function. As mentioned before, there are $k \times m$ possible pairs for v. In fact, we do not have to consider all these $k \times m$ possibilities because some of them are obviously suboptimal. For example, suppose we have a pair (h_i', w_i') from v_l's list and another pair (h_j'', w_j'') from v_r's list, then we can construct a new pair $(\max(h_i', h_j''), w_i' + w_j'')$ for v. If $h_i' > h_j''$, then there is no need to consider the combination of (h'_i, w'_i) with (h''_z, w''_z) for any $z > j$ because $\max(h'_i, h''_z) = h'_i$ and $w'_i + w''_z > w'_i + w''_j$.

Stockmeyer proved that for each node v in the slicing tree, the following relation is true: $m \leq |L(v)| + 1$, where m is the number of shape function pairs for node v and $|L(v)|$ is the number of modules enclosed by v's rectangle. Also he proved that the time complexity of the algorithm is $O(n \lg n)$, where n is the number of modules in the circuit.

2.3.2 The Wong and Liu Algorithm

The algorithm proposed by Wong and Liu [3] uses a heuristic searching strategy called *simulated annealing* to produce a slicing floorplan. Before discussing the algorithm itself, let us first introduce the concept of simulated annealing.

The idea of simulated annealing comes from Boltzmann's probability distribution of atomic configurations as a function of temperature. By Boltzmann's law, the probability of any configuration i of atoms is given by $e^{-E(i)/(k_b T)}$, where $E(i)$ is the energy associated with configuration i; k_b is Boltzmann constant; and T is the temperature. Therefore, at a certain temperature the most likely configurations of the atoms are those with the lowest energy. Suppose a solid-state material is heated to a high temperature until it reaches a liquid state so that the atoms can move freely. Then it is cooled down slowly. As the temperature cools down, the freedom of motion of atoms becomes smaller and smaller. If the cooling is slow enough so that the material reaches its thermal equilibrium state at each temperature, then the atoms will arrange themselves in such a way that the total energy is minimized. This process is called *annealing*.

Simulated annealing is an optimizing algorithm that simulates an annealing process. Given an initial floorplan and an initial temperature T_0, the simulated annealing algorithm gradually decreases the temperature according to a certain schedule. At each temperature T, a number of modifications to the floorplan are generated randomly. Such modifications to a floorplan are referred to as *moves*. If a move results in a decrease in the cost function, it is accepted unconditionally. If the move increases the cost function, it is accepted with a certain probability. The acceptance probability is a function of the temperature and the change the move made to the cost function. The more the cost is increased by a move, the less likely that the move will be accepted. The number of moves made at each temperature is determined by the annealing schedule. The algorithm stops when a certain terminating condition is met. For example, the terminating condition can be that the number of moves accepted exceeds a pre-determined value.

Now let's come back to the Wong and Liu algorithm. The algorithm is based on a new representation of slicing floorplans. The representation is called the ***normalized Polish expression***. Assume that the n modules are each labeled by a number $i : 1 \leq i \leq n$, and the vertical and horizontal slices are represented by $*$ and $+$, respectively. The idea of a *Polish expression* comes from the observation that a traversal of a slicing tree results in an "arithmetic expression" with * and + as operators, and the module labels as operands. A Polish postfix notation for this "arithmetic expression" can be constructed by a pre-order traversal of the slicing tree. A Polish expression is said to be *normalized* iff there are no consecutive *'s or +'s in the sequence. Wong and Liu proved that there is a one-to-one correspondence between the set of normalized Polish expressions of length $2n - 1$ and the set of slicing structures with n modules.

In a normalized Polish expression, a sequence of operators is called a *chain*. From the definition of normalized Polish expression, we know that there are only two kinds of chains: $+ * + * + * \ldots$ or $* + * + * \ldots$. The *complement* of a chain is defined to be the chain obtained by interchanging operators * and +.

During the optimizing process, the following three kinds of moves are used to modify a given normalized Polish expression:

- M1. Swap two adjacent operands.
- M2. Complement some chain of operators.
- M3. Swap an operand with an adjacent operator.

Wong and Liu proved that these three kinds of moves are sufficient to ensure that it is possible to go from any normalized Polish expression to any other via a sequence of such moves.

The cost function takes both the total chip area of the circuit and the total wire length into consideration. The total wire length TL is computed by the formula

$TL = \sum_{1 \leq i,j \leq n} c_{i,j} \times d_{i,j}$, where $c_{i,j}$ is the number of interconnections between module i and module j, and $d_{i,j}$ is the Manhattan distance between the centers of module i and module j. The algorithm starts from an arbitrary initial floorplan and an initial temperature T_0. At each temperature, enough moves are made until either there are N downhill moves or the total number of moves exceeds $2N$ where N is a predetermined constant. The algorithm terminates if the number of accepted moves is less than 5% of all moves made at a certain temperature or the temperature is low enough.

2.3.3 Performance-Driven Placement Algorithm

Jackson and Kuh [6] proposed a placement method that optimizes the performance of a circuit by considering such factors as pin capacitance, intrinsic cell delay, interconnect processing parameters and chip timing requirements.

The method mainly solves the long path problem. The long path problem is concerned with paths that have the greatest delay. Consider a path p, let l denote the end cell of the path, r_l be the time a signal is required to arrive at the input of l, and a_l is the actual arrival time of the signal at the input of l. The path slack at l is defined as $s_l = r_l - a_l$. In order for the circuit to operate correctly, $s_l \geq 0$ has to be true for all l. The path whose endpoint cell has the minimum slack is the most critical path in the circuit. Maximizing the slack of the critical path is equivalent to maximizing the clock frequency or the time performance of the circuit.

The proposed algorithm optimizes the placement in a hierarchical manner. The algorithm moves from the top level of the hierarchy downward. At each level of the hierarchy, a linear programming algorithm is used to determine the cell placement. It is unavoidable that some overlap will exist between modules after linear programming (LP), so a partitioning process is invoked after each LP to get rid of the overlaps. Then the LP algorithm is re-run at the next level of the hierarchy with the feasible regions of the cells constrained to reflect the most recent partitioning. This process is repeated until the chip area is divided into suitably small regions that contain a small number of cells.

The key idea of the method is the linear program. The timing requirements and other related information is included in the linear constraints. The first set of constraints is on the bounding boxes of the nets. The bounding box of a net i is defined by four parameters: t_i, b_i, r_i and l_i, which represent the top, bottom, right, and left boundaries of the bounding box, respectively. Suppose the net i has k_i terminals attached to it. For each terminal, let $(x_{i,j}, y_{i,j})$ denote its position in the floorplan. The constraints on the bounding box of net i is determined by the following equation:

$$b_i \leq y_{i,j} \leq l_i$$

$$l_i \leq x_{i,j} \leq r_i$$

The electrical behavior of the interconnection is modeled by a lumped capacitance that depends on the dimension of the bounding boxes of each net. The constraint of the lumped capacitance of each net i is determined by the equation

$$C_i = \gamma(l_i - r_i) + \beta(t_i - b_i)$$

where γ and β represent unit capacitance of horizontal and vertical metal lines respectively.

Suppose a single delay and an equivalent resistance exist for each cell, then the delay model of the example circuit in Figure 2.4 is defined by the following expressions:

$$a_3 \geq a_1 + d_1 + R_1C_1$$

$$a_3 \geq a_2 + d_2 + R_2C_2$$

where a_i denotes the arrival time at the input of cell i, d_i represents the intrinsic delay of cell i, R_i represents the equivalent resistance of cell i, and C_j represents the lumped capacitance of net j. The timing requirements are defined by the following constraints:

$$r_{1,3} \geq a_1 + d_1 + R_1C_1 + M$$

$$r_{2,3} \geq a_2 + d_2 + R_2C_2 + M$$

where $r_{i,j}$ is the required arrival time of a signal at the input of j through path i, and M is a variable introduced to every delay equation for path endpoints. The value of M represents the minimum slack in the design. The *objective function* of the linear programming formulation is

$$\max(M - \alpha W)$$

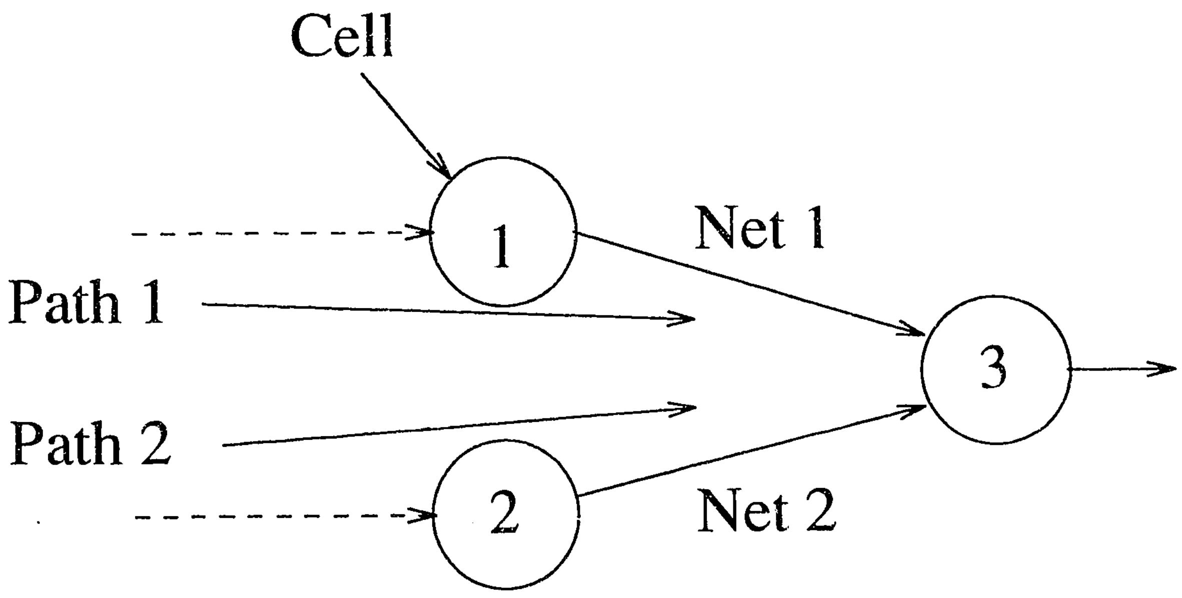


Figure 2.4: An illustration of path delay and timing requirement

In the formula, $\alpha = \frac{k\overline{R}}{n}$ where k is a user-defined weight constant, $\overline{R}$ is the average resistance of the nets, and n is the number of nets. Also in the formula, W is the total interconnection capacitance. The objective function reflects two important features of the circuit. M reflects the timing performance of the circuit, while αW reflects the weighted average interconnect delay. Maximizing M minimizes the delay of the paths, while maximizing $-\alpha W$ minimizes the sum of the half-perimeter of all nets, and therefore, minimizing the total wire length.

Following linear programming, a partitioning algorithm divides the chip area into two times the number of regions that existed at the previous hierarchical level. This partitioning assigns the cells to specific regions according to the LP result. As mentioned before, the purpose of partitioning is to get rid of the overlaps resulting from linear programming. Each time a partition is finished, a new set of constraints is introduced to specify feasible positions for each module.

2.4 Discussions and Conclusions

Floorplanning and routing are two closely related phases in the physical design of an integrated circuit. The quality of a floorplan has a profound impact on the quality of the routing. One of the major concerns in floorplanning is to arrange the modules in such a way that final chip area is minimized. In order to achieve this goal, we have to take routing into consideration during the floorplanning stage.

As mentioned before, wiring space estimating methods fall into three categories: *empirical estimates, stochastic estimates* and *procedural estimates.*

Empirical and stochastic estimates depend heavily on the previous experience of design and on the parameters of the statistical models. Such methods can only give a rough idea of wiring space. They do not take the current floorplan into consideration. Also they do not provide detailed information about wiring such as wiring space distribution. Therefore they are of little help to guide the floorplanning.

Generally speaking, procedural estimation methods are more accurate than empirical and stochastic methods in that they take the actual floorplan into consideration. The main disadvantage of procedural estimation is the time complexity of the method.

In the last few years, more and more work has been done attempting to reduce the time complexity of procedural estimation methods.

Several floorplanning algorithms were presented in this chapter. They try to floorplanning problem from different points of view. Stockmeyer's algorithm gives an optimal solution to the orientations of the modules in a slicing floorplan. The algorithm is of polynomial time complexity ($O(n^2)$), and therefore, is quite efficient. On the other hand, the algorithm does not consider the wiring aspects, therefore the solution it produces may not be optimal from the wiring point of view.

The most important feature of Wong and Liu's algorithm is the normalized Polish expression for floorplan representation. Some representations, such as a slicing trees, can have a number of representations for a single slicing structure. This can cause two problems for simulated annealing: (1) There is an unnecessary increase in the number of solution states. (2) The set of slicing structures is unevenly distributed over the set of representations, leading to undesirable biases toward some slicing structures. Because of the one-to-one correspondence between the set of slicing structures and the set of normalized Polish expressions, the above problems are eliminated. The disadvantage of the algorithm is that updating of the cost function is comparatively time consuming.

The method proposed by Jackson and Kuh directly optimizes the timing performance of the circuit. Since it minimizes the time slack of each path, it theoretically guarantees proper operation of the circuit. Of course, in practice such factors as operating temperature and simplification of the timing model will weaken the guarantee. Still the method can give the placement with the best performance because it provides the greatest margin for the variations on the performance. On the other hand, it is likely that the algorithm will result in a larger chip area compared with other algorithms because the area aspect of a floorplan is not explicitly addressed in the algorithm.

As mentioned before, wiring quality should be one of the major concerns during the floorplanning process. Stockmeyer's algorithm, although it gives an optimal orientation solution to a slicing floorplan, does not consider wiring at all. In Wong and Liu's algorithm, the wiring length was included in the cost function, but the estimation is not accurate. Also the wiring space was not directly addressed. Jackson and Kuh's algorithm optimizes the circuit performance, but its total chip area of the result may not be satisfactory. It is likely that there will be much empty space left unused in the final layout.

The main difficulty in dealing with wiring during the floorplanning process is the lack of an accurate yet efficient method to estimate the routing behavior. Most of the existing floorplaning algorithms [1, 3, 6, 15] address wiring by only using wire-length in the objective function, instead of wiring space. Although some algorithms can give fairly good floorplans, there is still quite a lot of empty space left unused after routing. We believe there is still room for improvement.

In this thesis, we propose a new method that re-arranges an existing slicing floorplan such that the extra space needed for wiring is minimized. The main idea of the method is to place the modules in such a way that the router can take full advantage of the empty space left over during floorplanning to route the interconnections. Also, a new method for estimating wiring space is proposed which provides some information about wiring space distribution.

Chapter 3

New Wiring-Space Optimization Algorithms

3.1 Introduction

Two algorithms for wiring space minimization are introduced in this chapter. The first algorithm uses the branch and bound method. This algorithm always gives the best solution in terms of the wiring space estimate, but its time complexity is such that it is only practical for small circuits. For large circuits, we introduce a heuristic algorithm derived from the branch and bound algorithm. The heuristic is a greedy algorithm in that the decisions are made based on local information instead of global information. That is, instead of considering all possible alternative solutions each time a decision is made, only those solutions that are most likely to lead to the best solution are considered. The algorithm does not necessarily give the best solution, but it can usually finish the search in a reasonable amount of time.

In the next section, we give a formal definition of the problem. The solution space of the problem is also discussed in this section. The new method for estimating wiring space and a lower bound on the estimate are described in section 3. In section 4, we discuss a pre-processing procedure which provides some necessary information for the branch and bound algorithm. In sections 5 and 6, the branch and bound algorithm and the greedy heuristic algorithm are introduced, respectively. Some search strategies aimed at speeding up the search process are discussed in section 7. The chapter ends with a summary in section 8.

3.2 The Problem

As mentioned before, the objective of the algorithms proposed in this thesis is to adjust the arrangement of an existing slicing floorplan so as to minimize the wiring space. The adjustments to module positions in a slicing floorplan are restricted to interchanging sibling rectangles and mirroring (flipping) modules in leaf rectangles.

The relative position of rectangles in an adjusted floorplan is indicated by a label A(v) attached to each node in the slicing tree. For internal nodes, A(v) can take on two values: F and R. The value F indicates that v's two children are in the *forward* or normal relative position. That is, the rectangle corresponding to v's left child v_l is placed to the left of the rectangle corresponding to v's right child v_r in the case of vertical slice, or below v_r's rectangle in the case of horizontal slice. Conversely, the label R indicates that v's two children are in the *reverse* relative position. That is, v_l's rectangle is placed to the right of v_r's rectangle in the case of vertical slice, or above v_r's rectangle in the case of horizontal slice. For leaf nodes, A(v) consists of two letters which describe the mirroring applied to module M(v). The first letter of A(v) can be either R (reverse) or F (forward or normal) indicating that the module is mirrored or not mirrored about the Y axis, respectively. Similarly, the second letter can be either U (upside-down) or R (right-side-up) indicating that the module is mirrored or not mirrored about the X axis, respectively. The four possible labels for A(v) at a leaf node are FR, FU, RR and RU.

Figure 3.1 illustrates the relative position adjustment of an example floorplan with its slicing trees. For the sake of simplicity, the slice direction is not marked for the nodes. Instead, the nodes are marked with their relative position $A(v)$. As can be seen from the figure, the rectangle containing module A was interchanged with the rectangle containing module C; the rectangle containing module D and F was interchanged with the rectangle containing module E; modules F and D were swapped; modules B and C were mirrored about the X axis; and modules B and F were mirrored about the Y axis.

The problem being addressed here will be referred to as the Relative Module Position, or RMP, problem. The problem is formally defined as follows:

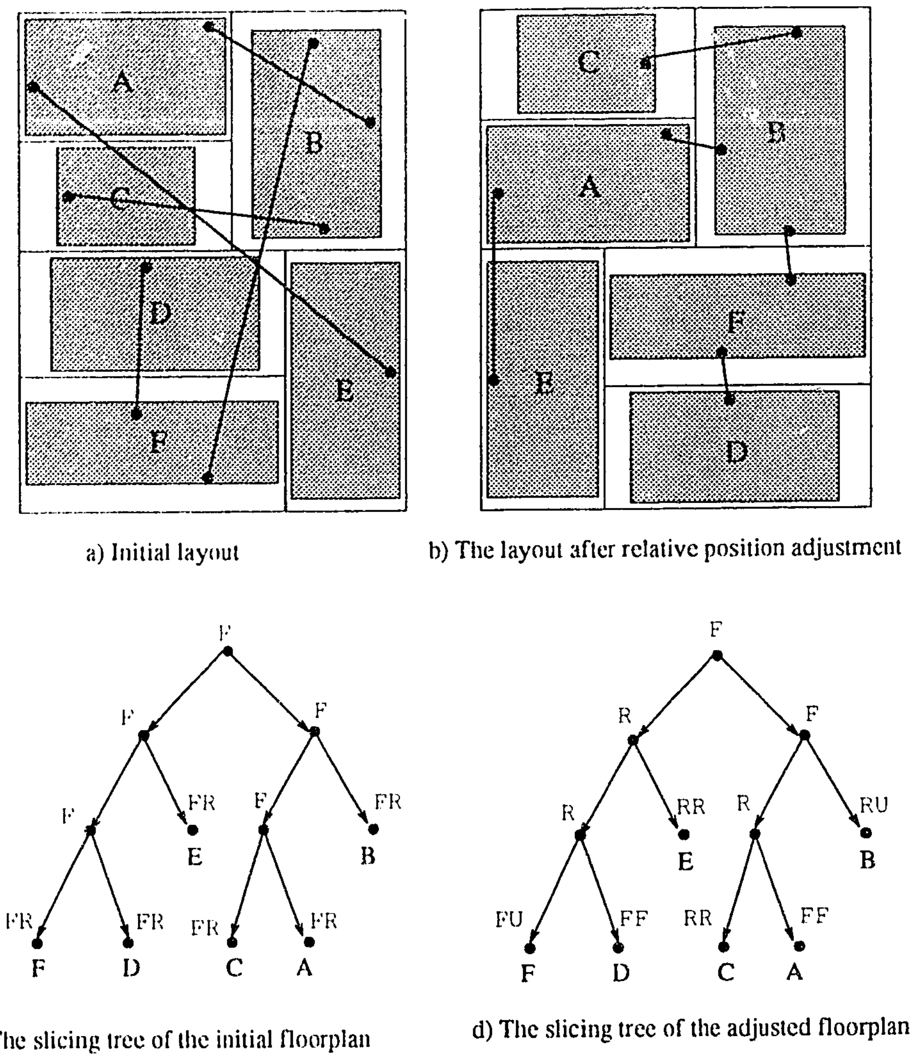
Given:

- *A slicing floorplan defined by a slicing tree.*
- *A circuit netlist which specifies the interconnections between the modules in the floorplan.*
- *Module dimensions and terminal locations for each module in the circuit.*

Determine:

$A(v)$ for all internal and leaf nodes in the slicing tree so that the wiring space estimate is minimized.

Each combination of relative positions assigned to the nodes in the slicing tree constitutes a possible solution to the problem. The set of all possible solutions is referred to as the *solution space* of the problem. The solution space can be represented by a *solution tree.* The root of the solution tree represents the initial state where no relative position has been assigned to any nodes in the slicing tree. Each internal node n in the solution tree other than the root represents a partial solution where relative positions have been assigned to some of the nodes in the slicing tree. Each of n's children represents a different possible relative position assignment at a node in the slicing tree. A path in the solution tree from the root to a leaf node gives a complete solution to the problem. Such paths are referred to as *solution paths.* The length of the solution paths, i.e. the depth of the solution tree, is equal to the number of



a) Initial layout

b) The layout after relative position adjustment

c) The slicing tree of the initial floorplan

d) The slicing tree of the adjusted floorplan

Figure 3.1: Relative position adjustment

nodes in the slicing tree. The objective of our algorithm is to, among all the possible solutions, pick up the one which yields the minimum wiring space estimate.

The RMP problem is very difficult. Consider a circuit with n modules. There are n leaf nodes in the corresponding slicing tree. Each leaf node has four possible assignments, so the number of combinations for leaf node assignment is $4^n = 2^{2n}$. The number of internal nodes in a binary tree is $n - 1$. Each internal node has two possible assignments, so the number of possible combinations of the assignments for the internal nodes is 2^{n-1}. Therefore, the total number of possible combinations of the assignments for the whole floorplan is 2^{3n-1}. That is, the number of solutions to the problem is 2^{3n-1}. Figure 3.2 illustrates an example slicing floorplan and its solution tree. The floorplan consists of only two modules placed one on top of the other. Yet there are 32 possible solutions.

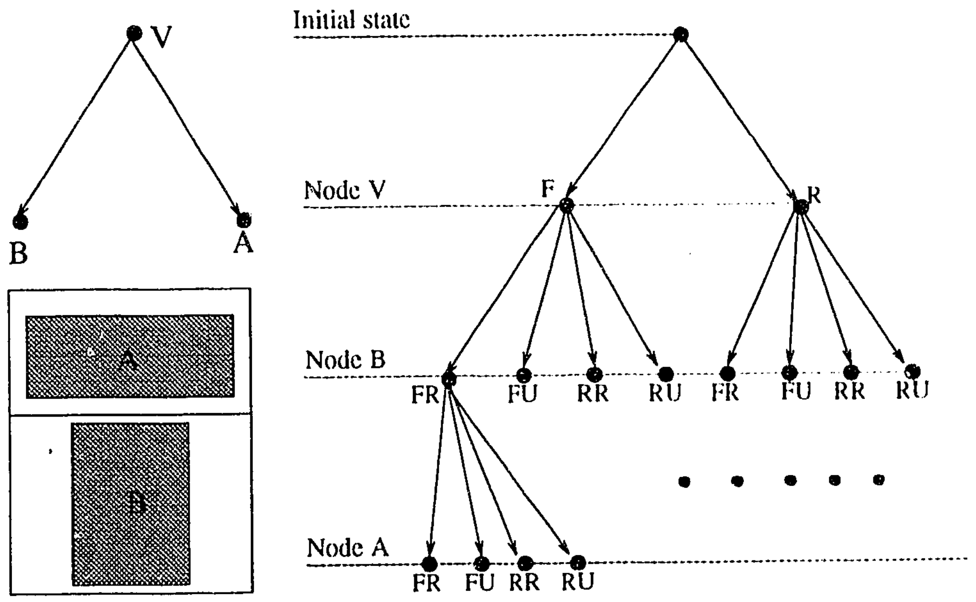


Figure 3.2: The problem solution space tree

A straightforward method to find the best solution, called *exhaustive search*, compares all the possible solutions and selects the one that yields the minimum wiring space estimate as the final solution. Although it is guaranteed to give the best solution, the method is too time consuming to be practical for problems of interesting size.

Quite a few techniques have been invented to speed up the search process. One of the most well known is called the *branch and bound* algorithm. Instead of searching all the way down through a solution path to get an exact value of the cost function for that solution, the branch and bound algorithm evaluates the prospect of the search along the current solution path and abandons any further search along the path if the path can not lead to a better solution than the best solution found so far. In this case, the algorithm cuts off the current path, backs up a node in the solution tree, and starts to search along a new path. This process is repeated until there are no more new solution paths to search. At this point the best solution found so far is indeed the best solution to the problem.

3.3 A Lower Bound on the Wiring Space Estimate

A good branch and bound algorithm can cut a lot of unnecessary search from a solution tree, and thus, dramatically reduce the search time. A key factor for a successful branch and bound algorithm is an efficient way to calculate a tight lower bound. If the lower bound is not tight enough, the algorithm still has to search along many paths much further before it realizes that such paths can not yield a better solution. On the other hand, if the lower bound is tight, but very complicated to calculate, the algorithm spends a great deal of time calculating lower bounds, therefore slowing down the search process. In this section we discuss how the lower bound on wiring space estimate is calculated for the RMP problem. Before discussing the lower bound, we must first explain how wiring space is estimated.

3.3.1 Estimating Wiring Space

As mentioned before, the main idea behind our wiring space minimization algorithms is to take full advantage of empty space left over in the floorplan for wiring the circuit. One way to take advantage of empty space is to make the wiring space distribution match the empty space distribution as closely as possible. In this way, much of the wiring can be placed in the empty area left over by the floorplanning process and thus, extra space required for wir[illegible]ed.

In order to take space distribution into account, we introduce the concept of a *space profile*. A space profile is a histogram which defines the amount of empty space or wiring space along the X or Y axis of a floorplan rectangle. Each interval in a space profile histogram is defined by a triple (s, e, h), where s is the starting point of the interval, e is the ending point of the interval, and h is the height or the amount of space in the interval.

In our estimate method, a space profile is used to describe one of three kinds of space: *empty space, wiring space*, and *wiring demand space*. We will discuss each kind of spac, and the corresponding space profiles in the following paragraphs.

Empty space is defined to be the space in a floorplan that is not occupied by any module of the circuit, and can be used for wiring the interconnections between the modules. An empty space profile of a node v, denoted by $EP(v)$, along the X/Y axis is a histogram which defines the empty space distribution along the X/Y axis.

Wiring space is estimated as follows. For each net, a minimum bounding box is computed that encloses all the terminals of the net. For each portion of the horizontal(vertical) interval of a net bounding box, there is at least one piece of wire that covers that interval. We assume that each net is routed by a horizontal wire and vertical wire that cover the horizontal and vertical extent of the net's bounding box. Then the space taken by these two pieces of wire is the minimum wiring space for the net. Thus, a *wiring space profile* of a node v, denoted by $WP(v)$, along the X/Y axis is a histogram of the minimum wiring space distribution along the X/Y axis.

The *empty space profile* describes the empty space distribution, and the *wiring space profile* describes the minimum wiring space distribution. By comparing the two profiles, we are able to tell if the existing empty space is enough to accommodate the wiring space. If the empty space is insufficient, we can compute the distribution of minimum extra space required for wiring along the X/Y axis. This gives rise to our third kind of space profile, the *wiring demand space profile.* A wiring demand profile, denoted by $WDP(v)$, describes the minimum amount of extra space needed for wiring. In other words, the height of each interval in a wiring demand profile along the X/Y axis gives the minimum amount of extra space needed to make enough room for wiring in that interval. The wiring demand profile is constructed by subtracting the empty space profile from the corresponding wiring space profile for a given rectangle. The details of the subtraction will be discussed later. Figure 3.3 illustrates various space profiles for an example floorplan. In the figure, the rectangles with dashed lines represent net bounding boxes.

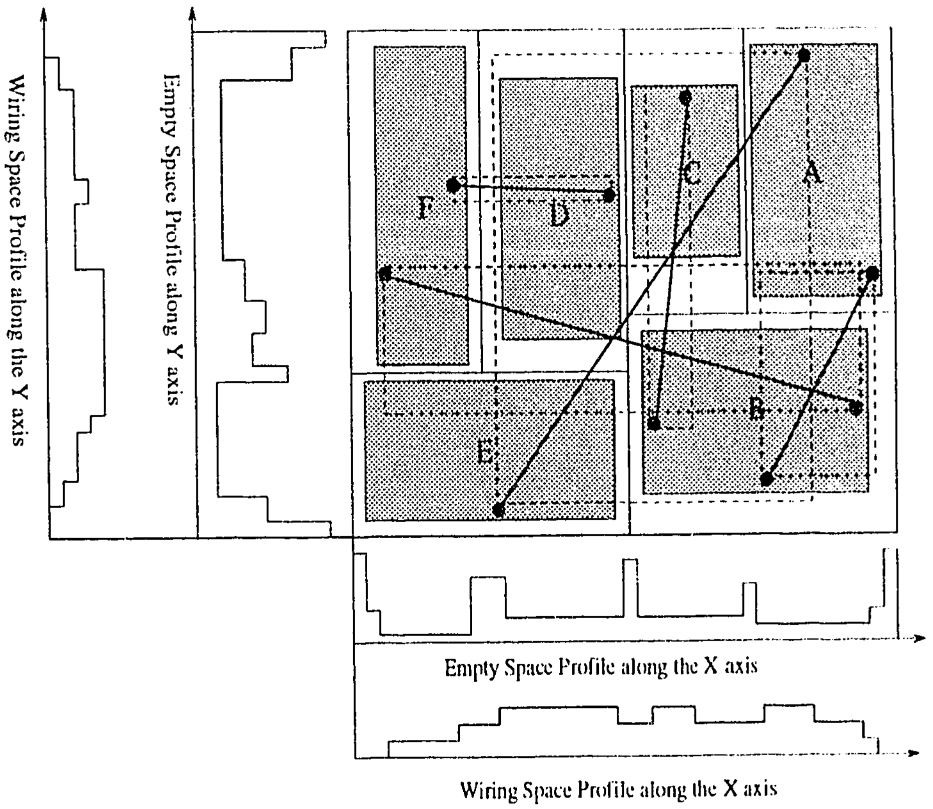

Wiring Space Profile along the Y axis
Empty Space Profile along Y axis
F
D
C
A
E
B
Empty Space Profile along the X axis
Wiring Space Profile along the X axis

Figure 3.3: Various space profiles of a circuit

Since the wiring demand profile specifies the extra amount of wiring space required, it is taken as our wiring space estimate. The objective of our wiring space minimizing algorithms is to minimize the maximum height of the wiring demand profiles along both the X and the Y axes. In doing this, the estimated extra space required for wiring the nets in the floorplan is minimized.

In order to manipulate space profiles, we need to specify some operations on them. The operations used in our algorithm include *FLIP*, *MAX*, *MIN*, *ADD*, *SUB* and *CONCAT*. We will discuss each operation in the following paragraphs.

A *FLIP* operation, denoted by FLIP(p), mirrors an existing profile p about the center of the profile. A MAX operation, denoted by MAX(p_1,p_2), takes two space profiles p_1 and p_2 as its operands and the result of the operation is a new profile. The resulting profile is such that at any point along the X/Y axis, the height of the point in the resulting profile takes the maximum value of the heights at the same point in the two operand profiles. A MIN operation, denoted by MIN(p_1,p_2), is similar to MAX operation except that the height at each point in the resulting profile takes the minimum value of the heights at the same point in the two operand profiles. An *ADD* operation, denoted by $p_1 + p_2$, adds two profiles together. It is similar to that of MAX operation except that, the height at any point in the resulting profile is the sum of the heights at the same point in the two operand profiles. A *SUB* operation, denoted by $p_1 - p_2$, subtracts profile p_2 from the profile p_1. It is similar to that of Add operation except that, at any point along the X/Y axis, the height in the resulting profile at the point is the difference of the heights in the two operand profiles at the same point. A *CONCAT* operation, denoted by CONCAT(p_1,p_2), produces a new profile by appending profile p_2 to the end of profile p_1. Figure 3.4 illustrates various operations on space profiles.

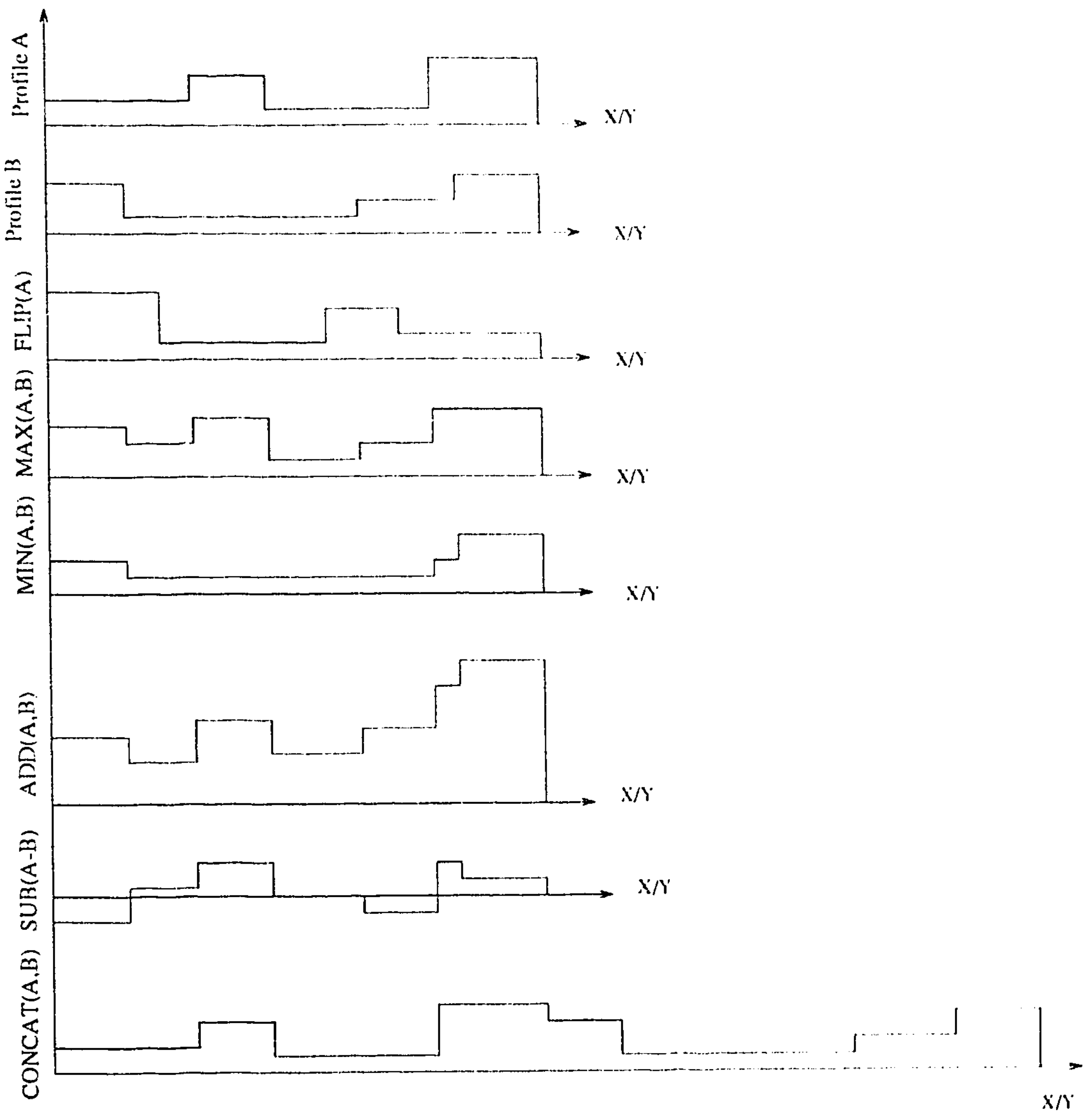


Figure 3.4: Space profile operations

Having introduced the concept of a space profile and the operations on space profiles, let's now discuss the algorithm for calculating the wiring demand profile for the circuit. The algorithm is called WDPC(Wiring Demand Profile Calculation). Figure 3.5 is the pseudo-code of the algorithm that calculates the wiring demand profile along the X axis. It should be noted that there is a new kind of profile in the algorithm, *PWP(N)*, the pseudo wiring space profile. A *pseudo wiring space profile* of a node N is the wiring space profile of those nets whose terminals completely lie inside N's rectangle but not completely inside any rectangle that is a child of N.

```
WDPC(N)
N: The current node to be processed;
EP(N): Empty space profile of node N;
WP(N): Wiring space profile of node N;
WDP(N): Wiring demand profile of node N;
PWP(N): Pseudo wiring space profile of node N;
{
        If (N is a leaf node) {
                Calculate EP(N);
                Calculate WP(N);
                WDP(N) = WP(N) - EP(N);
        }
        Else { /* An internal node */
                Get N's left child $N_l$;
                WDPC($N_l$);
                Get N's right child $N_r$;
                WDPC($N_r$);
                Calculate PWP(N);
                If ( N has vertical slice ) {
                   If ($N_l$ and $N_r$ are forward positioned) {
                      EP(N) = CONCAT(EP($N_l$),EP($N_r$));
                      WP(N) = CONCAT(WP($N_l$),WP($N_r$)) + PWP(N);
                   } ELSE {
                      EP(N) = CONCAT(EP($N_r$),EP($N_l$));
                      WP(N) = CONCAT(WP($N_r$),WP($N_l$)) + PWP(N);
                   }
                   WDP(N) = WP(N) - EP(N);
                }
                Else { /* the slice is horizontal */
                   EP(N) = EP($N_l$) + EP($N_r$);
                   WP(N) = PWP(N) + WP($N_l$) + WP($N_r$);
                   WDP(N) = WP(N) - EP(N);
                }
        }
        Return;
}
```

Figure 3.5: Pseudo-code for calculating wiring demand profile along the X axis

The algorithm traverses the slicing tree in a bottom-up manner to calculate the wiring demand profile for each node in the tree by recursively calling itself. Consider a recursive instance of WDPC(n) processing node n in the slicing tree. If n is a leaf node. then we know the dimension of n's rectangle as well as the dimension of the module m assigned to n. We can easily calculate the empty space profile of n from this information. For the wiring space profile at a leaf node, we only consider those nets whose terminals all lie inside n's rectangle. Then the wiring space profile is calculated from the bounding boxes of these nets. The wiring demand profile is obtained by simply subtracting the EP(n) from WP(n).

If n is an internal node, the calculation is a little more complicated. First, let us discuss the calculation of the wiring space profile for n. The nets in n can be classified into three categories: the nets with some, but not all, of their terminals inside n's rectangle; the nets with all their terminals inside n's rectangle, but not all inside any of n's children's rectangles; and the nets whose terminals all lie inside one of n's children's rectangles. For those nets in the first category, we can not determine the bounding boxes of these nets because we do not know the positions of those terminals outside n's rectangle at this point. Therefore, we will not consider such nets until we encounter a node whose rectangle completely encloses their terminals. For the nets in the third category, the wiring space is already included in either n_l's or n_r's wiring space profile $WP(n_l)$ or $WP(n_r)$. Therefore, there is no need to re-calculate the wiring space profiles of these nets at n. The nets in the second category are the only nets that need to be considered. We can easily calculate the wiring space profile of these nets by projecting the corresponding bounding boxes. As mentioned above, we call the wiring profile of these nets the pseudo wiring profile $PWP(n)$ of node n. From this we can calculate the wiring space profile $WP(n)$ for node n. If the slice at n is horizontal, we just add $WP(n_l)$, $WP(n_r)$, and $PWP(n)$ together to get $WP(n)$ along the X axis. If the slice is vertical, assume n's two children are forward positioned, we first concatenate $WP(n_r)$ to the end of $WP(n_l)$, then add the resulting profile to $PWP(n)$ to get $WP(n)$ along the X axis. If the children are in the reversed position, the calculation is similar except that the order of the concatenation is reversed.

The empty space profile along the X axis at an internal node n, $EP(n)$ is easy to calculate. If the slice is horizontal, we add $EP(n_l)$ and $EP(n_r)$ together to get $EP(n)$ along the X axis. If the slice is vertical, we concatenate $EP(n_r)$ to the end of $EP(n_l)$ to get $EP(n)$ along the X axis, provided that n's two children are forward positioned. Otherwise, the order of the concatenation is reversed. The wiring demand space profile $WDP(n)$ is calculated by subtracting the empty space profile $EP(n)$ from the wiring space profile $WP(n)$.

The various space profiles along the Y axis are computed in a similar manner.

3.3.2 The Minimum Bounding Box of a Net

As mentioned before, the wiring space profile is computed based on the minimum bounding box of the nets. In this sub-section, we discuss the computation of these bounding boxes.

The minimum bounding box of a net n can be described by four parameters: *l(n),*

$r(n)$, $t(n)$, and $b(n)$, which give the "best case" positions of the box's left, right, top and bottom boundaries respectively. In order to precisely define these parameters, it is necessary to first make the following supporting definitions.

- The expression $t \in n$ means that terminal t is one of the terminals connected by net n.
- $\pi(v)$ is a set of relative position assignments, $A(v_i)$, for all the nodes v_i in the subtree rooted at node v.
- S is the set of nodes in the slicing tree whose relative positions have already be assigned.
- C(S) is the set of all possible arrangements $\pi(R)$ which contain the arrangement defined for the nodes in S as a subset. Here R represents the root of the whole slicing tree.

The minimum bounding box parameters l(n) and r(n) are defined as follows:

$$l(n) = \max_{\pi \epsilon C(S)} \min_{t \epsilon n} x(t, \pi)$$

$$r(n) = \min_{\pi \epsilon C(S)} \max_{t \epsilon n} x(t, \pi)$$

where $x(t, \pi)$ is the X position of terminal t in arrangement π. Described informally, $l(n)$ is the rightmost position possible for the leftmost of n's terminals, given the assignments in S. Similarly, r(n) is the leftmost position of the rightmost of n's terminals, given the assignments in S. The definitions of t(n) and b(n) are similar to that of r(n) and l(n), respectively, except that they are based on the vertical position of the terminals.

To calculate the initial bounding boxes of each net, as well as to update them during the branch and bound process, two other parameters need to be computed. They are *xmin()* and *ymin()*. Before defining these parameters, we need the following definitions:

- $d_x(v, t, \pi)$ is the X distance from the left side of v's rectangle to terminal t in the floorplan defined by the arrangement π.
- $d_y(v, t, \pi)$ is the y distance from the top side of v's rectangle to terminal t in the floorplan defined by the arrangement π.
- $P(v)$ is the set of all possible arrangements for the subtree rooted at v.

xmin() and ymin() are defined as follows:

$$xmin(v, n) = \min_{\pi \epsilon P(v)} \max_{t \epsilon n \wedge t \epsilon v} d_x(v, t, \pi)$$

$$ymin(v, n) = \min_{\pi \epsilon P(v)} \max_{t \epsilon n \wedge t \epsilon v} d_y(v, t, \pi)$$

In other words, $xmin(v, n)$ is the shortest possible distance from the left side of v's rectangle to the rightmost terminal of n that is inside v's rectangle, over all possible arrangements of the modules inside v's rectangle. For each arrangement $\pi(v)$, there is another arrangement $\pi'(v)$ that is $\pi(v)$ mirrored about the Y axis. Therefore, it does not matter whether *xmin* is measured from the left side or the right side of v's rectangle, the values will be always the same.

The value of ymin() is determined similarly, except that the distance is measured vertically from the bottom or top of v's rectangle.

As can be seen from the above discussion, the values of xmin() and ymin() for each node in the slicing tree are independent of the relative position of the node. Therefore, they can be computed before the branch and bound process.

The computation of the net minimum bounding boxes using the parameters of xmin() and ymin() will be explained later in section 3.5.

3.3.3 Lower Bound on Wiring Demand Profile

In order to use the branch and bound algorithm to minimize the maximum height in the wiring demand profiles of the circuit along both X and Y axis, we need to compute lower bounds on these profiles.

As explained before, the wiring demand profile is constructed by subtracting the empty space profile from the wiring space profile. In order for the wiring demand profile to be a lower bound, the empty space profile must represent the maximum empty space available for wiring, while the wiring space profile must be the minimum space required for wiring. Thus, the problem of computing a lower bound on wiring demand profile is transformed into the problems of computing an upper bound on empty space profile, and of computing a lower bound on the wiring space profile. We will discuss each problem in the following paragraphs. For the sake of simplicity, we will only discuss the case of profiles along the X axis. The case of the profiles along

Y axis is dealt with in a similar way.

The computation of the upper bounds on the empty space profiles in a slicing tree is done in a bottom-up manner. Since we know the exact dimensions of a leaf node's rectangle and the dimensions of the module assigned to that node, it is straightforward to compute the empty space profile of a leaf node.

At each internal node v, it is assumed that the upper bound of the empty space profiles of its children is known. If the node has a horizontal slice, we just add the two empty space profiles of the children together to get an upper bound of empty space profile of the node. If the node has a vertical slice, the process is a little more complicated. Since we don't know whether the two children will ultimately be forward positioned or reverse positioned, we have to consider both possibilities. First we concatenate $EP(v_r)$ to the end of $EP(v_l)$ to get the empty space profile for the forward positioned profile $EP_f(v)$. Next we flip $EP_f(v)$ to get the reverse positioned profile $EP_r(v)$. Finally these two profiles are MAXed together to give $EP(v)$. If $EP(v_l)$ and $EP(v_r)$ are the upper bound empty space profiles of v_l and v_r respectively, then $EP(v)$ is the upper bound empty space profile of v. Since the empty space profiles for the leaf nodes are computed according to the actual dimensions of the modules and the relative position of the leaf node does not affect the empty space profile, these empty space profile are indeed the upper bound empty space profiles for these leaf nodes. Thus, it is guaranteed that $EP(v)$ is the upper bound empty space profile of v. Figure 3.6 illustrates how the upper bound empty space profiles

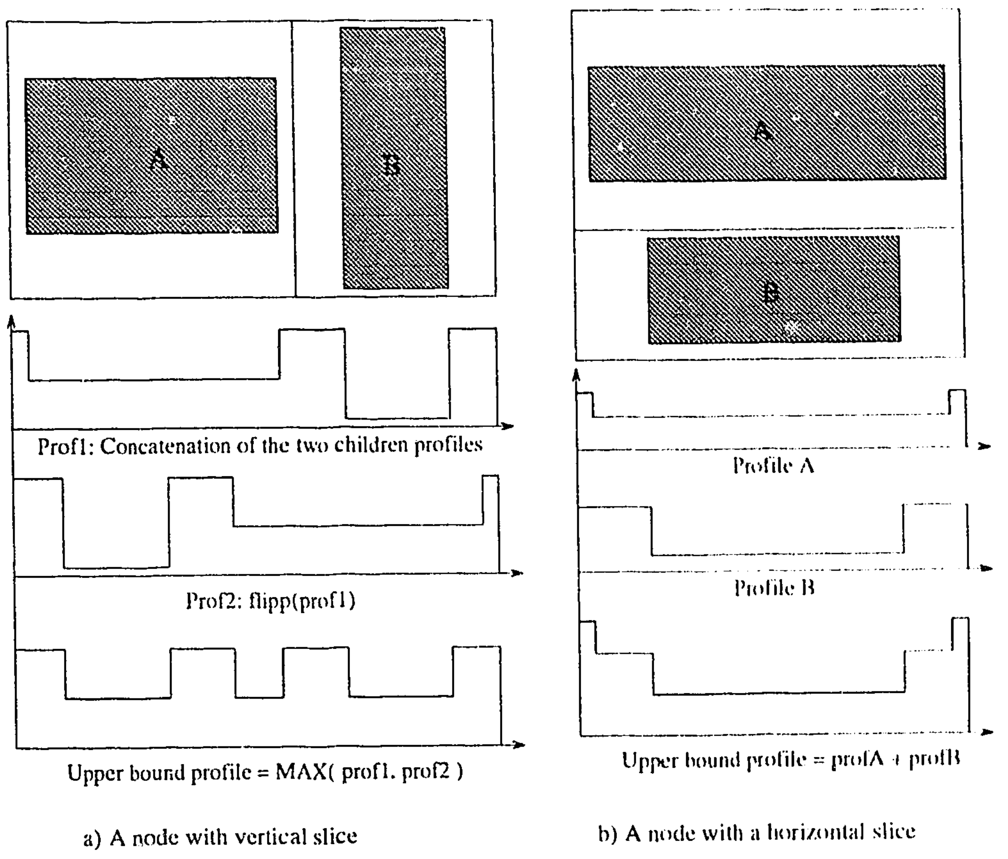

A
B
Prof1: Concatenation of the two children profiles
Prof2: flipp(prof1)
Upper bound profile = MAX(prof1. prof2)
a) A node with vertical slice
A
B
Profile A
Profile B
Upper bound profile = profA + profB
b) A node with a horizontal slice

Figure 3.6: Calculation of the upper bound for the empty space profile along the X axis

are calculated. To update the the upper bound space profile during the branch and bound process, either $EP_f(v)$ or $EP_r(v)$ is used as the upper bound once it has been decided which relative position to assign to v.

The computation of a lower bound wiring space profile is similar to that for the upperbound empty space profile. If the node v being processed is a leaf node, then the minimum bounding boxes of those nets whose terminals all lie inside v's rectangle are projected onto the X axis to give the lower bound wiring space profile. For an internal node v it is assumed that the lower bound wiring space profile of v's children, v_l and v_r, are known and are represented by $WP(v_l)$ and $WP(v_r)$ respectively. Suppose the node v has a vertical slice. First, we concatenate $WP(v_r)$ to the end of $WP(v_l)$ to get a the wiring space profile $WP'_f(v)$ for the case where v_l and v_r are in the forward position. Note that this wiring space profile only contains the wiring of those nets whose terminals all lie inside either v_l's or v_r's rectangle. Then we compute $PWP(v)$, the pseudo wiring space profile of v, by projecting the minimum bounding boxes of those nets whose terminals all lie inside v's rectangle but not all inside either v_l's or v_r's rectangle. The final forward positioned wiring space profile, $WP_f(v)$, is obtained by adding $PWP(v)$ and $WP'_f(v)$ together. The reverse positioned wiring profile, $WP_r(v)$ is obtained by flipping $WP_f(v)$. Since $WP(v_l)$ and $WP(v_r)$ are the lower bound wiring space profiles of v_l and v_r respectively, and $PWP(v)$ is constructed by projecting the minimum bounding boxes of the related nets, it is clear that $WP_f(v)$ and $WP_r(v)$ are the lower bound profiles of the forward positioned and reverse positioned wiring spaces profile respectively The final lower bound wiring space profile, $WP(v)$ is constructed by [illegible]ing $WP_l(v)$ and $WP_r(v)$ together. From the explanation of MIN operation, we can see that $WP(v)$ is indeed a lower bound on wiring space profile of the node v. To update the lower bound wiring space profile during the branch and bound algorithm, either $WP_f(v)$ or $WP_r(v)$ is used as the lower bound, depending on the relative position assigned to v.

3.4 The Pre-processing Algorithm

Before applying the branch and bound algorithm to a slicing tree, the slicing tree needs to be pre-processed to prepare some information needed by the branch and bound algorithm. The following information about each node is computed in the pre-processing algorithm:

- L(v): A list of nets whose terminals lie both inside and outside of v's rectangle.
- L1(v): A list of nets whose terminals all lie inside v's rectangle but not completely inside any of v's children's rectangles.
- Initial l(), r(), t() and b() for each net.
- Various initial space profiles for each node.
- $xmin()$ and $ymin()$ for each net at each node.

The pre-processing algorithm traverses the slicing tree bottom-up to compute the above information for each node. The computation of various initial space profiles is the same as the algorithm $WDPC(v)$ explained in section 3.3.1. The construction of the lists L(v) and L1(v) is straight forward and therefore, will not be discussed here. In this section, we will discuss how the initial values of l(), r(), t(), and b(), as well as $xmin()$ and $ymin()$ are computed.

Let us first explain the computation of $xmin()$. The computation of ymin() is handled in a similar manner and will not be explained in detail.

At any given node v, the pre-processing algorithm needs only to calculate $xmin(v, n)$ for nets $n \epsilon L(v)$. Suppose the node v being processed is an internal node with a vertical slice. Since the slicing tree is processed in a bottom-up manner, it can be assumed that $xmin(v_l, n)$ and $xmin(v_r, n)$ are already known, where v_l and v_r are v's left and right child respectively. At this point we do not know whether v_l and v_r will ultimately be forward positioned or reverse positioned, so we have to consider both possibilities. The xmin() value for the case where v_l and v_r are forward positioned is determined by:

$$xmin_f(v, n) = W(v_l) + xmin(v_r, n) = a$$

where $W(v_l)$ is the width of v_l's rectangle. Similarly, the xmin() value for the case where v_l and v_r are reverse positioned is determined by

$$xmin_r(v, n) = W(v_r) + xmin(v_l, n) = b$$

where $W(v_r)$ is the width of v_r's rectangle. The final value of $xmin(v, n)$ is determined by taking the minimum value of $xmin_f(v, n)$ and $xmin_r(v, n)$. The relationship among the above equations is illustrated in Figure 3.7.

It is possible that net n has terminals lying inside either v_l's rectangle or v_r's rectangle, but not both. Suppose n has no terminals inside v_l's rectangle. In this case, the value of $xmin(v_l, n)$ does not exist. Under such conditions, the value of $xmin(v, n)$ will take the value of $xmin(v_r, n)$.

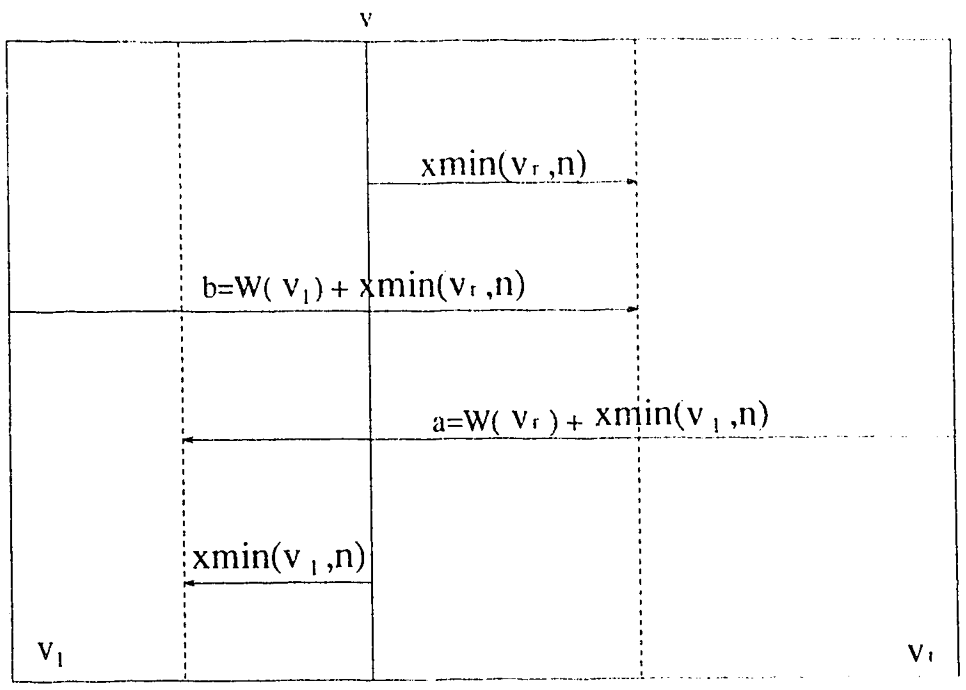


Figure 3.7: Calculation of $xmin()$

The value of $ymin(v,n)$ is determined by

$$ymin(v,n) = \max(ymin(v_l,n), ymin(v_r,n))$$

If v is a leaf node, $xmin(v,n)$ and $ymin(v,n)$ are determined directly from the layout of the module $m(v)$. Two cases need to be considered to determine the value of $xmin(v,n)$; that is, whether the module $M(v)$ is mirrored or not mirrored about the Y axis. $Xmin(v,n)$ is computed for both cases, and the minimum of the two values is taken as the final value of $xmin(v,n)$. The value of $ymin(v,n)$ is calculated similarly except that the cases considered are whether the module is mirrored about the X axis or not, and the distances are measured vertically.

Now let us discuss the computation of the initial l(n), r(n), t(n), and b(n). During pre-processing, we do not know the ultimate relative position of the nodes, therefore, we have to consider both forward and reverse relative positions for each internal node. Suppose the node v has a vertical slice, then the bounding box parameters of net n are computed by the following equations:

$$r(n) = \min(W(v_l) + xmin(v_r, n), \quad W(v_r) + xmin(v_l, n))$$

$$l(n) = W(v) - r(n)$$

$$t(n) = \max(\; ymin(v_r, n), \quad ymin(v_l, n) hspace 0.2cm)$$

$$b(n) = H(v) - t(n)$$

```
Pre-Process( v )
v: a node in the slicing tree.
{
        If( v is an internal node ) {
                $v_l$ = v's left child;
                $v_r$ = v's right child;
                Pre-Process( $v_l$ );
                Pre-Process( $v_r$ );
                Construct $L(v)$ and $L1(v)$;
                Calculate the various initial space profiles;
                For each net $n_j c L(v)$
                        Calculate $xmin(v, n_j)$ and $ymin(v, n_j)$
        }
        Else /* a leaf node */
                Construct $L(v)$ and $L1(v)$;
                Calculate the various initial space profiles;
                For each net $n_j c L(v)$
                        Calculate $xmin(v, n_j)$ and $ymin(v, n_j)$
        }
        return;
 }
```

Figure 3.8: Pseudo-code of the Pre-Processing algorithm

It should be noted that, at the stage of pre-precessing, we don't know the ultimate position of each node's rectangle. Therefore, the above values are calculated with respect to the lower-left corner of e's rectangle. Once the position of e's rectangle is known during the branch and bound process, these values are transformed into the global coordinate system.

Figure 3.8 is the pseudo-code for the preprocessing algorithm. The algorithm recursively calls itself to process each node in the sling tree in post order.

3.5 The Branch and Bound Algorithm

After the whole slicing tree has been pre-processed, the branch and bound algorithm can be applied to the problem. In this section we discuss the algorithm in detail. Figure 3.9 is the pseudo-code of the branch and bound algorithm, BandB().

Taking a queue of the nodes, λ, in the slicing tree as its input, BandB() operates by recursively processing the queue λ. The only restriction to the order in which the nodes appear in λ is that a node must appear before any of its children. The reason for this is that whenever BandB() picks up the next node from the head of the queue λ for processing, the parent of this node will have already been processed. Consequently the exact position of the parent is known. To simplify the discussion, we assume that the algorithm only optimizes the wiring demand profile along the X axis. The way to optimize the profile along the Y axis is the same.

As mentioned before, the objective of BandB() is to minimize the maximum height of the wiring demand profile of the whole circuit; that is, the maximum height of WDP(R), where R is the root of the slicing circuit. Let P represent the maximum height in WDP(R). BandB() globally maintains two measures associated with the wiring demand profile. The first one is P_{best}, that is, the best value of P found so far by BandB(). The other one is P_{low}, the lower bound on P given the current relative positions assigned to those nodes that have been already processed. The initial value of P_{low} is the maximum height of $WDP_{init}(R)$, the initial wiring demand profile of the root of the slicing tree, computed by the pre-processing algorithm. The initial value of P_{best} can be set to a large value. Also maintained by BandB() is a set of parameters l(), r(), t() and b() for each net in the circuit. The initial values of these parameters are provided by the pre-processing algorithm.

```
BandB($\lambda$)
$\lambda$: a queue containing all the nodes in the slicing tree in
    parent-first order.
$P_{low}$: Lower bound on wiring demand space profile.
$P_{best}$: The best value of wiring demand profile obtained so far.
{
    If $\lambda$ is not empty {
        $v_p$ = next node in $\lambda$;
        Remove $v_p$ from $\lambda$;
        If( $v_p$ is an internal node ) {
            For $A(v_p) \in \{F, R\}$ {
                Update($v_p$);
                If( $P_{low} < P_{best}$ ) then BandB($\lambda$)
                Undo all the changes made to $v_p$;
            }
        }
        Else { /* a leaf node */
            For each possible relative position {
                Update($v_p$);
                If( $P_{low} < P_{best}$ ) then BandB($\lambda$)
                Undo the changes made to $v_p$;
            }
        }
        }
        Else { /* the queue is empty */
            If( $P_{low} < P_{best}$ ) {
                $P_{best} = P_{low}$;
                Save the current set of relative position assignment
                  as the best solution found so far.
            }
        }
    }
```

Figure 3.9: Pseudo-code of the Branch and Bound algorithm

BandB() starts from an initial state where the queue λ is full, and no relative position has been assigned to any of the nodes in the slicing tree. BandB() proceeds from the initial state, assign.ng relative positions to the nodes in λ, one node at a time. A recursive instance of BandB() begins by removing the next node v_p from the queue λ. For the present, assume that v_p is an internal node. First BandB() assigns the relative position F to v_p and updates the related information affected by the assignment, particularly, P_{low}. If $P_{low} < P_{best}$ then it is possible to get a better solution than the best one found so far if BandB() searches along the current solution path. In this case, BandB() calls itself to process the remaining nodes in λ. Alternatively, if $P_{low} > P_{min}$, then the current solution path cannot lead to a better solution, and further searching along the solution path is abandoned. Regardless of the result of comparison between P_{low} and P_{best}, BandB() will always backtrack up to the current node, assign the relative position R to v_p, re-compute P_{low}, and repeat the comparison of P_{low} and P_{best}. Each time a backtrack is made, the changes to the related information due to the abandoned relative position assignments are undone so that these assignments will have no impact on the future search.

The leaf nodes are processed in a manner similar to that of an internal node. The only difference is that BandB() tries all four possible relative positions which are possible for a leaf node: FR, FU, RR and RU.

Eventually, the queue λ will be empty. At this point, every node in the slicing tree has been assigned a relative position and P_{low} is no longer just a lower bound. Instead, it is the actual maximum height of the global wiring demand space profile in the floorplan defined by the assignments. If $P_{low} < P_{best}$, then the current set of assignments of relative positions has the best wiring space profile found so far. Therefore, BandB() sets P_{best} equal to P_{low} and saves the current set of assignments as the best assignment found so far.

One of the most important aspects of the branch and bound algorithm is updating the relative information each time a relative position is assigned to a node v_p. The information which requires updating includes: the bounding box parameters of nets whose terminal positions are affected by the assignment; the empty space profile of v_p; the wiring space and wiring demand profiles of v_p; and the global wiring demand profile of the whole circuit. In the following paragraphs we will discuss how this information is updated.

The way to update the bounding box parameters during BandB() is slightly different from the way the initial values were calculated in the pre-processing algorithm. Now, the exact position of v_p's rectangle is known, so we can use the global coordinate system to specify the positions of the net bounding boxes. Consider a v_p with its left and right children being v_l and v_r respectively. Suppose v_p has a vertical slice and is assigned a forward relative position. The minimum bounding boxes of the nets in $L(v_l)$ are updated as follows:

$$r'(n) = X(v_p) + xmin(v_l, n)$$

$$r_{new}(n) = \max(r_{old}(n), r'(n))$$

$$l'(n) = X(v_p) + W(v_l) - xmin(v_l, n)$$

$$l_{new} = \min(l_{old}(n), l'(n))$$

where $X(v)$ is the X position of the lower-left corner of v's rectangle, and $W(v)$ is the width of v's rectangle. Similarly, the bounding boxes of the nets in $L(v_r)$ are updated as follows:

$$r'(n) = X(v_p) + W(v_l) + xmin(v_r, n)$$

$$r_{new}(n) = \max(r_{old}(n), r'(n))$$

$$l'(n) = X(v_p) + W(v_p) - xmin(v_r, n)$$

$$l_{new} = \min(l_{old}(n), l'(n))$$

Since the relative position assignment at a node with a vertical slice does not change the Y positions of the terminals, there is no need to update t(n) and b(n). Figure 3.10 gives an illustration of how the bounding box of a net in $L(v_r)$ is updated. If the node is assigned a reverse relative position, then the updating is similar except

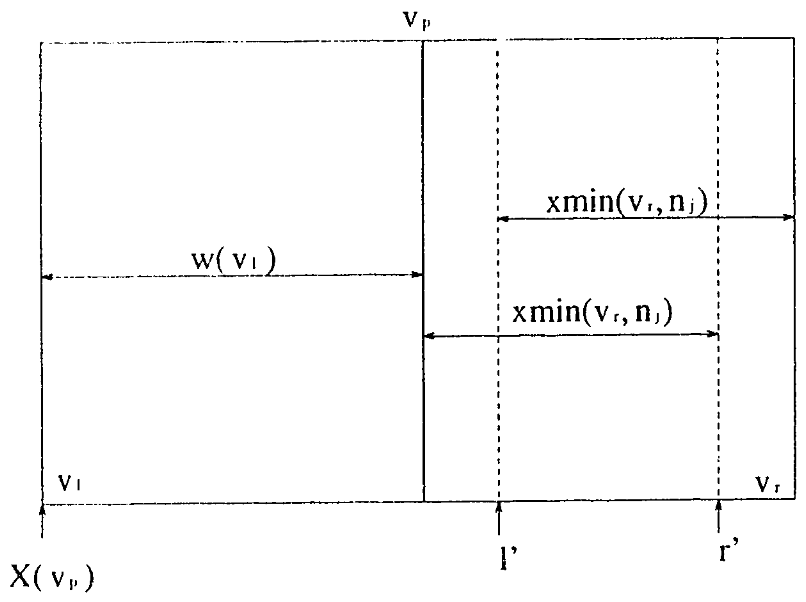


Figure 3.10: Updating of the minimum bounding box

that the roles of v_l and v_r are interchanged. In the case of a node with a horizontal slice, b(n) and t(n) will be updated in a similar manner.

If node v_p is a leaf node on the slicing tree, once $A(v_p)$ has been assigned a value, the exact positions are known for all of the terminals on modules $m(v_p)$. As a consequence, it is trivial to update the minimum bounding box for a net n_j which has terminals in v_p. The exact position of each terminal t is simply compared with the positions of the sides of n_j's minimum bounding box, and the minimum bound box is enlarged as required.

If a net's minimum bounding box is expanded as a result of the above update, the expanded sections of the bounding box are recorded and are later used to update various space profiles and P_{low}. One exception to this rule occurs when $l(n_j) > r(n_j)$.

In this case, the minimum bounding box is assumed to have zero width and no changes are made to P_{low}.

Now let us discuss how the wiring demand space profiles are updated. Consider the case where the node v_p is an internal node with a vertical slice, and we are updating its wiring demand profile along the X axis. As mentioned before, once a relative position has been assigned to v_p, the dimensions of some of the minimum bounding boxes may be expanded. All the sections of the expanded minimum bounding boxes are recorded and a wiring space profile is constructed from these expanded sections. We call such a profile $EWSP(v_p)$. Note that the nets involved in calculating $EWSP(v_p)$ are those nets in $L(v_l)$ and $L(v_r)$, where v_l and v_r are v_p's left and right children respectively. Assume that v_p was assigned a forward relative position, in this case the new wiring space profile is updated as follows:

$$WP_{new}(v_p) = concat(WP(v_l), WP(v_r)) + PWP(v_p) + EWSP(v_p) \tag{3.1}$$

The new empty space profile of v is updated by the equation

$$EP_{new}(v) = CONCAT(EP(v_l), EP(v_r)), \tag{3.2}$$

and the new wiring demand profile of v is updated by

$$WDP_{new}(v) = WP_{new}(v) - EP_{new}(v). \tag{3.3}$$

Substituting equation 3.1 and 3.2 into 3.3 we get

$$\begin{aligned} WDP_{new}(v_p) &= CONCAT(WP(v_l), WP(v_r)) - CONCAT(EP(v_l), EP(v_r)) \\ &\quad + PWP(v) + EWSP(v) \end{aligned}$$

Since

$$concat(WP(v_l), WP(v_r)) - concat(EP(v_l), EP(v_r))$$

$$= concat((WP(v_l) - EP(v_l)), (WP(v_r) - EP(v_r)))$$

$$= concat(WDP(v_l), WDP(v_r)),$$

equation 3.4 is equivalent to

$$WDP_{new}(v_p) = concat(WDP(v_l), WDP(v_r)) + PWP(v) + EWSP(v)$$

That is, the wiring demand profile of v_p's right child v_r is concatenated to the end of that of v_p's left child v_l. Then the expanded wiring space and the pseudo wiring space profiles of v_p are added to the result of the concatenation to get the new wiring demand profile. Since the ultimate purpose of updating is to update the wiring demand profile of the root, we don't have to update the empty space and the wiring space profile separately. Thus, the computing time is reduced. The case where v_p was assigned a reverse relative position is handled in a similar manner except that the roles of v_l and v_r are interchanged. Since the assignment of a relative position to v_p has no effect on parameters $t()$ and $b()$ when the slice direction is vertical, there is no need to update the various profiles along the Y axis.

Since P_{low} is computed based on the wiring demand profile of the whole circuit, $WDP(R)$, this profile also needs to be updated each time a relative position is assigned to a node. As explained in the previous section, each node v in the slicing tree contributes to $WDP(R)$ with its own $WDP(v)$. The root wiring demand profile is then updated by substituting v's contribution to $WDP(R)$, which was the old $WDP(v)$, with its updated wiring demand profile $WDP_{new}(v)$. The updating is expressed by the following equation:

$$WDP(R) = WDP(R) - WDP_{old}(v) + WDP_{new}(v)$$

Of course, some extra sections with zero height have to be appended to both the start and the end of $WDP(v)$ so that v's profile will be added to the right position in $WDP(R)$ where v's rectangle is placed.

Figure 3.11 is the pseudo-code of the UPDATE algorithm. For the sake of simplicity, only the updates to the wiring demand profiles along the X axis are explained.

```
UPDATE(v)
ExpdSec: a list of expanded sections of bounding boxes,
        initially empty.
{
    If(v is an internal node with a vertical slice) {
        For each net $n_j$ in $L(v_l)$ and $L(v_r)$ {
            Calculate $l_{new}(n_j)$ and $r_{new}(n_j)$;
            If the minimum bounding box was expanded
                Put the expanded section(s) into ExpdSec;
        }
        Calculate $EWSP(v)$ from ExpdSec;
        Calculate $WDP_{new}(v)$;
    }
    Else if(v is a leaf node) {
        For each net $n_j$ in $L(v)$ {
            Calculate $l_{new}(n_j)$ and $r_{new}(n_j)$;
            If the minimum bounding box was expanded
                Put the expanded section(s) into ExpdSec;
        }
        Calculate $EWSP(v)$;
        Calculate $WDP_{new}(v)$
    }
```

```
        $WDP(R) = WDP(R) - WDP_{old}(v) + WDP_{new}(v)$
        $Return;$
    }
```

Figure 3.11: Pseudo-code of the UPDATE algorithm

3.6 The Heuristic Algorithm

Although the branch and bound algorithm is much faster on average than exhaustive search, it still requires exponential time to solve the RMP problem. Therefore, it is only practical for relatively small circuits. For large circuits, we need a faster algorithm that does not necessarily give the best solution, but rather gives a reasonably good solution in a practical amount of time. In this section, we describe a heuristic algorithm derived from the branch and bound algorithm.

The heuristic algorithm is basically a greedy algorithm with look-ahead. The main idea is that, as we process each node in the slicing tree, the relative position that will most likely lead to the best solution is assigned to the node as part of the final solution. In order to take more global information into consideration, our algorithm looks several nodes ahead while making the decision for the current node. Specifically, each time a relative position is assigned to a node v, the next k-1 nodes in the queue λ are also taken into consideration. The branch and bound algorithm is applied to the k nodes to produce a best solution for these nodes. Then the relative position of v in the best solution is recorded as part of the final solution, and the k-1

```
HEURIS(v, k)
λ: A queue of the nodes in the slicing tree.
λ1: A queue of k nodes for the look-ahead search.
{
    Remove the next k nodes from λ and put them into λ1.
    While ( λ is not empty ) {
        BandB(λ1)
        Remove the first node v from λ1.
        Record A(v) as part of final solution.
        Remove the next node v_1 (if any) from λ;
        Put v_1 to the end of λ1;
    }
    Save the relative positions of the last k nodes in λ1
     assigned by BandB() as part of the final solution;
}
```

Figure 3.12: Pseudo-code of UPDATE algorithm

nodes following v are put back into the queue λ. The algorithm repeats this process to assign relative position to the next node in the λ, and so on.

Figure 3.12 is the pseudo-code of the look-ahead heuristic algorithm.

We can control the number of nodes to look ahead by adjusting the value of k. In fact, if k is set to the number of nodes in the whole slicing tree, then HEURIS() acts exactly as BandB().

HEURIS() does not necessarily give the optimal solution of the problem, but it can give a reasonably good solution in a practical amount of time. Suppose we have a slicing tree with n nodes. As mentioned before, the complexity of BandB() for the problem is $O(2^{3n-1})$. While in HEURIS(), the problem is divided into $n - k + 1$ sub-problems each of the complexity $O(2^{3k-1})$. Therefore, the complexity of HEURIS()

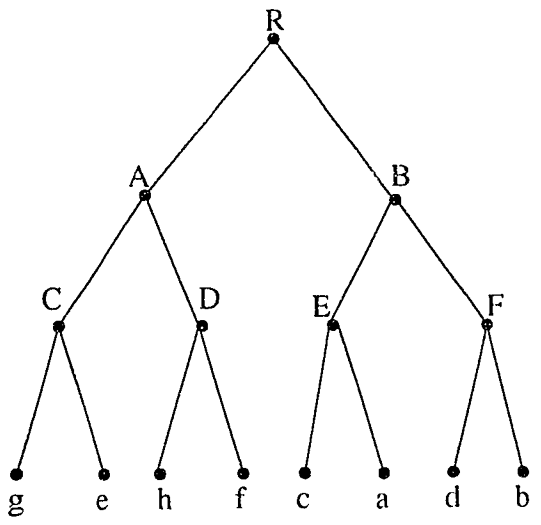
[illegible] the problem is $O((n-k)2^{3k-1})$. Since k is a constant, HEURIS() runs in linear time with respect to n.

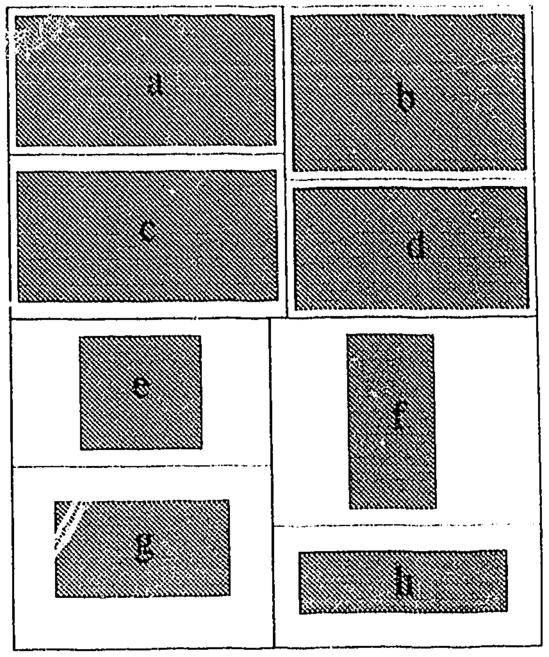
[illegible].7 Searching Strategies

The main problem with BandB() is the searching speed. In this section we will discuss some searching strategies aimed at speeding up the search process.

As mentioned before, the relative positioning of nodes with a vertical slice in the slicing tree, as well as mirroring the modules about the Y axis, has no effect on the various space profiles along the Y axis. This means the optimization of wiring demand profiles along the X and the Y axis can been done separately. This can dramatically reduce the search time. Suppose a slicing tree with n nodes is such that half of the internal nodes have vertical slices and half have horizontal slices. If the problem is solved as a whole, (that is, horizontal and vertical optimizations are done together) the complexity of the problem would be $O(2^{3n-1})$. On the other hand, if the horizontal and vertical optimization are done separately, the problem is then divided into two sub-problems each of the size of $n/2$, that is, the complexity of the problem would be $O(2 \times 2^{3n/2-1})$.

Another way to speed up the search process is to adjust the order in wh[illegible]ch nodes appear in the queue λ. As mentioned before, the only restriction on the or[illegible] is that a parent node must appear before any of its children. On the other hand, the order of the nodes in λ can affect the searching speed of BandB().



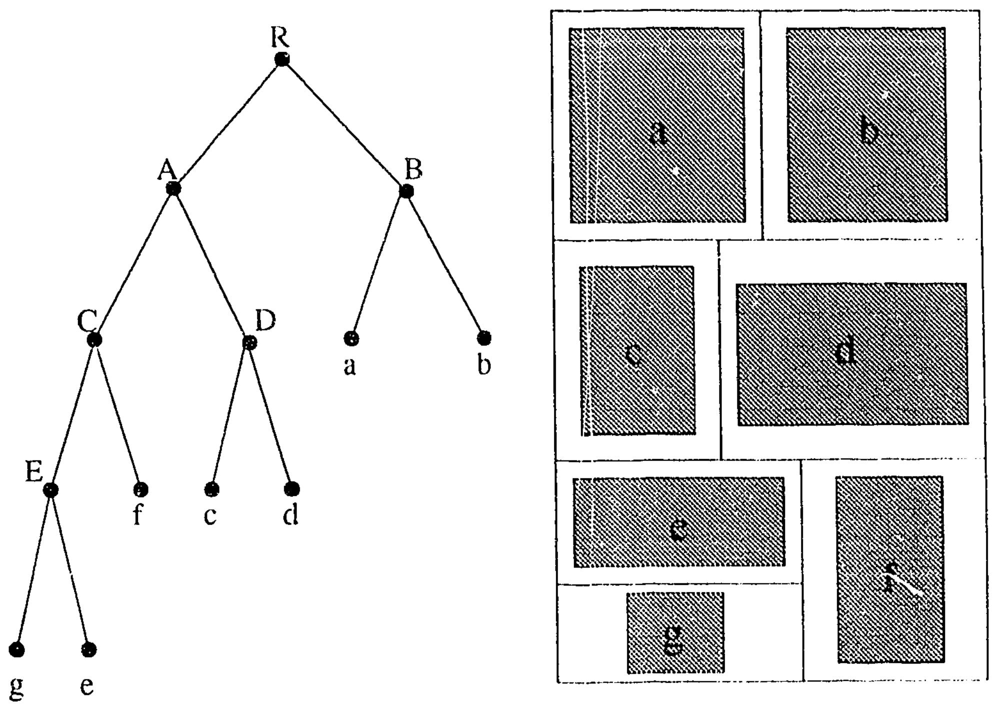


Breadth-first queue: (R, A, B, C, D, E, F, g, e, h, f, c, a, d, b)
Depth-first queue: (R, A, C, g, e, D, h, f, B, E, c, a, F, d, b,)
Bottle-neck-first queue: (R, B, E, c, a, F, d, b, A, C, e, g, D, f, h)

Figure 3.13: Breadth-first and depth-first queues of an example slicing tree

Typically there are two ways to arrange the nodes in the queue: breadth-first and depth-first. A breadth-first queue is constructed by putting the nodes into the queue level by level in the slicing tree. That is, the root is the first node in the queue followed by two of its children in left to right order, and so on. A depth-first queue can be constructed by putting the nodes into the queue in preorder. That is, put the current node into the queue, then recursively process the subtree rooted at the currents node's left and right child respectively. Figure 3.13 gives an example slicing tree together with it's breadth-first queue and depth-first queue.

Some new ways are proposed for constructing the depth-first queue so as to speed up the search. As can be seen from Figure 3.13, the conventional way to construct the queue is to always put the the nodes in the subtree rooted at the current node's left child before those in the subtree root at the current node's right child. In the example of Figure 3.13, there is less empty space in node B's rectangle than in A's rectangle. If the order in the queue is such that the nodes in the subtree rooted at A come before the nodes in the subtree rooted at B, then BandB() will always assign relative positions to A's subtree first before trying to arrange B's subtree. Since B's rectangle contains less empty space, it is more likely that the situation $P_{low} > P_{best}$ will occur when BandB() processes B's subtree. That is, BandB() may have to assign relative positions to all the nodes in A's subtree and some of the nodes in B's subtree before it realizes that the current arrangement can not lead to a better solution. If we put those nodes in B's subtree before those in A's subtree, the situation $P_{low} > P_{best}$ will occur while BandB() is arranging the nodes in B's subtree. Thus BandB() will realize that the current arrangement can not lead to a better solution before it tries to arrange those nodes in A's subtree and some unnecessary searching would be saved. We call this search strategy *bottleneck-first search.* The name comes from the fact that those nodes that are likely to contain a wiring bottleneck are processed first. The queue for bottleneck-first search is constructed in a similar way as the conventional depth-first queue, except that those nodes with a tighter wiring demand profile are processed before their siblings because bottlenecks are more likely exist in those nodes with a



Fewer-children-first queue: (R, B, a, b, A, D, c, d, C, f, E, g, e)

Figure 3.14: Fewer-child first queues of an example slicing tree

tighter wiring demand profile. The bottle-neck-first queue for the example slicing tree in Figure 3.13 is

$$(R, B, E, c, a, F, d, b, A, C, e, g, D, f, h)$$

In some floorplans, the slicing tree can be quite unbalanced. The lower bound is very loose as BandB() is processing the internal nodes at higher levels of the slicing tree. As it searchs towards the leaf nodes, the lower bound gets tighter and tighter. If we put those nodes with fewer children before their siblings into queue λ, BandB() would process some leaf nodes in the slicing tree early during the searching process through a solution path. Thus the lower bound becomes tighter more quickly as BandB() searches through a solution path, and hopefully BandB() can prune away many unnecessary searches earlier. We call such a search strategy the *Fewer-children-first* search. Figure 3.14 gives an example floorplan together with its slicing tree and the *fewer-children-first queue*.

Chapter 4

Implementation and Experimental Results

The majority of the implementation of the project consists of three parts: the branch and bound algorithm BandB(), the heuristic algorithm Heuris(), and an interface program Yal2edge(). The total size of the programs is about 5000 lines, including comments. The programs were developed on Sun Sparc Station 2 machines running the UNIX operating system. Most of the programs were written in C++.

There are several reasons for choosing C++ to implement the programs. First, C++ provides some machine level features. As mentioned before, execution time is one of the major concerns of our algorithms. If the machine level features provided by C++ are used to advantage, the efficiency of the programs can be greatly improved. Secondly, C++ supports object-oriented-programming. The *class* construct in C++

provides an encapsulation mechanism to implement abstract data types. The implementation details of a data type can be made inaccessible to client code that uses the type. Therefore the programs are easier to change and maintain. Finally, C++ is one of the most popular languages used in the field of software development. Most of the existing machines and systems have C++ compilers, thus programs written in C++ are highly portable.

The algorithms were evaluated by applying them to several benchmark circuits provided by the Microelectronics Center of North Carolina (MCNC). The resulting floorplans produced by the algorithms, as well as the original floorplans, were routed by the routing programs of Cadence EDGE, a VLSI CAD software, to evaluate the effectiveness of the algorithms.

The wiring space optimization programs take two files as their input. The first one is a .yal file written in YAL format which describes the geometry of the circuit modules and the interconnections between these modules. Although a YAL file can describe the floorplan of a circuit, it can not describe the slicing structure of a floorplan. The second input file, the .tre file, gives a description of the slicing tree of the floorplan.

The output of the wiring space optimization programs is a YAL file. In addition to the description of module geometry and interconnections of the circuit, the floorplan of the circuit, i.e, the position of each module, is also described in the YAL output file. The interface program *yal2edge*, along with some other interface programs provided by Cadence, translates the resulting floorplan from YAL format to EDGE format which is used by the Cadence system to route the floorplans.

4.1 Implementation

In this section, we explain the structure of three programs BandB(), Heuris(), and Yal2edge(). The programs BandB() and Heuris() were written in C++, while Yal2edge() was written in C.

The program BandB() implements the branch and bound algorithm. The program consists of five parts: *Input*, *Init*, *Prep*, *Bandb*, and *Yalout*. First, *Input* is invoked to read in the .yal file. After the .yal file is read in, *Init* is invoked to read in the .tre file and to construct the slicing tree of the floorplan. Next *Init* traverses the slicing tree to compute the dimension of the rectangles for each node in the slicing tree. After that, *Prep* is invoked to pre-process the slicing tree. It performs all the functions of the preprocessing algorithm as described in the previous chapter. Following the preprocessing, *Bandb* executes the branch and bound search. The final solution is recorded as the labels $A(v)$ for each node v in the slicing tree. Finally, *Yalout* computes the position of each module in the resulting floorplan and saves the result in a YAL file.

The program *Heuris()* is quite similar to the program *BandB()* except that the subroutine *bandb* is replaced with *heuris* which implements the heuristic algorithm explained in the previous chapter.

In order to evaluate the result of the wiring space optimization algorithms, we need to route several benchmark circuits to compare the routed floorplan before and after optimization. Cadence routing programs were used to route the floorplans. Before being processed by Cadence routing programs, the circuit and floorplan data need to be transformed from YAL format into a format accepted by Cadence. Yal2edge performs this transformation.

Cadence uses its own data format, called the *EDGE* format, to represent the circuits and floorplans. We can not directly transform YAL data into *EDGE* format because *EDGE* format is not open to users. Several interface programs are provided by Cadence to facilitate the transformation between the *EDGE* format and several other standard data formats. One of these programs is *EDIFIN* which transforms *EDIF* (Electronic Data Interchange Format) data into *EDGE* data. The problem is that the *EDIFIN* only translates the network information of the circuit described in an *EDIF* file, and the programs for translating module geometry from *EDIF* into *EDGE* format have not yet been implemented. Fortunately, Cadence provides a language called *SKILL* to access and manipulate the EDGE data base.

Our interface program, Yal2edge, takes the YAL file as its input, and generates two output files. The first one is a *SKILL* file which describes the module geometries in the circuit. The other one is an EDIF file which describes the interconnections between the modules. After running Yal2edge, the *SKILL* file is executed by Cadence to generate the module representations, and the *EDIF* file is read by EDIFIN to generate the net representations. After this, the circuit is ready for routing.

Benchmark	Number of modules	Number of nets	Number of terminals
Apte	9	40	214
Xerox	10	203	796
Hp	11	56	264
Ami33	33	89	480
Ami49	49	408	931

Table 4.1: The statistics of the benchmarks

4.2 Experimental Results

Five MCNC benchmarks were used to test the algorithms. The statistics of these benchmarks are shown in table 4.1. In this section, we will first examine the time performance of the optimizing algorithms, then we will present the experimental results of applying the optimization algorithms to the above benchmark circuits and the area results of routing the floorplans both before and after optimization.

The initial floorplans of the benchmark circuits were obtained from the paper published by Onodera et al. [22]. The mirroring of the leaf nodes can not be determined from the paper, therefore, they were assigned arbitrarily.

4.2.1 Time performance of the algorithms

Table 4.2 gives the cpu time used by BandB() to optimize the floorplans of the three smaller benchmarks. Note that those benchmarks with the suffix "_2" are similar to

Benchmark	Search	CPU time (s)	Benchmark	Search	CPU time (s)
Apte	D	11.8	Apte_2	D	38.9
	B	10.3		B	21.9
	FC	15.3		FC	19.8
	BN	9.1		BN	10.4
Hp	D	38.9	Hp_2	D	159.1
	B	504.6		B	560.8
	FC	197.1		FC	204.0
	BN	24.5		BN	30.3
Xerox	D	598.6	Xerox2	D	595.6
	B	1150.8		B	580.1
	LC	66.8		LC	53.6
	BN	47.5		BN	46.1

Table 4.2: Time performance of the branch and bound algorithm

the corresponding benchmarks without the suffix except that one third of the sibling pairs of nodes were interchanged in the "_2" version. In the table, D represents depth-first search; B represents breadth-first-search; FC represents fewer-children-first search; and BN represents bottleneck-first search. The user time was measured in seconds.

From table 4.2, it can be seen that bottle-neck-first search outperforms the other three searching strategies in all cases. In some cases such as *hp* and *xerox*, the improve-

Benchmark	k=1	k=4	k=8	k=16
Apte	1.9	5.5	15.5	
Hp	2.7	9.8	39.9	
Xerox	4.8	14.5	43.4	
Ami33	12.0	52.4	250.4	9116.0
ami49	48.0	211.7	971.7	43524.0

Table 4.3: Time performance of the heuristic algorithm

ments are quite substantial. In most cases, the fewer-child-first search outperforms depth-first and breadth-first search. There is no clear evidence as to which is better between depth-first search and breadth-first search. BandB() was also applied to the two larger benchmarks, ami33 and ami49, but the program failed to finish after several hours.

Table 4.3 gives the time performance of the heuristic algorithm on the benchmark circuits with various values of k, the number of nodes to look ahead. As can be seen from the table, the search time increases dramatically with the value of k. Another interesting thing about the table is that when k=8, Heuris() spent more time on the small circuits such as *Apte* than BandB() did. The reason for this is that as k approaches n, the number of nodes in the slicing tree, the heuristic algorithm performs more and more like a branch and bound algorithm. Since the heuristic algorithm has to spend some time on the overhead, it can be even slower than the branch and bound algorithm. As the size of the circuit increases, the heuristic algorithm will be much faster than the branch and bound algorithm.

4.2.2 Routing Results of the Floorplans

Both the initial floorplans and the floorplans after wiring space optimization were routed by the Cadence routing program. Before discussing the results of the routing, let us first examine the wiring space estimates of the benchmark circuits before and after optimization.

Table 4.4 gives the wiring space estimates before and after wiring space optimization. In the table, $P_{init}(X/Y)$ and $P_{opt}(X/Y)$ represent the maximum height of the wiring demand space profile along the X/Y axis before and after optimization, respectively. The negative value means that there is more empty space in the floorplan than the minimum wiring space required. Let's take Apte_2 as an example. The table shows that $P_{init}(X)$ is 148, that means the initial floorplan must be expanded vertically at least by the amount of 148 microns to make enough space for the wiring. On the other hand, $P_{opt}(X) = -92$ means that there is no need to expand the optimized floorplan vertically because there is enough empty space in the floorplan to accommodate the wiring. In fact, the narrowest margin between the empty space and the required wiring space over the entire horizontal extent of the floorplan is 92. Of course, this conclusion is based on the assumption that the wiring space estimate is 100% accurate. In reality, the floorplan will always need to be expanded to accommodate wiring. First, our wiring space estimate only gives the minimum space required

Benchmark	$P_{init}(X)$	$P_{opt}(X)$	$P_{init}(Y)$	$P_{opt}(Y)$
Apte	-44	-92	150	150
Apte_2	148	-92	198	150
Hp	194	122	162	96
HP_2	194	122	162	96
Xerox	228	184	394	358
Xerox_2	300	184	394	358
Ami33	99	-54	83	-5
Ami49	440	112	284	-520

Table 4.4: The optimization results on the wiring space estimate

for wiring and the actual wiring space is in fact larger than estimated, and secondly, some of the empty space can not be used for wiring because of the mismatch between the empty space and the wiring space distributions. Therefore it is impossible for the floorplan to get smaller after routing.

As can be seen from table 4.4, the wiring den. and profiles have been improved by the optimizing algorithms in all cases. The heuristic algorithm was also applied to the three smaller benchmarks *Apte, Hp, and Xerox*, with several values of k. In most cases, the optimal solution was achieved by the heuristic algorithm.

Table 4.5 gives the estimated areas for the benchmark circuits. The total area of a circuit is computed as follow: The width of the circuit is computed by adding the width of the unrouted floorplan and the maximum height of the corresponding wiring

Benchmark	Total Area			Wiring+Empty area		
	before	After	Improv.	Before	After	Improv.
Apte	51190520	50832440	0.7%	4628892	4270812	7.74%
Apte_2	52270696	50832440	2.75%	5709068	4270812	25%
Hp	10630100	10158272	4.46%	1799516	1327688	26%
Hp_2	10630100	10158272	4.46%	1799516	1327688	26%
Xerox	25168048	24769164	1.59%	6120728	5418868	6.86%
Xerox_2	25471024	24769164	2.76%	6120728	5418868	11%
Ami33	1829250	1591352	13%	672801	434903	35%
Ami49	50109768	45577000	9.05%	14664344	10131576	31%

Table 4.5: Estimated areas before and after optimization

demand profile along the Y axis. The height of the circuit is computed by adding the height of the unrouted floorplan and the maximum height of the corresponding wiring demand profile along the X axis. The estimated wiring + empty area is computed by subtracting the total module area from the total estimated chip area.

In order to evaluate the ultimate effect of the algorithms on the floorplans, we need to compare the routed floorplans before and after the optimization. Table 4.6 gives some statistics for the routed floorplans.

In the table, the total chip area and the empty+wiring area, before and after wiring space optimization, are given for each floorplan. The percentage of improvements were calculated for each kind of area. Although the improvement over the total chip area

Benchmark	Total Area			Wiring+Empty area		
	before	After	Improv.	Before	After	Improv.
Apte	51891508	51092072	1.54%	5329880	4530444	15.0%
Apte_2	54088336	51092072	5.54%	7526708	4530444	39.8%
Hp	11516940	11175868	2.96%	2686356	2345284	12.7%
Hp_2	11559338	11175868	3.32%	2728754	2345284	14.5%
Xerox	26796505	26511994	1.06%	7746209	7161698	7.55%
Xerox_2	29870244	26511994	11.24%	10519948	7161698	31.92%
Ami33	2390016	2273388	4.88%	1233567	1116939	9.54%
Ami49	51779094	50543038	2.39%	16333670	15097614	7.56%

Table 4.6: Actual areas of the routed floorplans before and after optimization

is not very significant, the improvement over the wiring area plus empty area, which ranges from 7.55% up to 39.8% is quite significant.

By comparing tables 4.5 and 4.6 we can see that the estimated area is consistent with the actual area. In some cases, such as APTE, the estimated area is close to the actual area. But in some other cases, such as Ami33, the difference between the estimated area and the actual area is considerable. There are several reasons for the errors in the estimate. First, our estimate only gives the minimum area of the floorplan. While estimating wiring space, for each net we assumed that there was only one horizontal wire and one vertical wire covering the horizontal and the vertical extent of the net's minimum bounding box. In reality, there can be several wires covering a certain interval of a bounding box. The second reason is that our estimate of empty and wiring space is one dimensional. That is, all the empty spaces and the wiring spaces are projected on to the X/Y axis to construct the corresponding profiles. Even though the empty space profile and the wiring space profile may match perfectly, this does not necessarily mean that all the empty space can be used for wiring. In reality, there will always some space left unused.

Chapter 5

Conclusions and Future Work

Most existing algorithms that take wiring into consideration during floorplanning take wire length as a measure for wiring. In this thesis, we presented two closely related algorithms aimed at minimizing wiring space in a circuit. One of them is the branch and bound algorithm which gives the optimal solution to the problem. The other is a heuristic algorithm derived from the branch and bound algorithm. The heuristic algorithm is not guaranteed to gives the optimal solution to the problem, but it can give a reasonably good solution in a practical amount of time for large circuits.

The experiment of applying the above algorithm to some benchmark circuits shows that the algorithms are effective. In all cases, the total chip area for routed floorplan was improved. An interesting phenomenon worth noting is that the heuristic algorithm was also applied to the small benchmark circuits, in most cases optimal solutions were obtained.

Some searching strategies aimed at speeding up the search process were also introduced, and experiments show that they are effective. The most impressive searching strategy is the bottle-neck-first search which reduced the searching time by up to 90% compared with ordinary depth-first search.

Also introduced in this thesis is a new method of estimating wiring space. Most of the existing wiring space estimation methods either estimate wire length or estimate wiring space based on statistics or previous design experience. The new method presented in this thesis gives some information about the distribution of wiring space along the X and Y axis. Therefore, it can give more guidance to the floorplanning process.

Although our wiring space estimating method can give some information about wiring space distribution, it is one dimension oriented. It can only give the distribution of empty space and wiring space along the X and Y axis. Even if the wiring space profile matches the empty space profile closely, it does not necessarily mean that the actual wiring space distribution will match the empty space distribution. The reason for this is that the space profiles describe space distribution in one dimension, while in reality the space is distributed in two dimensions. One way to solve this problem is to generalize our space estimating method to two dimensions so that it can give a more accurate estimate. Another way to solve the problem is to estimate channel density. In this way we can get a more accurate estimate of wiring space distribution.

Another possible application of our new algorithm includes using it in standard cell layouts. Since the cells are arranged in rows and the space between rows is used as channels for wiring, we can maintain a space profile for each channel: therefore, the two dimensional problem is eliminated.

Another possibility for future work is to generalize our algorithm so that, in addition to flipping the modules and swapping sibling rectangles, it can also take the orientation of the modules and the direction of slices into consideration. In doing this the total chip area could be further reduced.

Bibliography

[1] D. P. Lapotin and S. W. Director, "Mason: A Global Floor planning Approach for VLSI Design" in IEEE Trans. on CAD of ICs and Systems, Vol. CAD.5, No. 4, pp. 477-489, 1986.

[2] W. Dutton and L. Sha, "An Analytical Algorithm for Placement of Arbitrary Sized Rectangular Blocks" in Proc. of IEEE 22nd Design Automation Conference, pp. 602-608, 1985.

[3] C. L. Liu and D. F. Wong, "A New Algorithm for Floorplanning Design" in Proc. of IEEE 23rd Design Automation Conference, pp. 101-105, 1986

[4] P.G. Paulin and J.P.Knight, "Force-Directed Floorplanning in Automatic Data Path Synthesis" in Proc. of ACM/IEEE 24th Design Automation Conference, pp. 195-202, 1987.

[5] A. A. Azepieniec, "Integrated Placement/Routing in Sliced Layouts" in Proc. of 23rd Desgin Automation Conference, pp. 300-307, 1986.

[6] M. A. B. Jackson and E. S. Kuh, " Performance-Driven Placement of Cell Based IC's" in Proc. of ACM/IEEE 26th Design Automation Conference, pp. 370-375, 1989.

[7] A. Herrigel and W. Fichtner, "An Analytical Optimization Technique for Placement of Macro-Cells" in Proc. of ACM/IEEE 26th Design Automation Conference, pp. 376-381, 1989.

[8] G. Zimmerman, "A New Area and Shape Function Estimation Technique for VLSI Layouts" in Proc. of ACM/IEEE 25th Design Automation Conference, pp.60-65, 1988.

[9] W. M. Dai and E. S.kuh, "Simutaneous Floor Planning and Global Routing for Hierarchical Building Block Layout" in IEEE Trans. on CAD, Vol. CAD-6, No. 5, Sept. 1987.

[10] F. J. Kurdahi and A. C.Parker, "PLEST: A Program for Area Estimation for VLSI Integrated Circuits" in Proc. of IEEE Intl. Conference on Computer Aided Design, pp. 467-473, 1986.

[11] M. Pedram and B. Preas "Accurate Prediction of Physical Design Characteristic for Random Logic" in Proc. of IEEE Intl. Conference on Computer Design, pp. 100-108, 1988.

[12] W. M. Dai. B. Escherman, and E. S. Kuh. "Hierarchical Placement and Floorplanning in BEAR" in IEEE Trans. on Computer-Aided Design. Vol. 8. No. 12. 1989.

[13] A. A. El Gamal, "Two-Dimensional Stochastic Mod[illegible] [illegible]rconnection in Master Slice Integr[illegible]d Circuit" in IEEE Trans. [illegible]it and System Design. Vol. CAS-28, No. 2, 1981.

[14] A. A. Gamal El and Z. A. Ayed, "A Stochastic Model for Interconnection in Custom Integrated Circuits" in IEEE Trans. on Circuit and System, Vol. CAS 28, No. 9, 1981.

[15] D. Jepsen and D. Gelatt. "Macro Placement by Monte Carlo Annealing" in Proc. of IEEE Intl. Conference on Computer Design, pp. 495-498, 1983.

[16] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing" in Sicence, Vol. 220, No. 4598, pp. 671-680, 1983.

[17] B. Lokananthan and E. Kinnen, "Performance Optimized Floorplanning bu Graph Planarization" in Proc. of 26th Design Automation Conference, pp. 116-121, 1989.

[18] S. Prasitjutrakul and W. J. Kubitz, "Path-Delay Constrained Floorplanning: A Mathematical Programming Approach for Initial Placement" in Proc. of 26th Design Automation Conference [illegible]. 364-369, 1989.

[19] C. Sechen and A. Sanjiovanni-Vincentelli, "The TimberWolf Placement and Routing Package" in IEEE Journal of Solid State Circuits, Vol. sc-20. No. 20. pp. 510-522, 1985.

[20] L. Stockmeyer, "Optimal Orientation of Cells in Slicing Floorplan Designs" in Information and Control, Vol. 59, pp. 91-101, 1983.

[21] A. R. Newton and A. L. Sangiovanni-Vincentelli, "Computer-Aided Design for VLSI Circuits" in IEEE COmputer, Vol. 19, No. 4, pp. 38-63, 1986.

[22] H. Onodera, Y. Taniguchi, and K. Tamaru, "Branch-and-bound Placement for Building Block Layout" in Proc. of 28th Design Automation Conference, pp. 433-439, 1991.

[23] L. Thomas and R. Miller, "A Robust Framework for Hierarchical Floorplanning with Integrated Global Wiring" in Proc. of IEEE Intl. Conference on Computer Aided Design, pp. 148-151, 1990.

[24] A. A. Szepieniec and R.Otten, "The genealogical approach to the latout problem" in Proc. 17th Design Automation Conference, pp. 535-542, 1980.

[25] T. Ohtsuki, N. Suziyama, and H. Kawanishi, "An optimization technique for integrated circuit layout design" in Proc. ICCST-Kyoto, 1970, pp.67-68.

[26] S. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs" in Bell Systems Tchnical Journal Vol. 49, No. 2, pp. 291-307, Feb. 1970.

[27] S. Kirkpatric, C. D. Gelatt and M. P. Vecchi, "Optimization by Simulated Annealing" in Science, Vol. 220, No. 4598, pp. 671-680, 13 May 1983.

[28] W. M. Dai, T. Asano, and E. S. Kuh, "Routing Region Definition and Ordering Scheme for Building-Block Layout" in IEEE Trans. Computer Aided Design, Vol. CAD-4, No. 3, pp. 189-197, July 1985.

[29] D. W. Hanson "Interconnection Analysis" in Physical Design Automation of VLSI Systems, Chapter 2, pp. 31-64. Benjamin Cummings, Menlo Park, CA, 1988.