Efficient Approaches in Interconnect-driven Floorplanning

LAI Tsz Wai

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

August, 2003

Abstract

As technology moves into the deep submicron era, the complexity of VLSI circuit design grows rapidly. Interconnect optimization has become an important concern in floorplanning today. In this thesis, two fast and effective approaches in interconnect-driven floorplanning are proposed.

The first approach measures the wiring congestion as the wire density on the boundary of different regions in a floorplan. These regions can be defined naturally by using the twin binary trees (TBT) floorplan representation. In order to increase the efficiency of our floorplanner, a fast algorithm for the least common ancestor (LCA) problem is used to compute the wire density. From the experimental results, the number of unroutable wires can be reduced efficiently by using this kind of interconnect-driven floorplanning.

The second approach is called the simple buffer planning method. It aims at improving the feasibility of buffer insertion of the output floorplan solution. It counts the number of blocked nets without computing the exact buffer locations. Dynamic programming and a table look-up approach are used to decide whether a net is blocked. The decision can be made in constant time with a linear pre-processing time. We combine the simple buffer planning method with the wire density evaluation model in a two-stage simulated annealing process. Experimental results have shown its efficiency in reducing the number of unroutable wires due to wiring congestion and unsuccessful buffer insertion.

摘要

隨著科技發展逐漸遠離次微米的紀元，超大規模集成電路設計的複雜性正在迅速增長，這個現象在模件間的互連最為明顯。故此，在現今的佈局規劃中，優化互連電路已經成為一個很重要的考慮因素。在這篇論文中，我們將會發表兩個既簡單而有效率的優化互連電路佈局規劃方案。

在第一個方案中，我們以孿生二叉樹作為佈局規劃的表示法。在佈局規劃中，不同的區域會被孿生二叉樹表示法自然地定義出來。在這些區域的邊界上，我們會以電路密度來量度線路擠擁的情況。為了增加佈局規劃的效率，我們用了一個解決「最近公祖先」的快速算法來計算線路密度，實驗證明這個方法可以在短時間內減少不可繞的線路數目。

第二個方案叫做「簡單緩衝編制方法」，這個方案旨在提高緩衝嵌入的可行性。這個模型會利用以格子為基礎的傳統方法將佈局劃分為二維空間的格子結構，這個模型能夠在不需準確計算最佳緩衝位置下，數算出有多少個被阻塞的網絡。我們可以運用動態規劃和表格查詢的方法去決定一個網絡是否被阻塞，而借助一個線性時間的預處理，我們就可以在常數時間內作出這個決定。我們利用二級模擬降溫法將電路密度評估方案和這個簡單的緩衝編制方案一起放在佈局規劃中，實驗結果證明這些方案能有效地減少那些因為線路擠擁和未能嵌入緩衝所做成的不可繞線路。

Acknowledgments

First of all, I must thank my supervisor, Prof. Evangeline Fung Yu Young. She is a patient supervisor who provides ideas, guidance, advices and encouragements when necessary. I have learnt a lots from her in these two years.

Secondly, I would like to thank Prof. Chris C. N. Chu for providing us the idea of using an efficient least common ancestor algorithm for our wire density model. Without this idea, we cannot achieve a fast runtime of the floorplanner.

Thirdly, I need to thank Mr. Joseph Chun Sing Lau for providing a simple global router and a program to generate large circuits for the experiments. At the mean time, I also need to thank Mr. Bruce Chiu Wing Sham and Mr. Keith Wai Chiu Wong for providing the details of their floorplanners, so that I can make a reasonable comparison in the experiment.

Finally, I have to thank all of my colleagues. Without them, I will not have this enjoyable school life in my memory.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In 1960s, *integrated circuit (IC)* technology was widely used in computers for microprocessor, memory module and other interface chips. With the advances of the *Very Deep Sub-Micron (VDSM)* technology, IC has evolved from *Small Scale Integration (SSI)* ,which consists of a few transistors, to *Very Large Scale Integration (VLSI)*, which consists of millions of transistors. According to the Moore's Law [10], it was predicated that the number of transistors in an IC would double every 1.5 years. Therefore, it is possible to build a processor with billions of transistors running at several GHz. In the coming years, the technology of VLSI will continue to scale down as shown by the prediction in Table 1.1. The transistors will become smaller, less resistive, faster and conducting more electricity. Also, the interconnections in a chip will become longer and denser in the future. This rapid growth has brought many challenges to VLSI circuits and its automation.

Year	1997	1999	2001	2003	2006	2009
Technology (μm)	0.25	0.18	0.15	0.13	0.1	0.07
Number of transistors	11M	21M	40M	76M	200M	520M
Across chip clock (MHz)	750	1200	1400	1600	2000	2500
Chip size (cm^2)	3.00	3.40	3.85	4.30	5.20	6.20
Wiring levels	6	6-7	7	7	7-8	8-9

Table 1.1: Technology roadmap [9].

1.1 VLSI Design Cycle

From designing a chip specification to producing a packaged VLSI chip, several steps are involved in the VLSI design cycle as shown in Figure 1.1. These steps include system specification, architectural design, functional design, logic design, circuit design, physical design, fabrication, and packaging and testing.

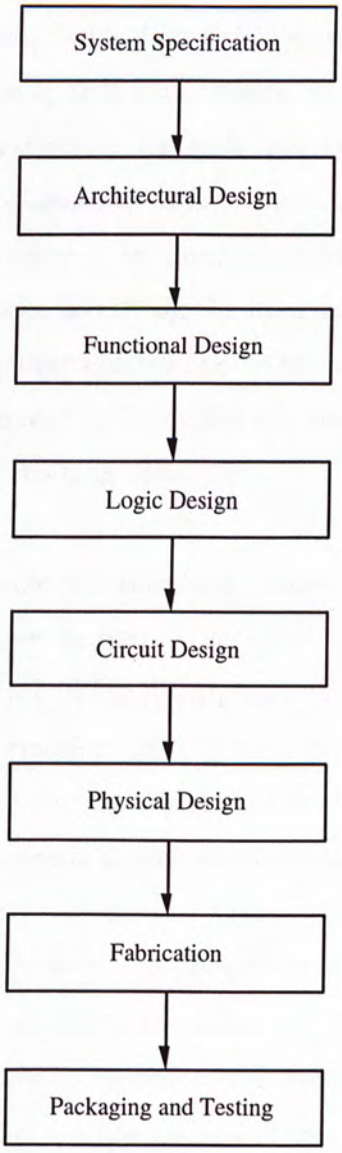


Figure 1.1: A simple VLSI design cycle [1].

System specification is a high level descriptions of the requirements of the design. It specifies the size, speed, power, and functionality of the VLSI system. After defining the system specification, we can design the basic architecture of the system in *architectural design.* In architectural design, a *Micro-Architectural Specification (MAS)* is obtained by making decisions on the number of ALUs and floating point units, the number and structure of pipelines, the size of caches, and whether using *Reduced Instruction Set Computer (RISC)* or *Complex Instruction Set Computer (CISC)*, etc. Given a MAS, we can predict the system performance, die size, power consumption, etc. After the architectural design, the main functional units of the system and their interconnections are identified in *functional design.* In functional design, a timing diagram of the units is obtained. It specifies the input, output and timing of each unit without specifying the internal details of each unit. The information of the timing diagram can help to make improvement on the overall design process. It can also facilitate the efficiency of debugging the whole system before moving to the later steps.

In *logic design*, the boolean expressions, control flow, word width, register allocation of the design are derived. These logical information is described in a *Register Transfer Level (RTL)* description, which consists of boolean expressions and timing information. RTL is then expressed in a *Hardware Description Language (HDL)*, such as VHDL and Verilog. Using this description, the logic design of the system is simulated and tested to verify its correctness. Based on the logic design, the circuit representation is developed in *circuit design.* By considering the speed and power requirement of the design, the boolean expressions in logic design are converted into a detailed circuit diagram, which is called a *netlist.* In this circuit representaion, the cells, gates, transistors and interconnections are defined. Then, the circuit representation will be converted into a geometric representation, which is called a *layout*, in

physical design. Both the circuit elements and interconnections are expressed in geometric representation. Before the fabrication step, various verifications and validation checks are performed on the layout. After a final checking of the layout, a chip is produced in the *fabrication* step. Finally, the chip will be packaged and then tested in *packaging and testing* to ensure that it functions well and can meet all the system specifications.

In this thesis, we will focus on physical design in the design cycle. Due to the huge number of circuit elements, interconnects and system requirements, physical design is further partitioned into several phases to reduce the design complexity and produce better management.

1.2 Physical Design Cycle

In VLSI design cycle, physical design converts a circuit diagram into a circuit layout. A layout is a geometric representation of the circuit, so that the circuit can be converted into a photo-lithographic mask in fabrication. Due to the complexity of VLSI chip design, physical design is partitioned into several phases for better management as shown in Figure 1.2. These phases include partitioning, floorplanning, placement, routing, compaction, and extraction and verification. If the design requirements, like timing and size, cannot be achieved, we may need to go back to some phases or even the beginning of the cycle to repeat the process again.

Due to the huge number of transistors contained in a VLSI chip in the deep submicron era, it is difficult to provide an efficient *computer aided design (CAD)* tools to solve the physical design problem. Therefore, the first phase in the physical design cycle will be *circuit partitioning.* In this phase, a large

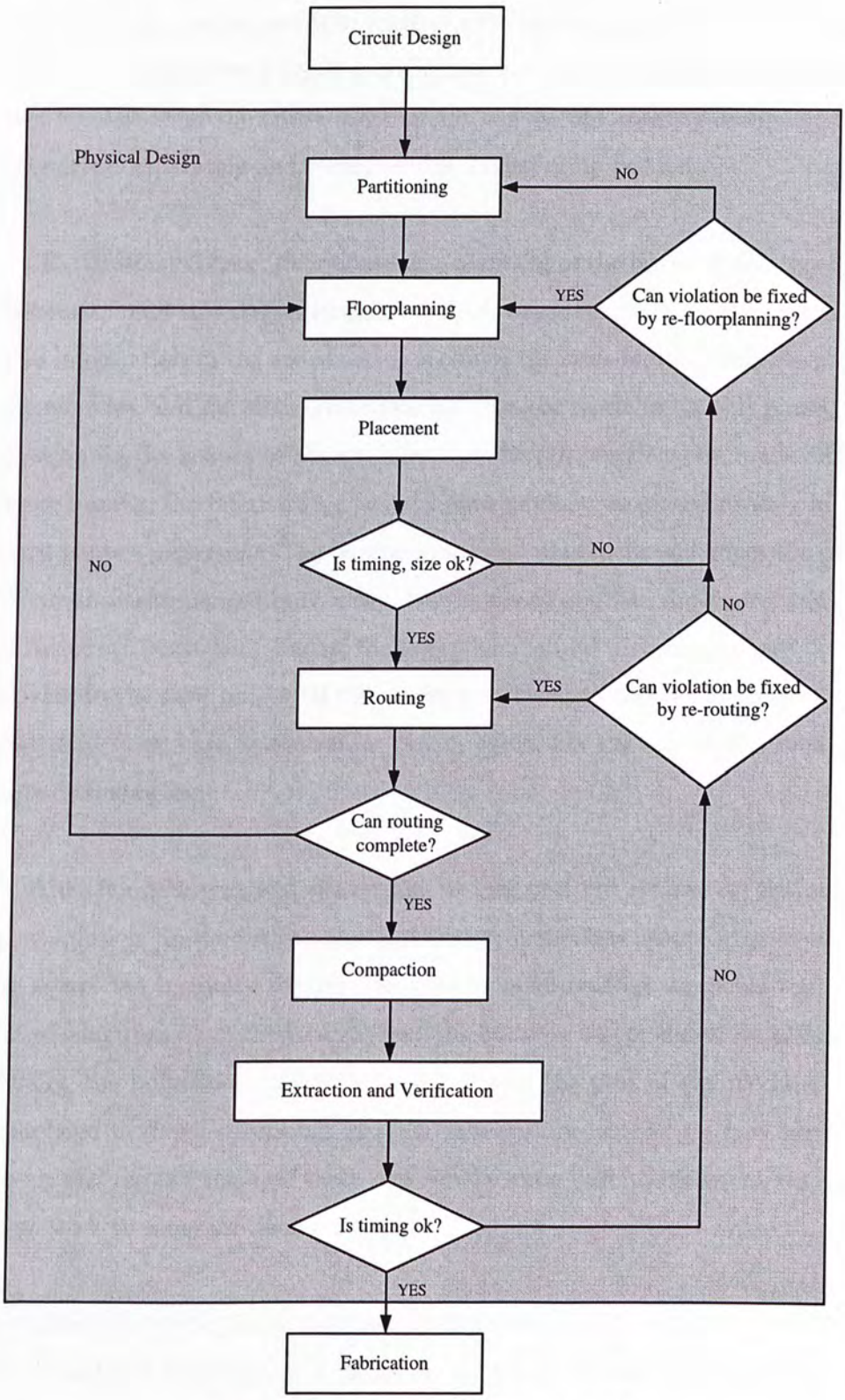


Figure 1.2: Physical design cycle [1].

circuit will be partitioned into smaller sub-circuits (modules) recursively, so that the problem size is small enough to be solved efficiently. After partitioning, the layout of the entire large circuit can be obtained by designing each sub-circuit separately and recursively in a bottom-up fashion.

In the second phase, *floorplanning*, a planning of the layout which can optimize the circuit size and performance is made based on the circuit specification. The information in the specification includes the sizes and possible shapes of the modules, and the interconnections between the modules. In this phase, the decision on the shapes of the modules and the pin positions are made. After floorplanning, the internal logic cells in each module are placed exactly in the third phase - *placement*. The floorplanning and placement will affect the overall circuit performance significantly. It is better to evaluate the timing and size of the circuit accurately during the floorplanning and placement phase before moving to the later phases. If the circuit requirements cannot be achieved, we may need to go back to the earlier phases again, like partitioning or even the logic designing step.

After floorplanning and placement, we can perform *routing* on the placement solution. In the routing phase, the interconnections between the modules are completed by global routing and then detailed routing. In global routing, the routing regions of the interconnections between the modules are planned. Finally, the point-to-point connections between the pins of the modules are completed in detailed routing. If some nets are unroutable, we may need to rip-up and reroute them. If there still remain some unroutable wires, we need to go back to some previous phases.

After the routing phase, most geometric information have been worked out. The layout will be compressed to obtain a smaller chip size in *compaction*. In

the *extraction and verification* phase, the layout will be checked for any design rule violation. A circuit is extracted from the layout of which the performance and reliability will be verified before the fabrication step. It is observed that the quality of the solutions in the earlier phases plays an important role to increase the efficiency of the physical design cycle. These critical phases include partitioning, floorplanning and placement. In this thesis, we will focus on the floorplanning problem in the VLSI physical design cycle.

1.3 Floorplanning

In the VLSI design cycle, floorplanning plays an important role. After circuit partitioning, the initial specification for each module can be obtained. It includes the areas of the modules, the possible shapes (aspect ratios) of the modules, the number of terminals (pins) of the modules and the netlists (interconnections between the modules) of the circuit. In the floorplanning phase, we are going to plan the position and shape of each module. The floorplan obtained should optimize the circuit in terms of chip area, total wirelength, routability, delay of critical path and heat dissipation. Minimizing the chip area can reduce the deadspace and increase the yield. As technology moves into the deep submicron era, circuit sizes and complexities grow rapidly. The interconnections between modules will become longer and denser in the future, so minimizing the interconnect cost in floorplanning has become ever more important than before. The following defines the floorplanning problem:

Problem Formulation 1.1 (Floorplanning) *Given a set of modules (M_1, M_2, M_3, ... M_n) with areas (A_1, A_2 ,A_3, ... A_n) respectively, each module M_i is associated with two aspect ratio bounds a_{ri} and a_{si}, that specify the lower and upper bound of the aspect ratio of module M_i respectively such that $a_{ri} \leq \frac{h_i}{w_i} \leq a_{si}$ where h_i and w_i are the height and width of M_i respectively. We want to*

obtain a non-overlap packing of the set of modules by finding the position (x_i, y_i) *and dimension* (w_i, h_i) *for each module* M_i *so that the circuit performance can be optimized.*

As the floorplanning problem is proved to be NP-complete, different approaches have been employed to solve it. These approaches include analytical approach, simulated annealing approach, genetic algorithm approach, force directed approach and constraint based approach.

In 1991, a linear programming approach [11] is proposed. It is based on a mixed integer linear program. The placement constraints are treated as linear functions with integer variables. For a problem with n blocks, the number of linear functions is $O(n^2)$ and this method is only practical for circuit with at most $n = 10$ modules. In 1998, a convex formulation [12] is proposed to solve the floorplan area minimization problem. By handling the aspect ratio constraints indirectly, the number of variables and constraints can be reduced.

Besides solving the floorplanning problem by linear programming, it can also be solved by stochastic searching methods. Many floorplanners are based on the simulated annealing approach [2, 3, 13, 14, 15, 16, 17, 18] or the genetic approach [5, 19]. Simulated annealing [2, 3] is commonly used in floorplanning. The pseudo-code of a general simulated annealing algorithm is shown in Figure 1.3. In simulated annealing based floorplanner, a floorplan is represented by a floorplan representation (for example, sequence representation or tree representation) and the quality of a floorplan is evaluated by a cost function. The annealing process starts with an initial floorplan solution x_0 at temperature T_0. In each iteration, a small change will be made to the solution and the temperature T will be cooled down according to the cooling rate c. Each floorplan solution will be evaluated by the cost function. If there is a

cost reduction, the modified solution will be accepted as a new solution. Otherwise, the acceptance of a worse solution will be determined randomly with a probability of $e^{-k\Delta f/T}$ where k is a constant to adjust the order of magnitude and Δf is the drop in the value at the cost function before and after the modification. There will be a higher chance to accept a worse solution at high temperature. This feature allows the annealing process to climb out of local minima. Finally, the annealing process will terminate when the temperature is cooled down to the terminating temperature T_t.

Another stochastic searching approach is genetic algorithm [5]. In this approach, evolutionary mechanism is applied. The pseudo-code of a general genetic algorithm is shown in Figure 1.4. The algorithm starts with a set of initial solutions called population. By using two types of genetic operators, crossover and mutation, a better population can be obtained iteratively by means of evolution. Here, evolution refers to the improvement of the population quality. In crossover, a new solution is created by combining two solutions in the population. In mutation, a solution is picked randomly and a small change is applied on it.

Many floorplanners use simulated annealing with a proper floorplan representation. Most of the floorplan representations are designed to represent the topological relationship between the modules. A well designed floorplan representation can increase the efficiency and effectiveness of the simulated annealing process by providing a faster floorplan realization and solution space with less redundancy. In the next section, different types of floorplan representations will be introduced.

```
Simulated Annealing(N, T_0, T_t, c)
    x = x_0          /* Initial solution */
    T = T_0          /* Initial temperature */
    While(T ≥ T_t)
        For loop = 1 to N
            x' = mutate(x)
            Δf = cost(x') - cost(x)
            r = random number between 0 and 1
            If(Δf < 0 or r < exp(-kΔf/T))
                x = x'
            Endif
        Endfor
        T = T × c        /* Cool Down */
    Endwhile
    return x
```

Figure 1.3: Simulated annealing [2, 3, 4].

```
Genetic Algorithm(P, R_c, R_m)
    X = {x_1, x_2, ..., x_P}        /* Initial Population */
    While(stopping criterion is not met)
        X' = ∅
        /* Create children by crossover */
        While(number of children created < P × R_c)
            select two solutions, x_i and x_j, from X
            x' = crossover(x_i, x_j)
            X' = X' ∪ {x'}
        Endwhile
        select P solutions from X ∪ X' as a new population and call it X
        While(number of solutions mutated < P × R_m)
            select one solution x_k from X
            x' = mutate(x_k)
            X' = X' ∪ {x'}
        Endwhile
        X = X'
    Endwhile
    return the best solution in X
```

Figure 1.4: Genetic algorithm [5, 4].

1.3.1 Types of Floorplan and Floorplan Representations

There are three main kinds of floorplan: slicing, non-slicing and mosaic floorplan as shown in Figure 1.5. A *slicing* floorplan is shown in Figure 1.5(a). It is a floorplan that can be obtained by recursively dividing a rectangle into two by using either a horizontal or a vertical cut. A widely used slicing floorplan representation is *normalized Polish expression* [20], which is proposed in 1986. It used a binary slicing tree to represent the slicelines and the modules. It can represent a slicing floorplan with no redundancy and the size of the solution space is $O(n!2^{3n-3}/n^{1.5})$ where n is the number of modules.

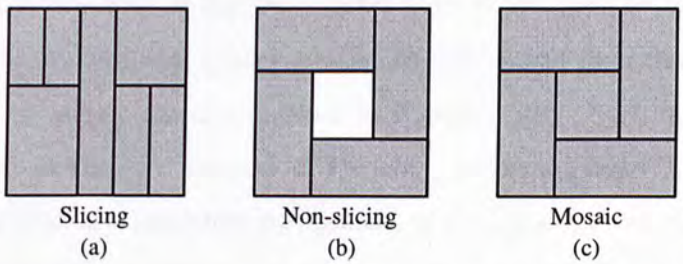


Figure 1.5: Examples of the three main kinds of floorplans.

A *non-slicing* floorplan is any general floorplan that is not necessarily obtained by recursively dividing a rectangle into two as shown in Figure 1.5(b). It is the most general kind of floorplan. There are many representations proposed recently for this kind of general floorplan. A *sequence pair (SP)* representation for non-slicing floorplan is proposed in paper [13]. It uses a pair of sequences to represent the topological relations between the modules. It is widely used recently because of its simplicity. However, its solution space is $O((n!)^2)$, which is very large and redundancies exist. After SP is proposed, another non-slicing floorplan representation called *bounded-sliceline grid (BSG)* is proposed in paper [21]. An n-by-n grid structure is used to place n modules. The size of its solution space is $O(n!C(n^2, n))$, which is also very large and contains a lot of redundancies. For both SP and BSG, constraint graphs are needed to be

constructed for floorplan realization, and it is a time consuming process. In 1999, an *O-tree* representation with a significant reduction in the size of the solution space $O(n!2^{2n-2}/n^{1.5})$ is proposed in paper [22]. A floorplan can be realized directly from its O-tree representation in linear time. In 2000, a *B*-tree* representation which is very similar to O-tree and shares the same advantages is proposed in paper [14]. Although both O-tree and B*-tree have a very small solution space and floorplan realization can be preformed very efficiently, they can only represent partial topological information and module dimensions are needed to define the topological relationships between the modules.

In 2000, a new kind of floorplan called *mosaic floorplan* is proposed in paper [6]. Mosaic floorplan is very similar to non-slicing floorplan except that it contains no empty rooms as shown in Figure 1.5(c). Each module corner, except those at the four corners of the chip, is formed by a T-junction. In mosaic floorplan, the non-crossing segment of a T-junction can slide along the crossing segment and represents the same packing as shown in Figure 1.6. The first mosaic floorplan representation is *corner block list (CBL)* which is proposed in paper [6]. In this paper, a floorplan is represented by three sequences. The size of the solution space of CBL is $O(n!2^{3n})$ [15]. Floorplan realization of CBL can be performed in linear time. However, not all CBL will correspond to a valid floorplan. Additionally, as mosaic floorplan cannot represent those non-slicing floorplans with empty rooms, an *extended CBL (ECBL)* representation is proposed in paper [23], that includes dummy blocks of zero area in the set of modules to represent all non-slicing structures. In paper [7], another mosaic floorplan representation - *twin binary trees (TBT)* is proposed. It has a smaller solution space of $O(n!2^{3n}/n^{1.5})$ and has an one-to-one mapping to all mosaic floorplan. Based on TBT, a representation called *twin binary sequences (TBS)* is proposed in paper [15], that can generate every non-slicing floorplan uniquely and efficiently without redundancy in linear time. As a new

and promising floorplan representation, TBT is used in our research work and in our floorplanners. In conclusion, the categories of floorplans can be summarized as in Figure 1.7. Slicing floorplan is a subset of mosaic floorplan, while mosaic floorplan is a subset of general floorplan.

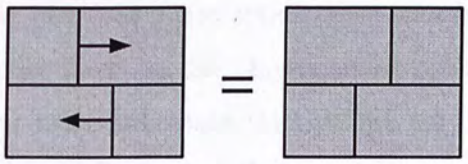


Figure 1.6: A property of mosaic floorplan [6].

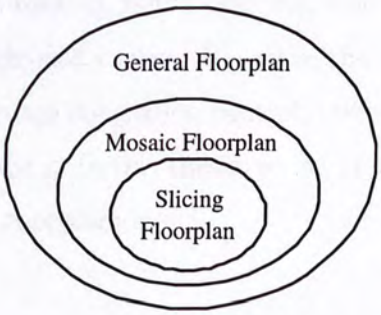


Figure 1.7: Categories of floorplans [7].

1.3.2 Interconnect-driven Floorplanning

Traditionally, most floorplanners [19, 24, 25, 13, 21, 22, 6, 15] aim at minimizing the chip area so as to increase the yield. However, as technology moves into the deep submicron era, the number of transistors, the complexity of a circuit and the number of interconnections between the modules are increasing rapidly. The interconnections between modules will become longer and denser in the future. In some advanced systems, a significant portion of about 80% of the clock cycle is consumed by interconnections [9]. In the international technology roadmap for semiconductors in 2001 [8], it has been predicted that the

delay of global wire will continue to increase with the scaling down of the technology, especially for the wires without buffer insertion as shown in Figure 1.8. It has been shown that the interconnect delay for a wire without repeater insertion will increase quadratically as the wirelength increases, while it will only increase linearly with proper repeater insertions [26, 27, 28]. Also, it is shown that the delay of a 2cm global interconnect can be reduced by a factor of 7× by optimal buffer insertion [29]. Interconnect optimization in floorplanning has become ever more important than before. As floorplanning is at the beginning phase of the VLSI physical design cycle, an interconnect-optimized floorplan will favor the applicability and performance of the later stages like global routing, detailed routing, buffer insertion, and, most importantly, allow timing closure to be achieved earlier. Recently, the major approaches for interconnect optimization are congestion control, buffer planning, device sizing, and wire sizing and spacing. In this thesis, we will focus on congestion control and buffer planning in floorplanning.

There are many congestion optimization floorplanners [30, 31, 32, 16, 33, 34, 35] proposed in recent years. In 1999, a multi-stage simulated annealing approach is proposed in paper [30] to address the congestion control problem. In interconnect planning, simple geometry routing is performed based on L-shaped and Z-shaped wires. A three-stage simulated annealing approach is used to combine the three different interconnect estimation methods. These three methods are the half-perimeter bounding box wirelength estimation, L-shaped routing and Z-shaped routing. In 2000, Chang et al. proposed a four-stage simulated annealing approach for interconnect-driven floorplanning in paper [32]. In their multi-layer global wiring planning, interconnect topology optimization, layer assignment, buffer insertion, and wire sizing and spacing

are performed. In addition, it also provides an optional post-processing optimization for the resulting floorplan. In 2001, Lou et al. proposed a probabilistic approach to estimate routing congestion in paper [34]. The probabilistic model computes all possible ways that a router can route a net. Congestion is estimated by computing the probabilistic track usage of each grid. In 1999, Wang et al. studied the behavior of congestion minimization in paper [31]. It is shown that the congestion cost alone is a poorly behaved cost function. There is a correlation between wirelength and congestion, and we should need to consider them together. A new objective called overflow with look-ahead is proposed to reduce congestion. In 2000, Wang et al. pointed out in paper [16] that the bounding box router used in paper [31] cannot measure congestion accurately, so a realistic global router is used instead. In addition, two post-processing algorithms, the flow-based cell-centric algorithm and the net-centric algorithm, are proposed.

Reducing wiring congestion in floorplanning is not enough. We should also consider buffer insertion so that timing closure can be achieved as early as possible [36, 37] in the design cycle. In 1999, Cong et al. proposed the concept of feasible region for buffer insertion in paper [36]. A buffer clustering technique is used to plan the buffer locations effectively. In paper [37], the concept of independent feasible region is proposed. A similar clustering technique is used to place the buffers greedily in the available space. In 2000, Tang and Wong proposed to make use of the max-flow min-cut algorithm in network flow to place the buffers in polynomial time in paper [38]. In paper [39], Dragon et al. reduced the buffer planning problem to an integer multi-commodity flow problem. It routes the net by using available buffer blocks, such that the separation x between adjacent buffers are controlled to be within a range $[low, up]$. In 2001, Alpert et al. modelled the buffer planning problem in a tile graph in paper [40]. A four-stage heuristic is proposed to plan

the wires and the buffers. These four stages are initial steiner tree construction, wire congestion reduction, buffer assignment and final post-processing. In 2002, a probabilistic model to estimate wiring congestion and buffer planning is proposed in paper [17]. The estimations are based on the demand and supply analysis of the routing and buffer resources. Dynamic programming is used to estimate the buffer usage. In paper [18], the best buffer locations with respect to the variable interval buffer insertion constraint are computed using dynamic programming. After the buffer locations are decided, probabilistic analysis is applied to estimate the wiring congestion.

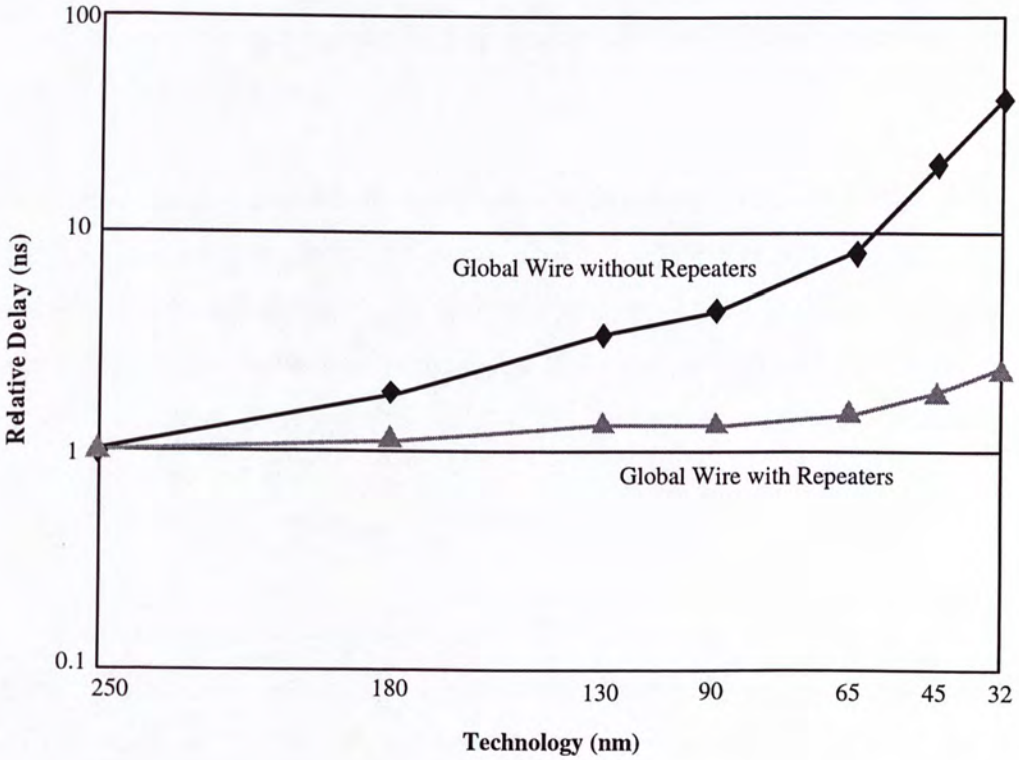


Figure 1.8: Relative delay for global wiring versus feature size [8].

1.4 Motivations and Contributions

As stated before, floorplanning plays an important role in the VLSI physical design cycle. It sets up a ground work for a good layout. A good layout is important for the later stages in the physical design cycle. In this deep sub-micron era, the number of transistors and interconnects are growing rapidly. The wires will become longer and denser in the future. More routing space and buffers are needed to ensure timing closure and design convergence. If we did not carefully plan the routes of the nets and reserve sufficient spaces for buffer insertions when we were designing a circuit layout, there would be a high chance of having a lot of unroutable wires in the routing phase. As a result, we need to plan the routing spaces and buffer locations as early as possible in floorplanning.

Our research is focused on interconnect-driven floorplanning which includes congestion control and buffer planning. Firstly, we have reviewed several literatures about floorplanning. They include floorplan representations, congestion control and buffer planning in floorplanning. As our methods are based on simulated annealing, we need to study some recent floorplan representations, and choose one of them to be used in our floorplanners. We have used the twin binary trees representation (TBT) [7] in our research work finally.

We make use of the characteristic of TBT and propose a novel congestion estimation method. It is called *wire density*. It provides a simple, indirect but efficient congestion evaluation model other than the complicated grid-based approach [30, 16]. Instead of estimating the congestion at each grid using global routing, we evaluate congestion as the wire density passing through the boundary of different regions in a floorplan. It is because a floorplan, that has high wire density on average, has a greater chance of having congestion

problem. In addition, we have made use of a fast algorithm [41] for the *least common ancestor (LCA)* problem to compute the wire densities at different regions efficiently. Experimental results have shown that an interconnect optimized floorplan of a complex circuit can be obtained in less than three minutes.

Thirdly, we propose a table look-up approach to solve the buffer planning problem. This simple buffer planning evaluation method aims at improving the feasibility of buffer insertions of the floorplan solution. Also, it aims at considering buffer planning efficiently without accurately computing the best possible buffer locations. In our method, we are going to determine whether a net can have all its buffers inserted. In order to achieve an efficient evaluation, we will use dynamic programming and a table look-up approach to obtain such buffer planning information in constant time. In this approach, a floorplan is divided into a two-dimensional grid structure. The concept of *feasible grid* is introduced. By accessing the look-up tables, we can decide whether a net is blocked because of buffer insertion failure immediately. Together with the congestion control by the wire density evaluation model, experimental results have shown the efficiency and effectiveness of our floorplanner in reducing the number of unroutable wires.

1.5 Organization of this Thesis

This thesis is organized as follows. After giving an introduction and background information in this chapter. We will give a literature review on floorplanning. We will review on different floorplan representations in Chapter 2, and different approaches in congestion estimation and buffer planning in Chapter 3. After the literature review on related research areas, our proposed interconnect-driven floorplanners will be presented. In Chapter 4, details of the wire density model to evaluate congestion will be given. In Chapter 5,

the idea and implementation of our simple buffer planning method will be discussed. We will also combine the wire density model and the simple buffer planning method in our floorplanner. Experimental results of this floorplanner and other interconnect-driven floorplanners [17, 18] will be compared. Finally, a conclusion is drawn in Chapter 6.

Chapter 2

Literature Review on Floorplan Representation

In floorplanning, the characteristics of a floorplan representation can greatly affect the performance of a searching process in terms of size of the solution space, memory usage and ease of floorplan realization for evaluation. There are three kinds of floorplans: slicing, non-slicing and mosaic floorplan. In this chapter, representations for these three kinds of floorplans will be reviewed.

2.1 Slicing Floorplan Representation

2.1.1 Normalized Polish Expression

A slicing floorplan can be obtained by cutting the chip recursively into rectangular modules using horizontal and vertical slicelines. In 1986, Wong and Liu described this kind of hierarchical structure by a rooted binary tree, called *slicing tree*, in paper [20]. In this slicing tree, an internal node is labelled by a '$*$' or a '$+$' to represent a vertical or horizontal sliceline respectively while the leaf nodes corresponding to the circuit models are labelled with the module names as shown in Figure 2.1. A *Polish expression* can be obtained by performing a post-order traversal on the slicing tree. A *normalized Polish expression* is

defined as a Polish expression with no consecutive $*$'s or $+$'s. It is shown that a normalized Polish expression can be obtained by performing a post-order traversal on a slicing tree, which is constructed by always representing the cuts from right to left and from top to bottom. In their paper, it is shown that there exists a one-to-one correspondence between the normalized Polish expression and the slicing floorplan. The size of the solution space and the memory usage of this representation is $O(n!2^{5n-3}/n^{1.5})$ and $\theta(n(\lceil lgn \rceil + 1) - 1)$ bits respectively where n is the number of modules. Also, a slicing floorplan can be realized from its normalized PE in $O(n)$ time.

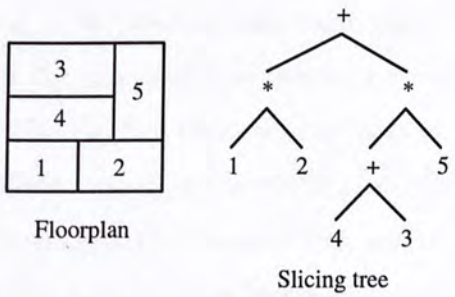


Figure 2.1: An example of a normalized Polish expression.

2.2 Non-slicing Floorplan Representations

2.2.1 Sequence Pair (SP)

In 1995, Murata et al. proposed a non-slicing floorplan representation called *sequence pair (SP)* in paper [13]. A sequence pair of a set of modules is a pair of combinations of the module labels. For example, $s = (abcd, bacd)$ is a sequence pair of the set of modules $\{a, b, c, d\}$. We can derive the relative positions between the modules from a sequence pair s by the following rules:

1. If $s = (\cdots a \cdots b \cdots, \cdots a \cdots b \cdots)$, module b is on the right side of module a.

2. If $s = (\cdots a \cdots b \cdots, \cdots b \cdots a \cdots)$, module b is below module a.

A sequence pair can be converted into a floorplan layout by constructing a pair of constraint graphs which represent the horizontal and vertical placement relationships. Each constraint graph has a source and a sink to denote the chip boundaries. In the horizontal constraint graph $G_H(V_H, E_H)$, the source and the sink represent the left-most and the right-most boundaries of the chip respectively while those of the vertical constraint graph $G_V(V_V, E_V)$ represent the bottom-most and the top-most boundaries of the chip respectively. A directed edge (u, v) in $G_H(V_H, E_H)$ denotes that module u is on the left side of module v, and its weight $w(u, v)$ is the width of module u (w_u). Similarly, a directed edge (u, v) in $G_V(V_V, E_V)$ denotes that module u is at the bottom of module v, and its weight $w(u, v)$ is the height of module u (h_u). The position of each module can be obtained by finding the longest path from the source to each vertex in $G_H(V_H, E_H)$ and $G_V(V_V, E_V)$. An example is shown in Figure 2.2. In their paper, the two constraint graphs can be constructed directly from the sequence pair by the following rules:

1. If $s = (\cdots a \cdots b \cdots, \cdots a \cdots b \cdots)$, add a directed edge (a, b) weighted w_a to G_h.

2. If $s = (\cdots a \cdots b \cdots, \cdots b \cdots a \cdots)$, add a directed edge (b, a) weighted h_b to G_v.

The time complexity of this floorplan realization is $O(n^2)$ where n is the number of modules. However, it can be optimized to $O(nloglogn)$ in paper [42].

The memory usage and the size of the solution space of the sequence pair representation are $\theta(2n(\lceil lgn \rceil))$ bits and $O((n!)^2)$ respectively.

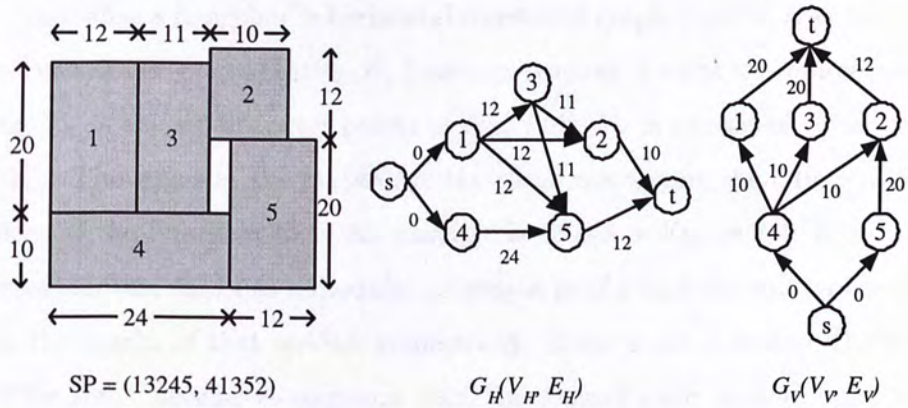


Figure 2.2: An example SP and its corresponding constraint graphs.

2.2.2 Bounded-sliceline Grid (BSG)

Nakatake et al. proposed another non-slicing floorplan representation using a meta-grid structure in paper [21] in 1996. This meta-grid is called *bounded-sliceline grid (BSG)*. The circuit modules are placed in the rooms defined by the BSG structure U_{BSG}.

Definition 2.1 *A **BSG** structure U_{BSG} is constructed by a set of horizontal line segments $H_{i,j}$ and a set of vertical line segments $V_{i,j}$ as follows:*

$$U_{BSG} = \begin{array}{l} \{V_{i,j} \mid i,j \in \mathbb{Z}^+ \text{ and } i+j \text{ is an even number}\} \\ \bigcup \ \{H_{i,j} \mid i,j \in \mathbb{Z}^+ \text{ and } i+j \text{ is an odd number}\} \end{array}$$

where

$$H_{i,j} = \{(x,y) \mid x,y \in \mathbb{R}, i-1 < x < i+1 \text{ and } y = j\}$$
$$V_{i,j} = \{(x,y) \mid x,y \in \mathbb{R}, x = i \text{ and } j-1 < y < j+1\}$$

For a floorplan with n modules, the greatest dimension of the BSG is $n \times n$. The rectangular spaces bounded by the horizontal and vertical segments are

defined as *rooms* as shown in Figure 2.3. In this representation, circuit modules will be placed in these rooms.

To realize a floorplan, a horizontal constraint graph $G_H(V_H, E_H)$ and a vertical constraint graph $G_V(V_V, E_V)$ are constructed. Except the source and the sink, V_H is the set of center points of $H_{i,j}$ while V_V is the set of center points of $V_{i,j}$. The edges of the graphs are the edges connecting the adjacent center points of the line segments. An example is shown in Figure 2.4. If an edge e passes a room, that has a module, its weight in G_H and G_V will be the width and the height of that module respectively. If the room is empty, the weight will be zero. Similar to sequence pair, the longest path finding algorithm is performed to compute the layout of the floorplan. The time complexity of this floorplan realization is $O(n^2)$ and the size of the solution space of the representation is $O(n^2!/(n^2-n)!)$. By storing the coordinates of each room that has a module, the memory usage of BSG is $\theta(2n(\lceil lgn \rceil))$ bits.

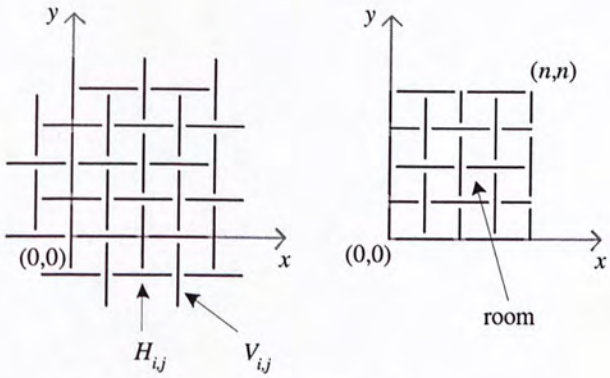


Figure 2.3: BSG structure and rooms.

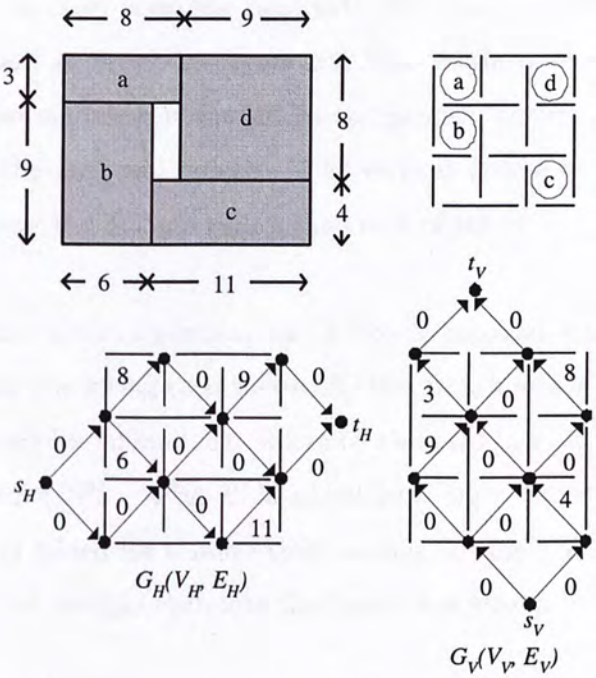


Figure 2.4: An example of floorplan realization using BSG.

2.2.3 O-tree

In 1999, Guo et al. proposed an *O-tree* representation for non-slicing floorplan in paper [22]. An O-tree is an ordered tree which can have an arbitrary number of branches (children) for each internal node. A horizontal and a vertical O-tree are needed to give an *admissible placement.* A placement is *admissible* if and only if the placement is compacted and no modules in the placement can move down or move left. The horizontal O-tree gives a placement, which is compacted in the x-direction, while the vertical O-tree gives a placement, which is compacted in the y-direction. Given the dimensions of the modules, a vertical O-tree can be constructed from a horizontal O-tree, which represents the placement, and vice verse. Therefore, either a horizontal O-tree or a vertical O-tree is stored in the actual implementation. Using the horizontal O-tree as an example, an O-tree T_o is a rooted directed tree. The root represents the left boundary of the chip. There is a directed edge connecting a parent

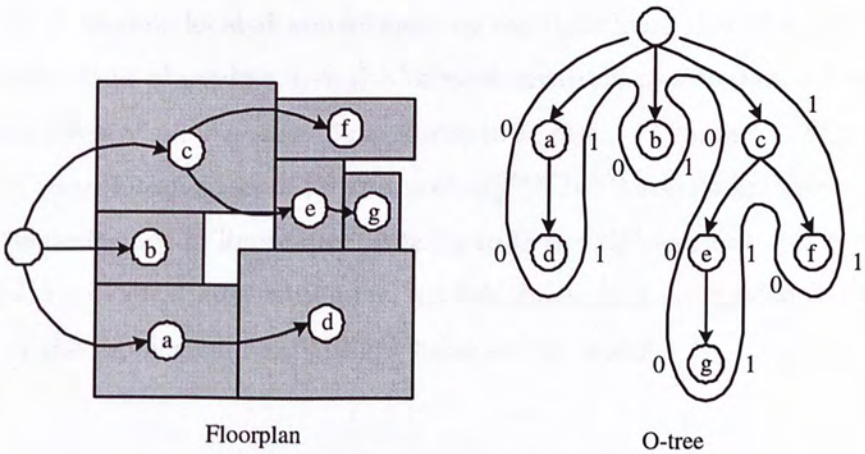
to its child if the child is on the right side of its parent with zero separation in the x-direction as shown in Figure 2.5. The weight of the edge is equal to the width of the module represented by the parent. The weights of the edges connecting to the root will be zero. The vertical O-tree can be constructed similarly by using the bottom edge as the root of the tree.

In the actual implementation, an O-tree is encoded into two sequences (T, π). T is a $2n$ bits string used to identify the branching structure of the tree where n is the number of modules. It can be obtained by visiting the tree using *depth first search* (DFS). A bit '0' is added for a move that descends an edge, while a bit '1' is added for a move that ascends an edge. The permutation π is the sequence of module visited in the depth first search.

The memory usage and the size of the solution space of O-tree are $\theta(n(2+\lceil lgn \rceil))$ bits and $O(n!2^{2n-2}/n^{1.5})$ respectively. Floorplan realization of O-tree using constraint graphs can be done in linear time. The drawback of O-tree is that it can only represent partial topological information. A particular O-tree can correspond to different placement solutions, for example, Figure 2.5 and 2.6 show two floorplans that correspond to the same O-tree. The dimensions of the modules are needed to give a unique placement solution.

2.2.4 B*-tree

Based on the O-tree representation, Chang et al. proposed the *B*-tree* representation in paper [14] in 2000. Similar to O-tree, a horizontal and a vertical B*-tree are needed to give an admissible placement. Using the horizontal B*-tree as an example, a B*-tree is an ordered binary tree whose root is the bottom-left module, which can be constructed in a recursive fashion according



$(T, \pi) = (00110100011011, \text{adbcegf})$

Figure 2.5: An example O-tree.

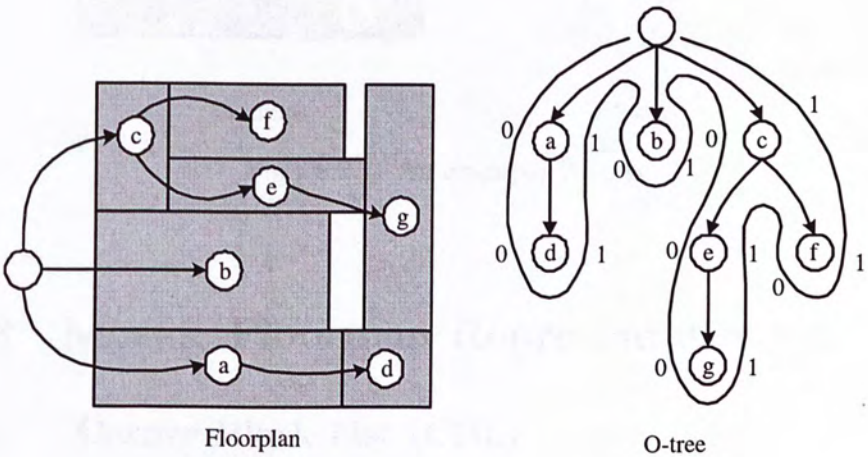


Figure 2.6: Another floorplan that corresponds to the same O-tree in Figure 2.5.

to the depth first search order. Starting from the root, the left sub-tree is constructed before the right sub-tree. The left child of module n_i is the lowest unvisited module located immediately on the right hand side of n_i. Similarly, the right child of module n_i is the leftmost unvisited module located immediately on top of n_i. An example is shown in Figure 2.7. Similar to O-tree, the size of the solution space of B*-tree is $O(n!2^{2n-2}/n^{1.5})$ and floorplan realization can be performed in linear time. Similar to O-tree, B*-tree can only represent partial topological information and module dimensions are needed to give the rest of the topological relationships between the modules.

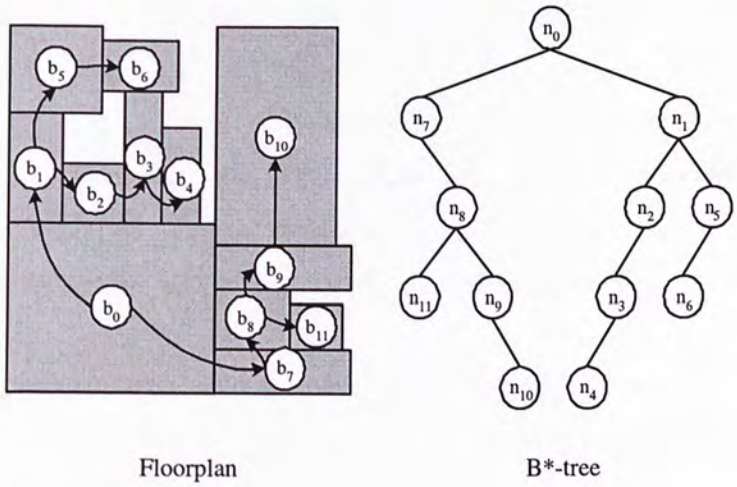


Figure 2.7: An example B*-tree.

2.3 Mosaic Floorplan Representations

2.3.1 Corner Block List (CBL)

In 2000, Hong et al. proposed the concept of mosaic floorplan and its first representation called *corner block list (CBL)* in paper [6]. In their paper, a *corner block* is defined as the upper rightmost block in a floorplan. A corner block

list consists of a 3-tuple (S, L, T) and is obtained by recursive *block deletion* of the floorplan.

A corner block is deleted in each block deletion step, the labelling of the deleted block is then added to S. L records the type of the corner block. There are two types of corner blocks and are defined by the T-junction at its bottom left corner. If the T-junction is a T rotated by 90° anticlockwisely ($\vdash$), the corner block is said to be vertically oriented and a bit '0' is recorded in L. If the T-junction is an inverted T ($\perp$), the corner block is said to be horizontally oriented and a bit '1' is recorded in L. The list T will store the number of T-junctions covered by the bottom edge or by the left edge of the corner block if it is vertically or horizontally oriented respectively. The number of 1's corresponds to the number of T-junctions covered. A bit '0' is added to T to delimit each block deletion record. The information of the last deleted block will not be added to L nor T. Finally, a CBL is obtained by writing these lists in the reserve order. An example is shown in Figure 2.8.

The memory usage of this 3-tuple representation is $\theta(n(3 + \lceil lgn \rceil))$ bits where n is the number of modules. The size of the solution space is $O(n!2^{3n})$ [15]. Besides, a CBL can be transformed to its corresponding floorplan in $O(n)$ time by using *block insertion*, which is a reversed operation of block deletion. Also, CBL can only be used to represent a mosaic floorplan which cannot have empty rooms as in non-slicing floorplan. As a result, paper [23] solves this problem by inserting a more than sufficient number ($O(n^2)$) of extra dummy blocks.

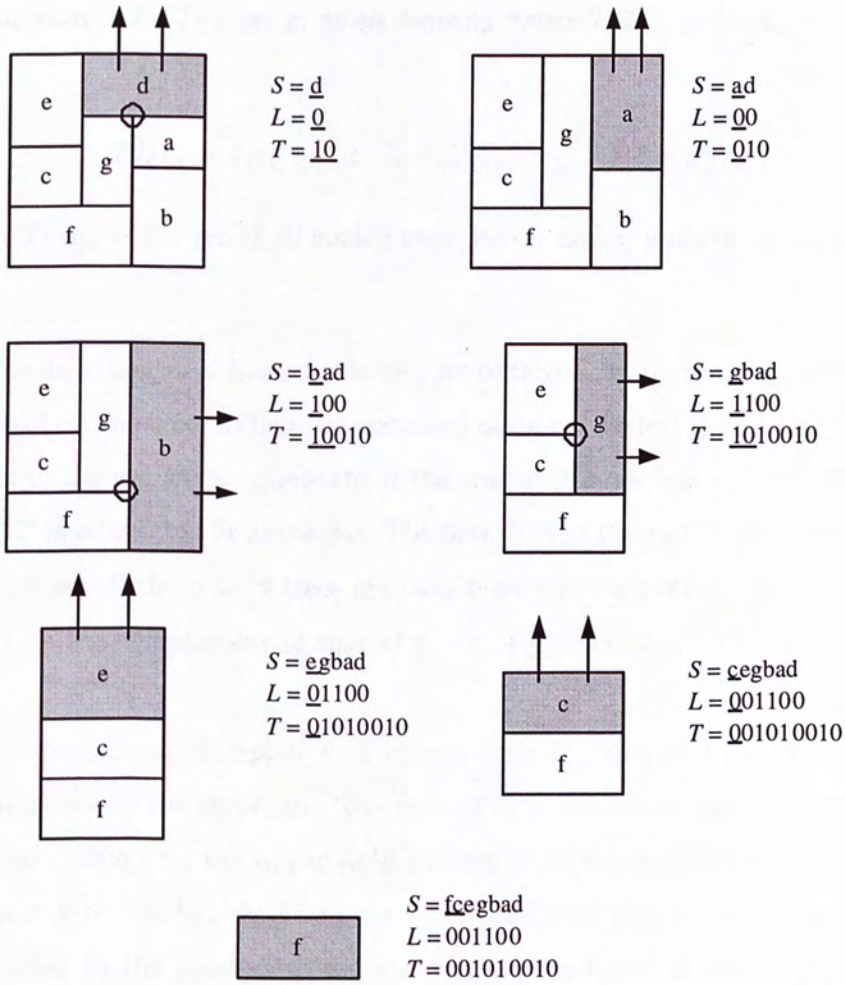


Figure 2.8: An example to compute CBL from a floorplan.

2.3.2 Twin Binary Trees (TBT)

The *twin binary trees (TBT)* floorplan representation was first proposed in the paper [7] in 2001. It shows an one-to-one mapping between TBT and mosaic floorplan. The definition of twin binary trees is as follows:

Definition 2.2 *The set of* ***twin binary trees*** $TBT_n \subseteq Tree_n \times Tree_n$ *is the set:*

$$TBT_n = \{(t_1, t_2) \mid t_1, t_2 \in Tree_n \text{ and } \theta(t_1) = \theta^c(t_2)\}$$

where $Tree_n$ *is the set of all binary trees with* n *nodes, and* $\theta(t)$ *is the labelling of* t.

The labelling of a binary tree can be obtained by performing an in-order traversal on the tree. When the traversed node has no left children, a bit '0' is added to the sequence. Similarly, if the traversed node has no right children, a bit '1' is added to the sequence. The first '0' and the last '1' in the labelling are omitted. If the pair of trees are twin binary to each other, the labelling of t_2 will be the complement of that of t_1, i.e., $\theta(t_1) = \theta^c(t_2)$.

Given a mosaic floorplan F, a pair of trees (t_1, t_2) can be constructed by travelling along the slicelines. The root of t_1 is the upper right corner of the chip. By connecting the upper right corners of all the modules, the tree edges connecting to the left child are the horizontal slicelines while the tree edges connecting to the right child are the vertical slicelines. Similarly for t_2, the root of t_2 is the lower left corner of the chip. By connecting the lower left corners of all the modules, the tree edges connecting to the right child are the horizontal slicelines while the tree edges connecting to the left child are the vertical slicelines. It has been shown that the pair of trees constructed in this way must be twin binary to each other. Besides, it is observed that the in-order traversal of the pair of trees are the same [15]. An example is shown

in Figure 2.9, $\theta(t_1) = 10010$ and $\theta(t_2) = 01101$, so $\theta(t_1) = \theta^c(t_2)$. In addition, their in-order traversals are both $ABCFDE$. The size of the solution space of TBT is $O(n!2^{3n}/n^{1.5})$.

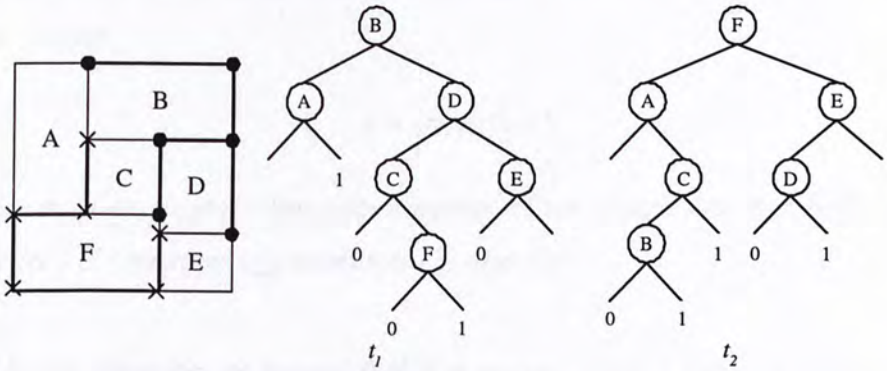


Figure 2.9: Construction of TBT from a floorplan.

2.3.3 Twin Binary Sequences (TBS)

Based on TBT, Young et al. proposed a representation called *twin binary sequences (TBS)* in paper [15] in 2002. It has been shown that the pair of trees (t_1, t_2) constructed from a mosaic floorplan must be twin binary to each other where $\theta(t_1) = \theta^c(t_2)$ and the in-order traversal of the pair of trees are the same. Based on these characteristics, a sequence π is used to represent the in-order traversal of the trees and a sequence α is used to represent the labelling $\theta(t_1)$ and $\theta^c(t_2)$. However, the in-order traversal and the labelling together cannot identify a pair of trees uniquely. Therefore, two more bit sequences are needed to identify them uniquely. These two sequences will record the structural information of the tree with a bit '0' representing the root of the tree and a node that is the right child of its parent and a bit '1' representing a node that is the left child of its parent. A sequence β is used to store these directional bit for t_1 following the order of π while another sequence β' is used to store the directional bits for t_2 similarly. Therefore, the twin binary sequences is a

4-tuple representation $s = (\pi, \alpha, \beta, \beta')$. The TBS for the example in Figure 2.9 is $s = (\pi = ABCFDE, \alpha = 10010, \beta = 000111, \beta' = 001001)$.

Definition 2.3 *A **twin binary sequence** s for a floorplan with n modules is a 4-tuple:*

$$s = (\pi, \alpha, \beta, \beta')$$

where π is an in-order traversal sequence of the n modules, and both (α, β) and (α^c, β') must satisfy condition (1) and (2):

In the following, we assume that $\alpha = \alpha_1\alpha_2\alpha_3 \ldots \alpha_{n-1}$ and $\beta = \beta_1\beta_2\beta_3 \ldots \beta_n$ ($\beta' = \beta'_1\beta'_2\beta'_3 \ldots \beta'_n$).

1. *In the bit sequence $\beta_1\alpha_1\beta_2\alpha_2 \ldots \beta_{n-1}\alpha_{n-1}\beta_n$ ($\beta'_1\alpha^c_1\beta'_2\alpha^c_2 \ldots \beta'_{n-1}\alpha^c_{n-1}\beta'_n$), the number of 0's is one more than the number of 1's.*

2. *For any prefix of the bit sequence $\beta_1\alpha_1\beta_2\alpha_2 \ldots \beta_{n-1}\alpha_{n-1}\beta_n$ ($\beta'_1\alpha^c_1\beta'_2\alpha^c_2 \ldots \beta'_{n-1}\alpha^c_{n-1}\beta'_n$), the number of 0's is more than or equal to the number of 1's.*

Floorplan realization of TBS is very efficient as stated in paper [15]. It can be done by scanning the sequences once from right to left. Given a TBS $s = (\pi, \alpha, \beta, \beta')$, we will start with an initial packing P with module π_n only. The algorithm then processes the modules from right (π_{n-1}) to left (π_1). Assume that we are processing module π_i now, if the corresponding α_i is equal to '0', it means that π_i is going to be inserted to the packing P from the top pushing $\pi_{i+1}, \pi_{i+2}, \ldots, \pi_k$ downward where k is the smallest number such that $i < k \leq n$ and $\beta_k = 1$. After adding module π_i to the packing P, $\beta_{i+1}, \beta_{i+2}, \ldots, \beta_k$ are deleted from β. The operation is similar when α_i is equal to '1'. For $\alpha_i = 1$, π_i is going to be inserted to the packing P from the left pushing $\pi_{i+1}, \pi_{i+2}, \ldots, \pi_k$ to the right where k is the smallest number

such that $i < k \leq n$ and $\beta'_k = 1$. After adding module π_i to the packing P, $\beta_{i+1}, \beta_{i+2}, \ldots, \beta_k$ are deleted from β'. An example is shown in Figure 2.10. The size of the solution space and the memory usage of this representation are $O(n!2^{3n}/n^{1.5})$ and $\theta(n(3+\lceil lgn \rceil)-1)$ bits. In addition, it is shown that every non-slicing floorplan can be generated uniquely and efficiently by inserting an exact number of empty rooms to a mosaic floorplan represented by a TBS. Besides, the lower bound and upper bound on the number of empty rooms are $n - 2\sqrt{n} + 1$ and $n - 1$ respectively.

2.4 Summary

In this chapter, we have reviewed several floorplan representations for slicing floorplan, non-slicing floorplan and mosaic floorplan. A table summarizing the characteristics of these representations is shown in Table 2.1. For slicing floorplan, the most popular representation is the normalized Polish expression (PE) [20]. This representation is simple to use and floorplan realization can be done in linear time. However, it can represent slicing floorplan only.

For non-slicing floorplan, a widely used representation is sequence pair (SP) [13]. It is also a simple representation which contains two sequences of module labellings. The disadvantage of SP is its large solution space. There exists a large amount of redundancy in this representation. Another non-slicing floorplan representation is the bounded-sliceline grid (BSG) representation [21]. Its solution space is also large and it contains a lot of redundancies. There are two tree representations for non-slicing floorplan, O-tree [22] and B*-tree [14]. These two representations have smaller solution space and less redundancies exists. Floorplan realization using O-tree and B*-tree can be done in linear time because its tree structure has already contained part of the topological

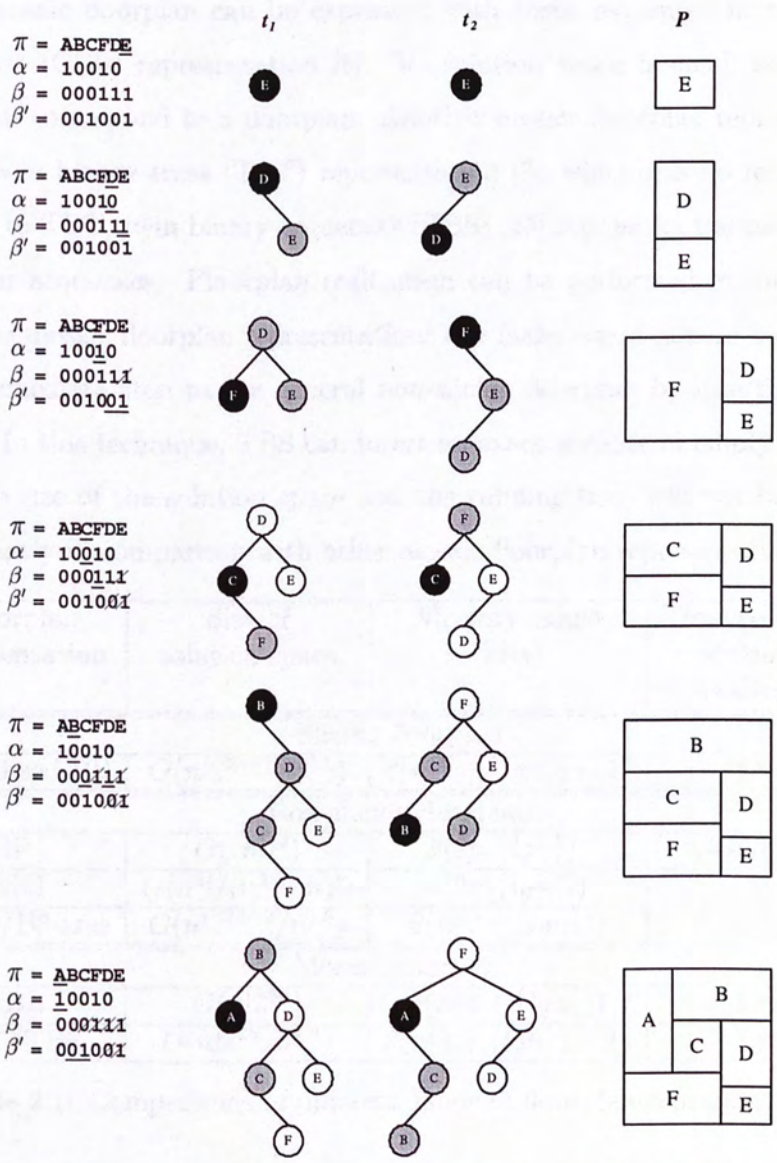


Figure 2.10: A floorplan realization example of TBS.

information. However, the dimensions of the modules are needed to be provided to represent the rest of the topological information.

A mosaic floorplan can be expressed with three sequences in the corner block list (CBL) representation [6]. Its solution space is small but not all CBL can correspond to a floorplan. Another mosaic floorplan representation is the twin binary trees (TBT) representation [7], which has no redundancy. Similar to TBT, twin binary sequences (TBS) [15] represents the pair of trees into four sequences. Floorplan realization can be performed in linear time. All these mosaic floorplan representations can make use of mosaic floorplan as an intermediate step to the general non-slicing floorplan by inserting empty rooms. In this technique, TBS can insert an exact number of empty rooms so that the size of the solution space and the running time will not be affected significantly in comparison with other mosaic floorplan representation.

Floorplan representation	Size of solution space	Memory usage (bits)	Time complexity of floorplan realization
Slicing floorplan			
normalized PE	$O(n!2^{5n-3}/n^{1.5})$	$\theta(n(1+\lceil lgn\rceil)-1)$	$O(n)$
Non-slicing floorplan			
SP	$O((n!)^2)$	$\theta(2n(\lceil lgn\rceil))$	$O(nloglogn)$
BSG	$O(n^2!/(n^2-n)!)$	$\theta(2n(\lceil lgn\rceil))$	$O(n^2)$
O-tree/B*-tree	$O(n!2^{2n-2}/n^{1.5})$	$\theta(n(2+\lceil lgn\rceil))$	$O(n)$
Mosaic floorplan			
CBL	$O(n!2^{3n})$	$\theta(n(3+\lceil lgn\rceil))$	$O(n)$
TBT/TBS	$O(n!2^{3n}/n^{1.5})$	$\theta(n(3+\lceil lgn\rceil)-1)$	$O(n)$

Table 2.1: Comparisons of different kinds of floorplan representations.

Chapter 3

Literature Review on Interconnect Optimization in Floorplanning

3.1 Wirelength Estimation

In a circuit, the wirelength is related to circuit delay and routing demand [31, 32]. Many floorplaners [20, 43, 15, 31, 32, 17, 18] consider wirelength reduction as their major objective. However, the correlation between the real circuit wirelength and the wirelength estimated in floorplanning depends on the accuracy of the wirelength estimation process. For a two-pin net, the wirelength is usually computed as the shortest Manhattan distance between the source and the sink. However, for a multi-pin net, the situation becomes complicated. It is difficult to obtain a balance between the estimation efficiency and the accuracy.

There are three main kinds of wirelength estimation methods as shown in Figure 3.1. Figure 3.1(a) shows a simple but rough estimation method, which is called the *half-perimeter bounding box approach.* It measures the wirelength as the half-perimeter of the smallest bounding box containing the set of pins.

The second approach is called the *minimum spanning tree (MST) approach* as shown in Figure 3.1(b). A MST is constructed to connect all the pins with the minimum length. Using the well-know Prim's algorithm [44], a MST can be computed in $O(klgk)$ where k is the number of pins. Figure 3.1(c) shows the third estimation approach. It is called the *rectilinear Steiner tree (RST) approach.* The connections are rectilinear and steiner points are added to remove the overlapping net segments. These overlapping net segments can increase the wirelength by 6% to 9% [45]. RST is the most accurate wirelength estimation method among these three approaches. However, its running time is much longer than those of the others.

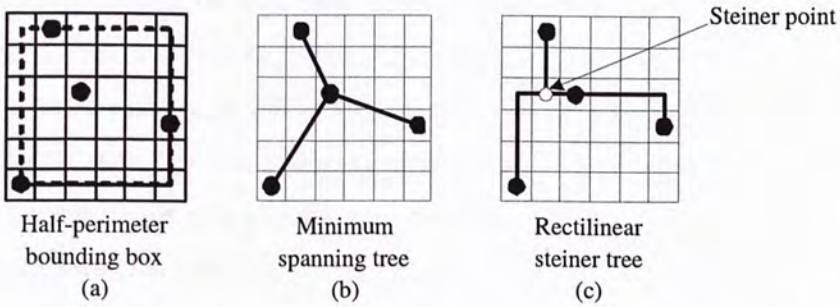


Figure 3.1: Different kinds of wirelength estimation methods.

3.2 Congestion Optimization

As the trend of the VLSI circuit design is going to have a huge number of interconnects, wiring congestion within a circuit has become a major concern in the VLSI physical design cycle. Congestion control in floorplanning is needed. Most congestion control approaches [30, 31, 32, 16, 33, 34, 35] are based on the estimations obtained by some global routing like operations. In these approaches, a floorplan is first divided into a two-dimensional grid structure as shown in Figure 3.2. Wires are routed in these grid units. There are

different kinds of routing models proposed. In Figure 3.3(a), an *L-shaped routing* is shown. A net is routed with exactly one bend in this simplest routing model. In Figure 3.3(b), a net is routed with two bends and it is known as *Z-shaped routing.* In general, a more realistic routing model should allow multi-bends in the route. This general routing model is called *multi-bend routing* as shown in Figure 3.3(c). Congestion overflow occurs in a grid when there are too many wires passed through the grid. It means that the interconnect demand has exceeded the supply of routing resources. Usually, the grid capacity (routing resources) can be computed according to the technology parameters, likes the number of metal layers, wire spacing and wire width as shown in Figure 3.4. Normally, a metal layer can only be used to place either horizontal or vertical wires. Different metal layers have different routing purposes with different technology parameters. For example, the upper metal layers are usually used for power or ground routing with thicker wires and larger wire spacing. The lower metal layers are usually used for local routing with thinner wires and smaller wire spacing. Using these parameters, we can compute how many routes can be placed inside a grid. After all the nets are routed using a suitable routing model, congestion will be evaluated in each grid as shown in Figure 3.5(a). We can estimate congestion based on two congestion estimation models. The first one is called the *track usage model.* In Figure 3.5(b), congestion is estimated as the horizontal and vertical track usage inside a grid. The second model is called the *grid-boundary crossing model* as shown in Figure 3.5(c). In this model, congestion is estimated as the number of routes crossing the four boundaries of a grid. Although these two estimation methods are different, their idea are the same. In the following sections, we will study in depth different literatures to address the congestion control problem using different routing models and different congestion estimation models.

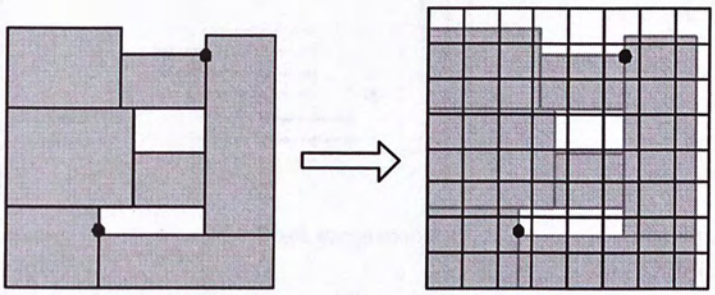


Figure 3.2: A two-dimensional grid structure constructed from a floorplan.

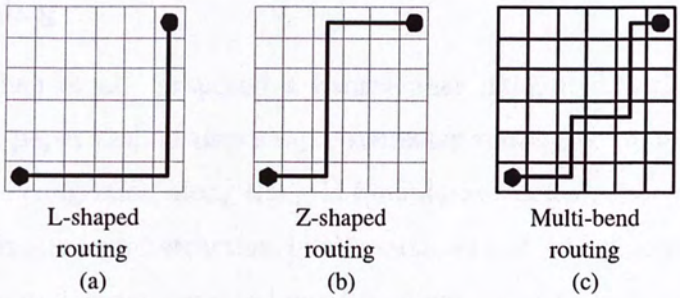


Figure 3.3: Different kinds of routing models.

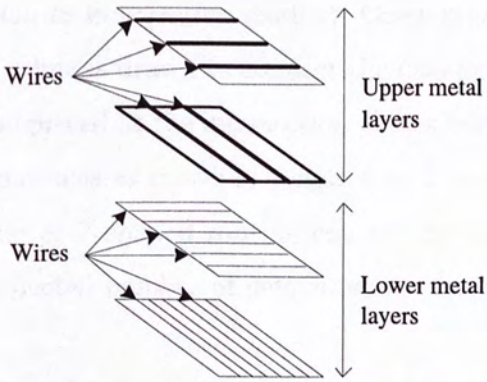


Figure 3.4: Metal layers for routing.

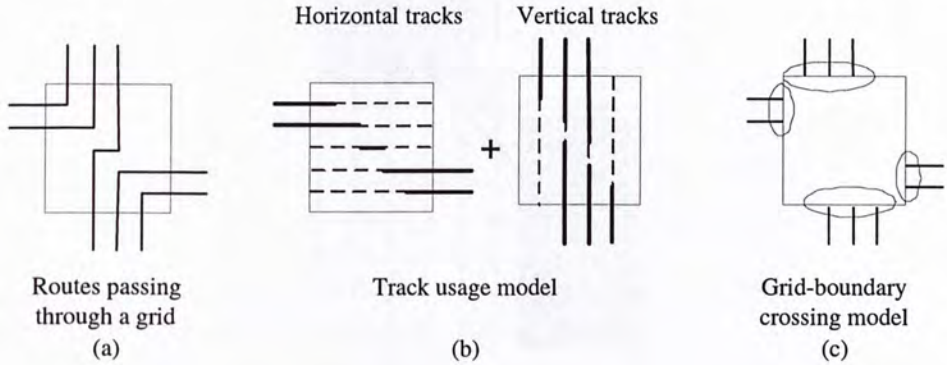


Figure 3.5: Different congestion estimation models.

3.2.1 Integrated Floorplanning and Interconnect Planning

In 1999, Chen et al. proposed a floorplanner integrated with interconnect planning in paper [30]. It uses simple geometry routing to route the nets and estimate the congestion along the grid boundaries. A floorplan is divided into a two-dimensional grid structure in the same way as global routing. Congestion is estimated as the expected number of nets crossing each grid boundary.

At the beginning, all multi-pin nets are decomposed into two-pin nets using the minimum spanning tree approach. Then, the pin positions are assigned using the *intersection-to-intersection* method. Given a two-pin net connecting modules A and B, a line is drawn to connect the centers of the modules. The two I/O pins will be placed at the intersection points between the line and the boundaries of the modules as shown in Figure 3.6. The nets are routed based on L-shaped routing or Z-shaped routing one by one. Finally, congestion is estimated as the expected number of nets crossing each grid boundary.

The floorplanner is based on a three-stage simulated annealing process. At high temperature, the annealing process behaves like a random walk and it

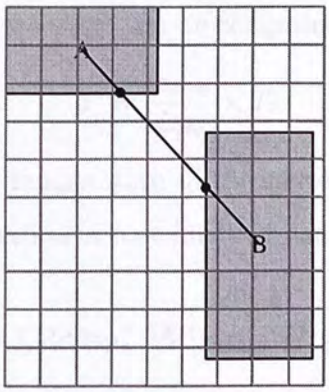


Figure 3.6: I/O pins assignment using intersection-to-intersection method.

is not sensitive to the cost function. It is because the large value of T has dominated the value of ΔC in the term $-\Delta C/T$, which determines whether a worsening move will be accepted. A rough wirelength estimation approach, half-perimeter bounding box, is used in this stage. In the second stage, the congestion cost is estimated using the L-shaped routing model. In addition, the wirelength is estimated by the minimum spanning tree approach. Finally, in the third stage, a more accurate routing model, Z-shaped model, is used for congestion estimation.

In their paper, the concept of temperature adjustment during cost function transition is explained. In order to cope with the discontinuity in switching the cost functions, the temperature is needed to be adjusted. In the transition between different stages, the difference in the order of magnitude of ΔC can be very large. It may lead to a sudden drop or rise in the acceptance rate P and the simulated annealing process may end prematurely. Therefore, the initial temperature T' of the next stage should be adjusted to maintain the original acceptant rate P. To achieve this, the original order of magnitude of ΔC_{old} using the old cost function is first computed, and the new order of magnitude of ΔC_{new} using the new cost function is then computed. The initial

temperature T' for the next stage can be computed as follows:

$$T' = \frac{\Delta_{new}}{\Delta_{old}} \times T_0$$

where T_0 is terminating temperature of the pervious stage. Using the new temperature T', the transition of cost function can be performed smoothly.

3.2.2 Multi-layer Global Wiring Planning (GWP)

Chang et al. proposed an interconnect-driven floorplanner with multi-layer global wire planning in paper [32] in 2000. They considered several interconnect optimization steps during floorplanning. They are interconnect topology optimization, layer assignment, buffer insertion, wiring sizing and spacing. The floorplanner is consisted of a four-stage simulated annealing process and an optional post-processing step.

Similar to paper [30], a rough cost function is used when temperature is high while a more accurate cost function is used when the temperature is low. In the first stage of the annealing process, the area, the longest wirelength and the total wirelength, which is estimated by the half-perimeter bounding box approach, are considered. The aim of considering the longest wirelength is to reduce the interconnect delay. It is because delay is directly proportional to the length of the critical path [28]. After the first stage, a good initial solution for interconnect optimization is obtained. In the second stage, delay is evaluated as the maximum net delay instead of the longest wirelength. The circuit delay is computed by a *global wire planning under area constraint (GWP-A)* method. This GWP-A method uses binary search and greedy wire packing to optimize layer assignment, wire sizing, wire spacing and buffer insertion. In the greedy wire packing, short wires are placed on lower metal layers while long wires are placed on upper metal layers in a greedy bottom-up manner [46]. Then, for each metal layer, the wire width and spacing are computed by

binary search. The binary search finds the optimal value for the wire width and spacing given the target delay. After obtaining the optimal wire width and spacing, the delay and the optimal number of buffers can be computed using the closed from expression in [47].

In the third stage, congestion is considered in the cost function. It can be computed by a *global wire planning under routibility constraint (GWP-R)* method. After defining the layer assignment and wire parameters in GWP-A, the congestion cost can be computed by using the Z-shaped routing model as in paper [30]. In the fourth stage, a more accurate model is used in GWP-R to evaluate congestion. It is the fast global routing using the *GA-tree algorithm* [48]. For each net, a routing graph is constructed by representing the nodes as the grids and the edges as the routes between adjacent grids. Larger weight is assigned to the edge crossing congested grid boundary. The route of a net can be found by computing a min-cost A-tree using the GA-tree algorithm. After all the nets are routed, congestion is estimated as the average number of nets passing through the grid boundaries.

3.2.3 Estimating Routing Congestion using Probabilistic Analysis

In 2001, Lou et al. proposed an accurate and detailed probabilistic analysis to estimate wiring congestion in paper [34]. The routing model used in the paper is the general multi-bend routing. It is a more realistic routing model resembling global routing. In their paper, a floorplan is first divided into a two-dimensional grid structure. Then, multi-pin nets will be decomposed into sets of two-pin nets. Probabilistic analysis of the horizontal and vertical track usage will be performed within the bounding box bounded by the source and

the sink of each net.

The congestion estimation model used in this paper is the same as in Figure 3.5(b). In a grid, congestion is estimated as the horizontal track usage and vertical track usage. Similar to paper [30], for a net covering a bounding box of dimension $(m \times n)$, the expected number of nets passing through the grid (i, j) is calculated as follows:

$$P_x(i,j) = \frac{F'_x(i,j)}{F(m,n)} \quad and \quad P_y(i,j) = \frac{F'_y(i,j)}{F(m,n)} \tag{3.1}$$

where $P_x(i,j)$ and $P_y(i,j)$ denote the probabilistic horizontal and vertical track usage in grid (i,j) respectively, while $F'_x(i,j)$ and $F'_y(i,j)$ are the number of possible routes using the horizontal and vertical tracks of grid (i,j) respectively. $F(m,n)$ is the total number of possible routes of the net where the bounding box containing the source and the sink has a dimension of $m \times n$.

The term $F(m,n)$ can be computed using dynamic programming as follows:

$$F(m,n) = \begin{cases} 1 & \textit{if } m = 1 \textit{ or } n = 1 \\ F(m-1,n) + F(m,n-1) & \textit{otherwise} \end{cases} \tag{3.2}$$

A look-up table is constructed to store each possible $F(i,j)$. It is because the computation of $F'(i,j)$ is also dependent on $F(i,j)$ as follows:

$$F'_x(i,j) = \begin{cases} F(m,n-1) & \textit{if } i{=}1 \textit{ or } j{=}1 \\ 1 & \textit{if } i{=}1 \textit{ and } j{=}n \\ \frac{F(m,n-j+1)+F(m,n-j)}{2} & \textit{if } i{=}1 \textit{ and } 1{<}j{<}n \\ F(m-i+1,n-1) & \textit{if } 1{<}i{<}m \textit{ and } j{=}1 \\ \frac{F(i,j)F(m-i+1,n-j)+F(i,j-1)F(m-i+1,n-j+1)}{2} & \textit{otherwise} \end{cases} \tag{3.3}$$

$$F'_y(i,j) = \begin{cases} F(m-1,n) & \textit{if } i{=}1 \textit{ or } j{=}1 \\ 1 & \textit{if } i{=}1 \textit{ and } j{=}n \\ F(m-1,n-j+1) & \textit{if } i{=}1 \textit{ and } 1{<}j{<}n \\ \frac{F(m-i+1,n)+F(m-i,n)}{2} & \textit{if } 1{<}i{<}m \textit{ and } j{=}1 \\ \frac{F(i,j)F(m-i,n-j+1)+F(i-1,j)F(m-i+1,n-j+1)}{2} & \textit{otherwise} \end{cases} \tag{3.4}$$

This probabilistic approach is quite efficient and accurate because the expected number of the track usage can be computed directly using the look-up table $F(i,j)$. Furthermore, the horizontal track and vertical track usage are estimated separately, this provides a more detailed and accurate evaluation for wiring congestion. Experimental results have shown that their estimation have high correlation with the post-route congestion measures.

3.2.4 Congestion Minimization During Placement

In 1999, Wang and Sarrafzadeh proposed in paper [31] a study of the correlation between wirelength and congestion. A consistent routing model is introduced. It is shown that wirelength minimization can reduce congestion globally. In addition, a new congestion minimization objective, called *overflow minimization with look-ahead*, is used in a post processing phase.

The congestion estimation model used in this paper is the grid boundary crossing model. Bounding box routing, which is similar to L-shape routing, is used. Congestion overflow can be computed as $d_e - s_e$ where d_e is the routing demand on grid boundary e, which can be computed as the number of nets passing through e. The term s_e is the routing supply. It can be computed as the maximum number of nets that are allowed to cross e according to the technology parameters. In their paper, the concept of consistency between the

routing model for wirelength estimation and the realistic routing model for congestion evaluation is proposed. Intuitively, the router used for wirelength calculation should be the same as that for congestion evaluation. However, experimental results show that consistency does not necessarily mean equivalent. Assume that the wirelength is computed in grid unit, a router used for computing wirelength and a realistic router for congestion estimation are said to be consistent if their computed values of wirelength are the same. It is because each unit of wirelength will contribute to one net crossing a grid boundary, which is similar to the definition of the routing demand d_e in congestion computation. As a result, if the routing model used for computing wirelength is consistent with the realistic router for computing congestion, we can reduce the wiring congestion by minimizing the wirelength only.

The congestion control is divided into two phases. In the first phase, the wirelength is minimized by the simulated annealing process. Experimental results show that global congestion is reduced after the first phase. A post processing is then done in the second phase to reduce local congestion. The objective function used in this post processing step is called overflow minimization with look-ahead. This post-processing step is a greedy algorithm and moves a cell or exchanges two cells if the move can reduce the direct overflow cost. The direct overflow cost of a move can be computed as $max\{d_e, s_e\} - max\{d'_e, s_e\}$, where d_e and d'_e are the routing demand of e before and after the move respectively. A larger value of the direct overflow cost implies a larger reduction on the overflow of the edge e after the move. Notice that the cost will be zero if $d_e < s_e$ and $d'_e < s_e$. In order to climb out of the local minima, the objective function is modified as $max\{d_e, s_e - \delta\} - min\{d'_e, s_e - \delta\}$, where δ is an adjustable parameter. The introduction of δ will reduce the chance of the cost to be zero even $d_e < s_e$ and $d'_e < s_e$. It leads to a modification on the solution even the routing demand does not exceed the routing supply.

3.2.5 Modelling and Minimization of Routing Congestion

Wang and Sarrafzadeh proposed paper [16], which is a follow up of [31], in 2000. In their paper, the authors show the correlation between the estimation obtained by a global router using the bounding box routing [31] and the realistic routing congestion measures. It is shown that the estimation of the bounding box approach proposed in [31] does not correlate well with the final congestion measures. Therefore, a realistic global router is used in their paper to estimate congestion. Furthermore, two post processing approaches are proposed to improve the performance on congestion control. They are the flow-based cell-centric algorithm and the net-centric algorithm.

Recall from [31] that reducing wirelength can reduce global congestion in the packing. Although congestion is reduced globally, congested spots will still occur in the highly connected clusters in the process of wirelength reduction. In order to balance all the wires to avoid these local congested spots, some post processing steps are needed. It is shown that the post processing step using the overflow minimization with look-ahead objective in [31] does not perform well. It is because it randomly moves the cells that lead to congestion reduction without locating the congested spots in the packing directly. As a result, two new algorithms are proposed. The first one is the flow-based cell-centric algorithm. It models cell movements as a transportation problem and places cells into grids by using the *max-flow min-cut algorithm* [49] as shown in Figure 3.7. In the flow network, the source s is connected to all cells c_i with weight 1 because one cell can only be placed in one grid. All the grids g_j will be connected to the sink t with weight s_j where s_j is the maximum number

of cells g_j can hold. The edges connecting the cell-grid pairs (c_i, g_j) represent the cell moves. There is a weight w_{ij} associates with each edge to denote the transportation (congestion) cost. By using the path augmentation method [50] of the network flow algorithm, cell assignments with the minimum congestion cost are found.

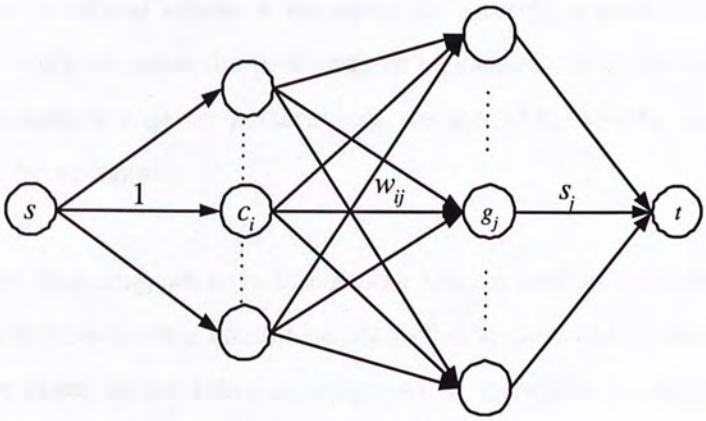


Figure 3.7: Flow-based cell-centric algorithm.

In the second approach, nets which lead to congested region are moved. Weight is assigned to each net to denote the congestion cost, which is equal to the number of overflowed grid boundaries it crosses. The nets are then sorted in a non-increasing order of these weights. By following this order, all the cells in the nets will be moved to reduce overflow. The operations will be repeated until there is no more reduction.

3.3 Buffer Planning

As we discussed before, interconnect optimization is a trend and a requirement in current VLSI physical design. In the previous section, we have reviewed on

some literatures to address the congestion problem. Unfortunately, the reduction of wiring congestion cannot reduce the delay of wires to achieve timing closure. To reduce the delay of the long critical wires, buffers are needed to be inserted along the wires. It has been studied that without buffer insertion, interconnect delay will increase quadratically with wirelength, but is linear with proper buffer insertions [51, 28]. However, buffers consume silicon spaces and cannot be placed wherever we want. As a result, a good buffer planning can significantly improve the performance of a circuit as shown in Figure 3.8. In order to achieve a better performance, we should take buffer insertions into account in floorplanning.

In buffer planning, we have to consider the separation x between adjacent buffers. This separation x should be controlled to be within a range $[low, up]$, that can be given by the users or computed by using the Elmore delay model [52]. Using this interval $[low, up]$, the regions for buffer insertions can be defined. These regions are called *feasible regions* [36, 37]. This concept is widely used in buffer planning. In this section, several literatures using different approaches to place the buffer positions are reviewed.

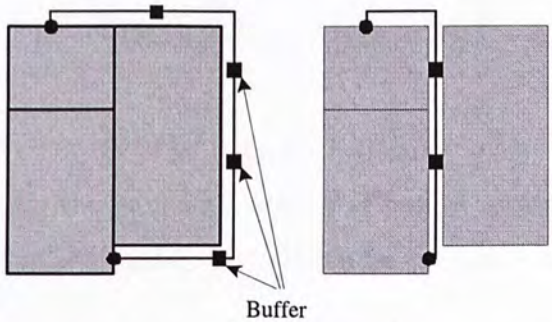


Figure 3.8: A better layout leads to a better routing with buffer insertions.

3.3.1 Buffer Clustering with Feasible Region

In 1999, Cong et al. first proposed the concept of *feasible region (FR)* for buffer insertion in paper [36]. Using the concept of feasible region, an effective *buffer block planning (BBP)* algorithm to perform buffer clustering is proposed.

A feasible region is defined as a region where a buffer of a wire can be placed in anywhere of it such that the delay constraint of the net can still be satisfied. The feasible regions for inserting k buffers by modelling the driver/buffer as a switch-level RC circuit [29] is shown in Figure 3.9. By using the Elmore delay model [52] with the key parameters for delay computation in Table 3.1, the feasible regions can be computed as follows:

For a long enough wire with length l, the minimum number of buffers n needed to be inserted to meet the delay constraint D_{req} is

$$n = \left\lceil \frac{K_2 - \sqrt{K_2^2 - 4K_1K_3}}{2K_1} \right\rceil \tag{3.5}$$

where

$$\begin{aligned} K_1 &= R_bC_b + D_b \\ K_2 &= D_{req} + \frac{R_o}{C_o}(C_b - C_l)^2 + \frac{C_o}{R_o}(R_b - R_d)^2 \\ &\quad -(R_oC_b + C_oR_b)l - D_b - R_dC_b - R_bC_l \\ K_3 &= \frac{1}{2}R_oC_ol^2 + (R_oC_l + C_oR_d)l - D_{req} \end{aligned}$$

After computing the minimum number of buffers required, n, the feasible region of the ith buffer $[x_{min}(i), x_{max}(i)]$ can be computed as:

$$\begin{aligned} x_{min}(i) &= \max\left\{0, \frac{K_5 - \sqrt{K_5^2 - 4K_4K_6}}{2K_4}\right\} \\ x_{max}(i) &= \min\left\{l, \frac{K_5 + \sqrt{K_5^2 - 4K_4K_6}}{2K_4}\right\} \end{aligned} \tag{3.6}$$

where

$$K_4 = \frac{(n+1)R_oC_o}{2i(n-i+1)}$$

$$K_5 = \frac{C_o(R_b - R_d)}{i} + \frac{R_o(C_l - C_b) + R_oC_ol}{n-i+1} +$$

$$\begin{aligned} K_6 = & \; nD_b - D_{req} + C_b\Big[R_d + (i-1)R_b + \frac{(n-i)R_ol}{n-i+1}\Big] \\ & + R_b[(k-1)C_b + C_l + C_ol] + \frac{R_oC_ol^2}{2(n-i+1)} + R_olC_l \\ & - \frac{(i-1)c(R_b - R_d)^2}{2iR_o} - \frac{(n-i)R_o(C_b - C_l)^2}{2(n-i+1)C_o} \end{aligned}$$

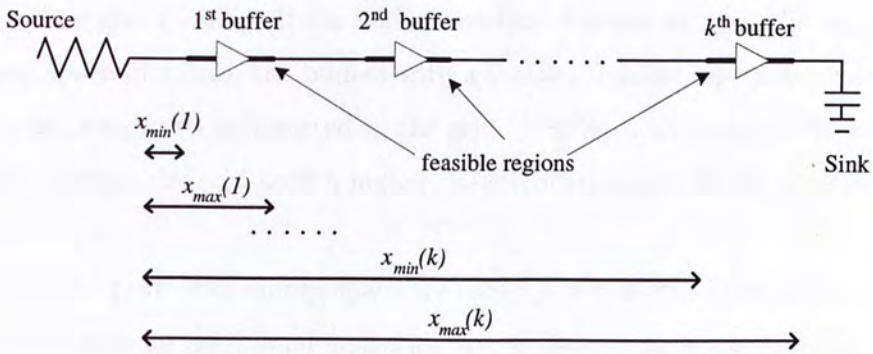


Figure 3.9: Feasible regions of k buffers.

Symbols	Parameters	Values
R_o	Unit wire resistance (Ω/mm)	0.075
C_o	Unit wire capacitance (fF/mm)	0.118
C_f	Unit wire fringing capacitance (fF/mm)	0.0641
D_b	Intrinsic repeater delay (ps)	36.4
C_b	Buffer capacitance (fF)	23.4
C_l	Load capacitance (fF)	23.4
R_b	Buffer resistance (Ω)	180
R_d	Driver resistance (Ω)	180

Table 3.1: Key parameters in the Elmore delay model using the $0.18\mu m$ technology [9].

After defining the feasible regions, the buffers of different nets are clustered into buffer blocks and placed in the empty spaces of the packing greedily. Given

a floorplan, the positions and dimensions of the routing channels are computed by building a horizontal and vertical polar graphs [53], which is similar to the floorplan realization using constraint graphs. Then, the channels are divided into grids as shown in Figure 3.10. Buffers will be inserted into these grids to form buffer blocks.

In the buffer block planning algorithm, a buffer will be inserted into the best grid for each net in each iteration. The algorithm will pick the grid with the largest amount of space for buffer insertions first, so that the overall floorplan area will not be increased. In each iteration, buffers are inserted into the picked grid greedily. If the buffer insertion demand exceeds the amount of empty space in a grid, the buffers with a smaller feasible region size will have a higher priority to be inserted in the grid. It is because buffers with a larger feasible region size will have a higher chance of successful buffer insertion.

If all the grids with empty space are used up, circuit modules will be shifted in order to make additional space for the buffers. The circuit module lying besides the channel with the largest buffer insertion demand will be picked to move first. It is because most of the unplaced buffers can be inserted into the grids of this new empty space without further expanding the chip area. If the picked grid is resulted from shifting a circuit module, only one buffer will be chosen to be inserted in that iteration. If there are more than one buffers which can be inserted into that grid, the one with the smallest feasible region size will be inserted. After inserting a buffer into a picked grid, the feasible regions of the unplaced buffers of the same net, the one which just has a buffer inserted, will be updated. The algorithm will terminate when all the buffers are placed.

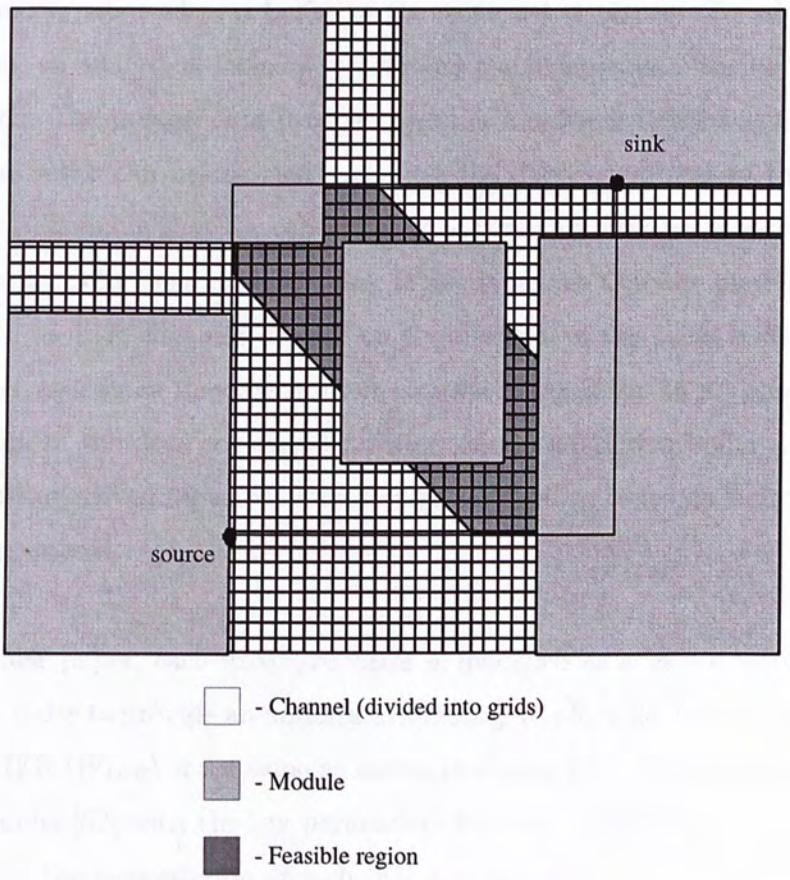


Figure 3.10: Buffer block planning.

3.3.2 Routability-driven Repeater Clustering Algorithm with Iterative Deletion

After the authors of paper [36] proposed the concept of feasible region, Sarkar et al. proposed the concept of *independent feasible region (IFR)* in paper [37]. Recalled from [36] that, the feasible regions of the unplaced buffers are needed to be recomputed when a buffer of the same net is placed. To address this problem, an analytical formula to compute the independent feasible region is proposed. The independent feasible region of a buffer is defined as the region where a buffer can be inserted such that the delay constraint of the wire is satisfied, assuming that the other buffers of the same wire are placed in whatever positions in their corresponding IFRs. It means that the placement of a buffer in its IFR does not depend on the position of the other buffers of the same net as long as they are in their corresponding IFRs. In addition, as the algorithm in [36] does not consider wiring congestion during buffer insertions, a congestion driven repeater block planning algorithm based on iterative deletion is proposed.

In their paper, each driver/repeater is modelled as a switch level RC circuit. In order to provide an uniform probability to place the buffers, the width of each IFR (W_{IFR}) is the same as shown in Figure 3.11. By using the Elmore delay model [52] with the key parameters for delay computation as shown in Table 3.1, the computation of each IFR is as follows:

For a long enough wire with length l which needs n buffers, the width of each IFR (W_{IFR}) to meet the target delay D_{tgt} with optimal delay D_{opt} is

$$W_{IFR} = 2\sqrt{\frac{D_{tgt} - D_{opt}}{R_o C_o (2n-1)}} \tag{3.7}$$

where

$$\begin{aligned} D_{opt} &= D(R_d, C_b, x_1) + D(R_b, C_l, l - x_n) \\ &\quad + \textstyle\sum_{i=2}^{n-1} D(R_b, C_b, x_i - x_{i-1}) + nD_b \end{aligned}$$

$$D(R, C, l) = \left(\frac{R_o C_o}{2}\right) l^2 + (RC_o + R_o C) l + RC$$

$$x_i = \frac{1}{n+1}\left(i \cdot l + \frac{(n-i+1)(R_b - R_d)}{R_o} + \frac{i(C_l - C_b}{C_o}\right)$$

After obtaining the width of each IFR (W_{IFR}), the independent feasible region of the ith buffer $[x_{min}(i), x_{max}(i)]$ can be computed as:

$$[x_{min}(i), x_{max}(i)] = \left[\left(x_i - \frac{W_{IFR}}{2}\right), \left(x_i + \frac{W_{IFR}}{2}\right)\right] \tag{3.8}$$

After computing the IFRs, the buffer locations can be planned using a congestion-driven iterative deletion [54] algorithm. A floorplan is first divided into a two-dimensional grid structure for routing. Then, for each IFR computed, it is further divided into finer two-dimensional grids, which are called *candidate repeater block (CRB)* locations, for buffer insertions as shown in Figure 3.12. The algorithm first computes the set of CRBs S_b to place the buffer b. If the whole IFR of buffer b is covered by a circuit block, the boundaries of the circuit block are used for buffer insertion. A bipartite graph $G(V, E)$ is constructed to represent the relationship of the buffers and the CRBs. The set of vertices V is defined as the set of buffers and the CRBs S_b. Each edge (b, c) denotes a placement of a buffer b to a CRB $c \in S_b$. In each iteration, the edge with the highest cost will be deleted where the cost is dependent on the congestion estimated by a Z-shaped simple global router as in [30] and the usage of the CRBs. After obtaining an assignment for each buffer b, the iterative deletion process is stopped and a congestion optimized buffer block planning is resulted.

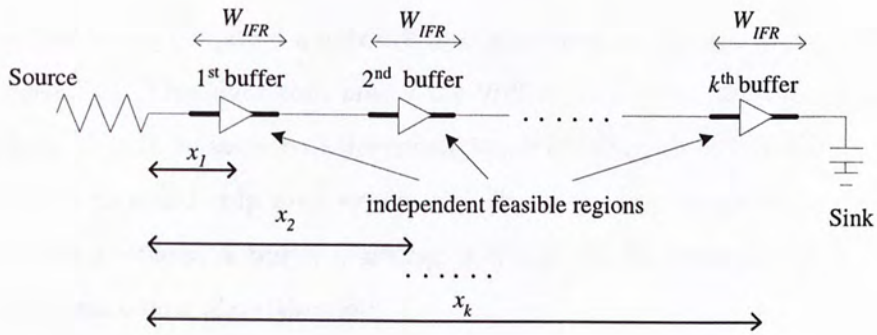


Figure 3.11: Independent feasible regions of k buffers.

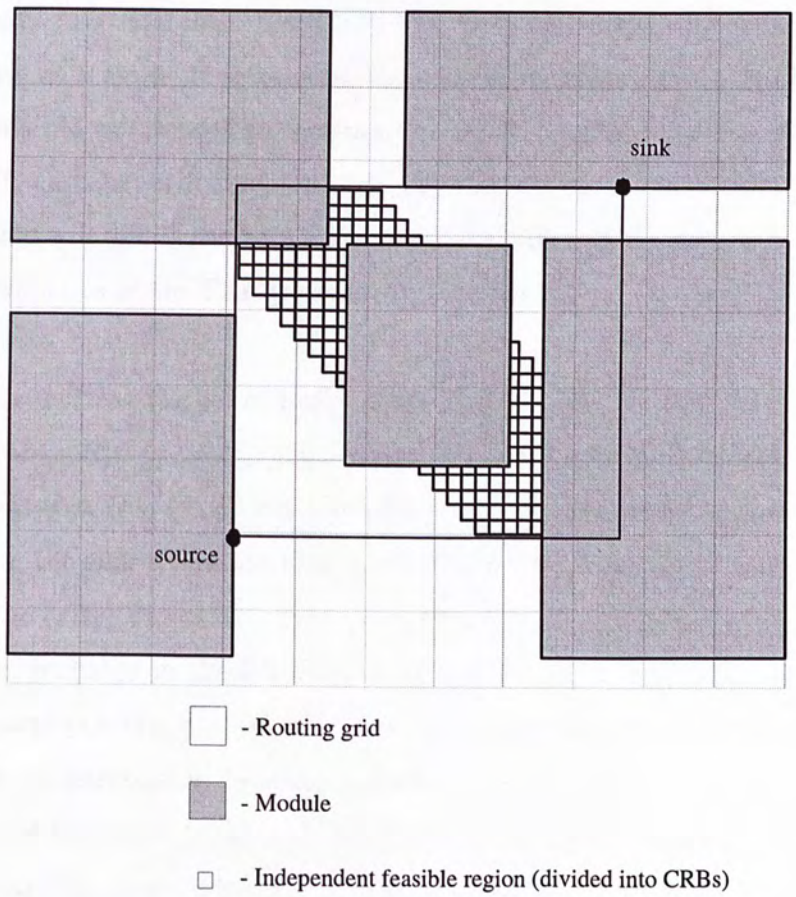


Figure 3.12: Repeater block planning.

3.3.3 Planning Buffer Locations by Network Flow

Tang and Wong proposed a network flow approach to plan the buffer locations in paper [38]. The algorithm places the buffers in the feasible regions defined in paper [36]. It makes use of the empty space between circuit blocks for buffer insertions to avoid chip area expansion. By modelling the problem as a network flow problem, a buffer planning solution can be obtained by using the min-cost max-flow algorithm [50].

In their paper, the empty space between circuit blocks are divided into different *buffer zones* Z_i for buffer insertions. Each buffer zone has a cost to specify its routability. Similar to [36], only one buffer can be inserted for each net at a time. It is because the feasible regions of the other buffers in the same net are needed to recompute once its previous buffer is placed. As a result, each net will contribute one feasible region in the layout only. The algorithm will divide the FRs into a set of disjoint *buffer rooms* r_i bounded by the boundaries of the FRs as shown in Figure 3.13.

After defining the set of buffer zones $Z = \{Z_1, Z_2, \ldots, Z_m\}$ and the set of buffer rooms $R = \{r_1, r_2, \ldots, r_w\}$ to place the set of buffers $B = \{b_1, b_2, \ldots, b_n\}$, a flow network $G = (V, E)$ can be constructed. The set of vertices V will be the source s, the sink t, and the elements in the set Z, R and B. The set of edges E will be (s, b_i) $\forall i = 1, 2, \ldots, n$, (b_i, r_j) $\forall i = 1, 2, \ldots, n$ and $\forall j = 1, 2, \ldots, w$ where r_j included in the FR of b_i, (r_j, r'_j) $\forall j = 1, 2, \ldots, w$ (note that each r_j has a corresponding r'_j), (r'_j, z_k) $\forall j = 1, 2, \ldots, w$ and $\forall k = 1, 2, \ldots, m$ where there is an intersection between r_j and z_k, and (z_k, t) $\forall k = 1, 2, \ldots, m$. The weights of the edges (s, b_i) and (b_i, r_j) are 1. As a buffer room r_j may belong to several FRs, so the edge (r_j, r'_j) with weight equal to the area of r_j is needed to bound the maximum number of buffers being inserted in r_j. As only the

intersecting area between the buffer zones and FRs can have buffer insertions, the weight of the edge (r'_j, z_k) is the size of the intersecting area between r_j and z_k. Finally, the weight of the edge (z_k, t) is equal to the area of z_k as the number of buffer insertions in a buffer zone is limited by the area of the zone. Recall that each buffer zone has a cost to specify its routability, so a min-cost max-flow algorithm can be applied to obtain a minimum cost buffer assignment.

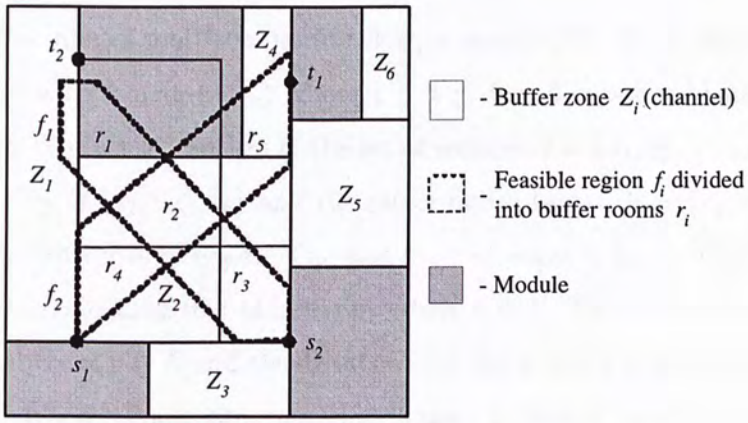


Figure 3.13: Buffer zones and buffer rooms.

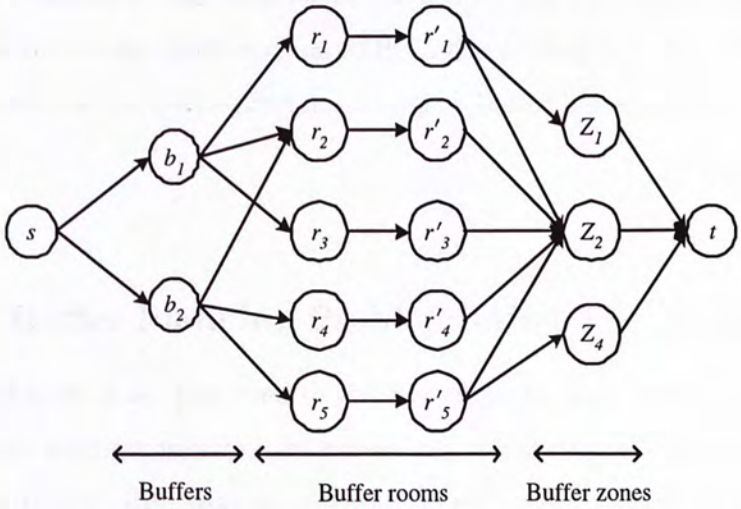


Figure 3.14: Flow network flow of the example in Figure 3.13.

3.3.4 Buffer Planning using Integer Multicommodity Flow

Dragon et al. proposed a buffer planning approach using integer multicommodity flow in paper [39]. The buffers are constrained to be separated by an interval x where $x \in [low, up]$. The terms low and up can be computed similarly as in paper [36, 37]. In their paper, the repeater parity constraint is considered. It will choose to use an inverter or a co-located pair of inverters (buffer) according to the signal parity condition.

For the integer multicommodity flow, a graph $G(V, E)$ is constructed for a problem with k nets (s_i, t_i) where $1 \leq i \leq k$ and n buffer blocks r_i where $1 \leq i \leq n$. The set V consists of the set of sources $S = \{s_1, s_2, \ldots, s_k\}$,the set of sinks $T = \{t_1, t_2, \ldots, t_k\}$ and the set of buffer blocks $R = \{r_1, r_2, \ldots, r_n\}$. There are two types of edges. The first type of edges is (u, u) which is a loop to denote a co-located pair of inverters where $u \in R$. The second type of edges is (u, v) where $u, v \in R$ and the distance between u and v is within the interval $[low, up]$. Integer linear programming is used to find a multicommodity flow such that the sum of flows of all the commodities is maximized. In the integer linear programming, the capacity of the source and the sink is equal to one while that of a buffer block is equal to the buffer's block capacity. This ensures that the number of paths passing through a buffer block will not exceed its capacity.

3.3.5 Buffer Planning Problem using Tile Graph

In 2001, Alpert et al. proposed to use tile graph to plan buffer locations and to estimate wiring congestion in paper [40]. In their method, a floorplan is divided into a two-dimensional grid structure and this structure is represented by a tile graph. A four-stage heuristic called *resource allocation for buffer and*

interconnect distribution (RABID) is then applied to optimize the routability of the floorplan.

Given a tile graph $G(V, E)$, the set of vertices V is the tiles and the set of edges E is $e(u, v)$ where u and v are neighboring tiles. Buffers are assumed to be inserted into the tiles and each tile $v \in V$ has a limited capacity $B(v)$. For wiring congestion, the grid boundary crossing model is used. In the tile graph, a grid boundary is represented by an edge $e(u, v)$. Therefore, there is a weight $W_{e(u,v)}$ assigned to each edge to specify the maximum number of wires that can cross the boundary between tile u and tile v. Using this tile graph, the amount of routing resources can be represented.

The RABID algorithm is divided into four stages, initial steiner tree construction, wire congestion reduction, buffer assignment and final post-processing. At the beginning, each net is routed as a rectilinear steiner tree. The wiring congestion will then be optimized in the second stage. In the second stage, all nets will be ripped-up and rerouted until the routing demand does not exceed the routing supply. It means that $w_{e(u,v)} \leq W_{e(u,v)}$ $\forall e(u, v) \in E$ where $w_{e(u,v)}$ is the number of wires crossing the boundary between tile u and tile v. The following cost function is used to evaluate congestion:

$$cost(e(u, v)) = \begin{cases} \dfrac{w_{e(u,v)} + 1}{W_{e(u,v)} - w_{e(u,v)} + 1} & if \ \dfrac{w_{e(u,v)}}{W_{e(u,v)}} < 1 \\ \infty & otherwise \end{cases} \tag{3.9}$$

When the number of wires crossing a boundary increases, the value of the term $W_{e(u,v)} - w_{e(u,v)}$ will decrease and the cost will increase. After planning the routes for all the nets, buffers are placed in the third stage. In the third stage, a net with longer delay will have a higher priority to have buffer insertions.

The buffers are placed such that the following cost function is minimized:

$$q(v) = \begin{cases} \dfrac{b(v) + p(v) + 1}{B(v) - b(v)} & \textit{if } \dfrac{b(v)}{B(v)} < 1 \\ \infty & \textit{otherwise} \end{cases} \tag{3.10}$$

where $b(v)$ is the number of buffers inserted into tile v and $p(v)$ is the expected number of nets passing through v. In the last stage, the unroutable wires will be ripped-up and rerouted.

3.3.6 Probabilistic Analysis for Buffer Block Planning

In 2002, Sham and Young proposed a probabilistic method to estimate congestion with buffer insertion taken into consideration in paper [17]. In their paper, adjacent buffers are constrained to be inserted at a flexible interval from each other as in [36, 37, 39]. A floorplan is divided into a two-dimensional grid structure. Probabilistic analysis on successful buffer insertion is performed in each grid. Congestion information is then computed according to the buffer insertion analysis.

In their paper, the constraint to place buffers are expressed as a *variable interval buffer insertion constraint* $[low, up]$, where low and up can be given by the users or computed analytically according to the Elmore delay model as follows:

$$\begin{cases} low = \sqrt{\dfrac{4 * (R_b * C_b + D_b)}{R_o * (C_o + C_f)}} \times \dfrac{1}{(2 * 10 * grid\_unit)} \\ up = \sqrt{\dfrac{4 * (R_b * C_b + D_b)}{R_o * (C_o + C_f)}} \times \dfrac{1}{(10 * grid\_unit)} \end{cases} \tag{3.11}$$

Using this constraint, the probability that a route l of wire k will have a buffer inserted at grid (x, y), notated by $b\_insert(x, y, l, k)$, can be computed as:

$$b\_insert(x, y, l, k) = \frac{N(dist)}{total} \tag{3.12}$$

where $N(dist)$ is the number of feasible ways of buffer insertions for l which will insert a buffer at the grid (x, y) that is at a shortest Manhattan distance $dist$ from the source of l, and $total$ is the total number of feasible ways of buffer insertions for l. An example to demonstrate the computation is shown in Figure 3.15. In actual implementation, dynamic programming is used to compute b_insert. Finally, the probability of successful buffer insertion at grid (x, y), which can hold $b\_space(x, y)$ buffers, notated by $b\_success(x, y)$, is computed as:

$$b\_success(x, y) = \min\left\{1, \frac{b\_space(x, y)}{\sum_{\forall k, \forall l} b\_insert(x, y, l, k)}\right\} \tag{3.13}$$

After computing the expected buffer usage ($b\_success(x, y)$) for each grid, the expected number of wires ($weight(x, y)$) passing through a grid (x, y) with respect to the buffer insertion constraint can be computed as:

$$weight(x, y) = \sum_{\forall k} \frac{\sum_{l \in L_k(x,y)} r\_success(l)}{\sum_{l \in L_k} r\_success(l)} \tag{3.14}$$

where L_k is the set of all routes for wire k, $L_k(x, y)$ is the set of all routes for wire k passing through the grid (x, y), and $r\_success(l)$ is the probability that the route l can be routed successfully from the source to the sink and satisfies the buffer insertion constraint. The computation of $r\_success(l)$ is based on the computed $b\_success(x, y)$ as shown in Figure 3.16. In practical implementation, the computation of $weight(x, y)$ is also performed by dynamic programming.

3.3.7 Fast Buffer Planning and Congestion Optimization

In 2003, Wong and Young proposed another approach to estimate congestion with buffer planning in paper [18]. The same buffer insertion constraint as

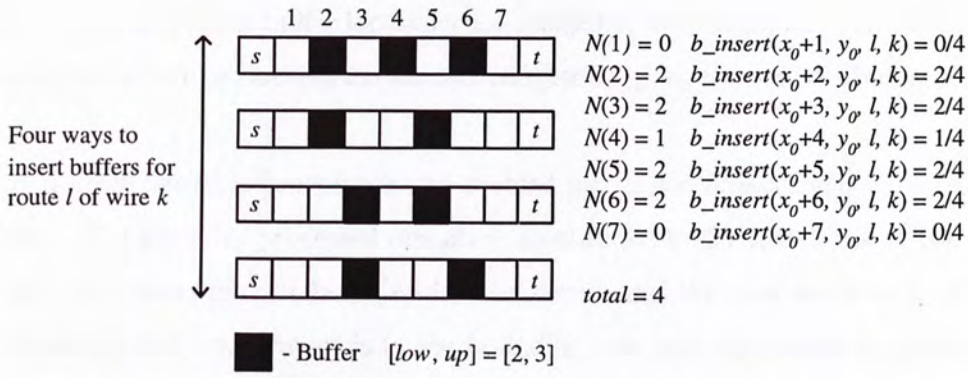


Figure 3.15: An example of computing $b\_insert(x, y, l, k)$ at each grid.

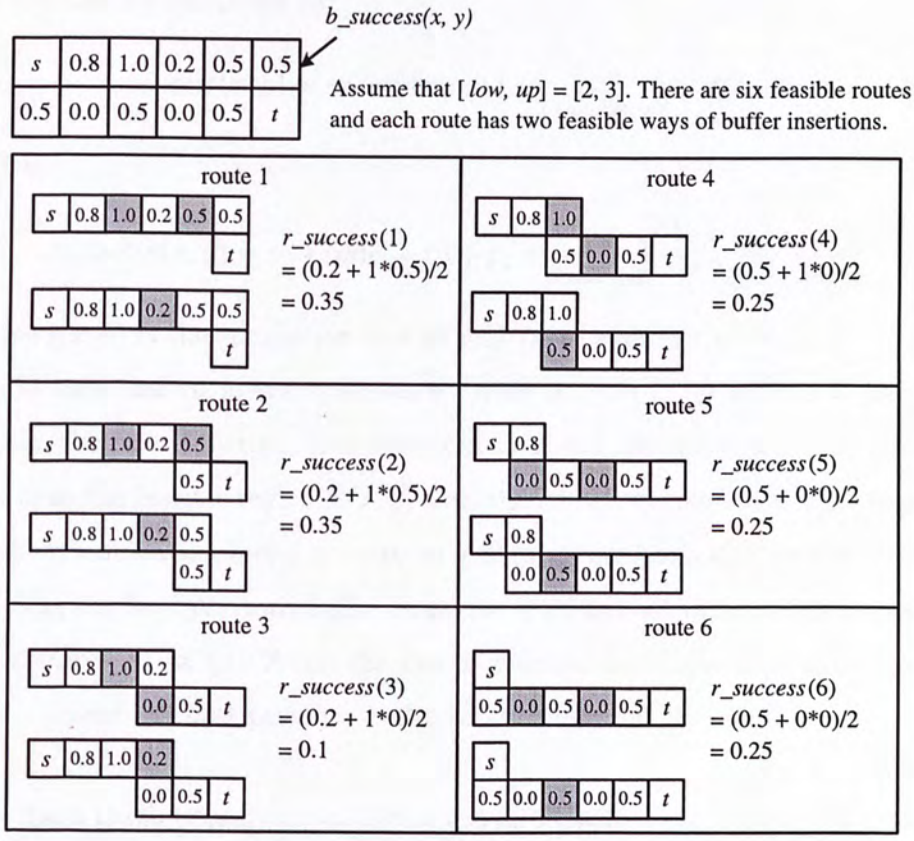


Figure 3.16: An example of computing $r\_success(l)$.

in [17] is used. Instead of using probabilistic approach to estimate the buffer locations, the exact buffer locations are computed in their paper. Probabilistic analysis is then performed to estimate congestion given the selected buffer plan.

In this paper, a floorplan is also divided into a two-dimensional grid structure. The nets are processed one after another for buffer insertions. Given a net, the bounding box bounded by the source and the sink are found. The algorithm will scan the grids in the bounding box from the source to the sink row by row. For each grid (x, y) which can be a feasible region for buffer insertion, the best previous buffer location is computed by dynamic programming assuming that a buffer is inserted at (x, y). A cost is assigned to each grid which can be computed as:

$$cost(x, y) = resource(x, y) + \min_{(a,b) \in R(x,y)} (cost(a, b)) \tag{3.15}$$

where

$$resource(x, y) = p_1 * cong(x, y) + p_2 * \frac{no.\ of\ buffers\ in\ (x,\ y)}{max.\ no.\ of\ buffers\ in\ (x,\ y)}$$

, $cong(x, y)$ is the congestion cost at grid (x, y) and $R(x, y)$ is the set of grids (a, b) such that (a, b) is at a distance d from the grid (x, y) where $d \in [low, up]$. From this cost function, it is observed that the algorithm will pick the grid (a, b) in the feasible region $R(x, y)$ with the minimum cost as the best previous buffer location of a buffer inserted at grid (x, y) as shown in Figure 3.17. After finding the best previous buffer location (a, b), its cost value will be computed and stored at (x, y). When the sink is reached, the algorithm can backtrack the sequence of best previous buffer locations for the net.

After buffer planning, congestion will be estimated using probabilistic method.

Firstly, each net will be broken into a set of sub-nets consisting of all source-buffer pair, buffer-buffer pairs and buffer-sink pair. Congestion will be computed for each subnet independently as follows:

$$cong(x, y) = \sum_{\forall subnet\ k} \frac{no.\ of\ possible\ routes\ for\ k\ passing\ (x,\ y)}{total\ no.\ of\ possible\ routes\ for\ k} \tag{3.16}$$

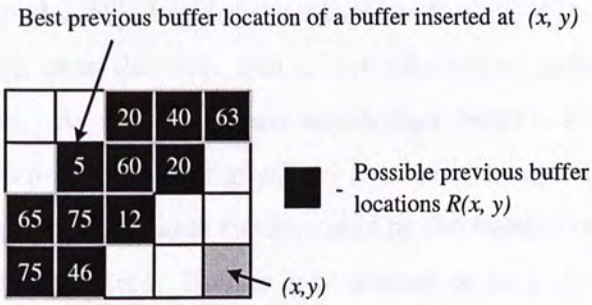


Figure 3.17: Computation of the best possible previous buffer location (a, b) of (x, y) in $R(x, y)$.

3.4 Summary

In this section, we review the previous works on congestion optimization and buffer planning. In congestion optimization, a floorplan is usually divided into a two-dimensional grid structure as in global routing. Simple global routing is performed to obtain the route for each net. As the number of wires that can pass through a grid is bounded, so over-congestion may occur when the routing demand of a grid exceeds the routing supply. In the literatures reviewed, most of them evaluate congestion as the expected number of wires passing through the grids or the expected number of wires passing across the grid boundaries. Different routing model can be used at different stages of the optimization. In an earlier stage, a rough but efficient routing model, for example, L-shaped or Z-shaped routing, can be used to obtain a good initial solution. When a later stage is reached, a more realistic routing model, for example, multi-bend

routing can be used.

For buffer planning, several previous works have been studied. The concept of buffer insertion inside feasible region is widely used. As the separation between adjacent buffers are constrained, the locations for buffer insertion are also constrained. These feasible regions can be computed according to the Elmore delay model [52]. Besides this insertion constraint, buffers should also be inserted such that the chip area is not affected as buffers also consume silicon resources. As a result, many approaches insert buffers in the empty space between circuit modules greedily. Similar to congestion optimization, some of the approaches divided the floorplan or the feasible regions into a two-dimensional grid structure. Buffers are assumed to be placed in the grids to form buffer blocks. Among the literatures reviewed, three major approaches can be concluded. One approach is to assign buffers into the grids by using graph algorithms, such as network flow, multicommodity flow and bipartite graph matching. Another approach is to use probabilistic methods to estimate the feasibility of buffer insertion in each grid. The last approach is to use dynamic programming to find the best possible buffer locations satisfying the buffer insertion constraint before evaluating the wiring congestion. Considering both wiring congestion and buffer insertion will be a major direction in many designing steps. Furthermore, efficiency will also be an important factor to be considered when the technology continues to scale down.

Chapter 4

Congestion Evaluation: Wire Density Model

The paper [55] on the content of this chapter has appeared in the proceedings of the *Design, Automation and Test in Europe(DATE)* 2003. [55]

4.1 Introduction

In the deep-submicron era, the complexities of VLSI circuits are growing rapidly. The interconnections between modules will become longer and denser in the future. Therefore, interconnect optimization in floorplan design has become ever more important than before. As floorplanning is at the beginning phase of the VLSI design cycle, an interconnect-optimized floorplan will favor the applicability and performance of the later stages like placement, global routing, detailed routing, etc, and, most importantly, allow timing closure to be achieved earlier.

Recently, some routability-driven floorplanners [30, 34, 35, 16, 17] are proposed, and most of them use the grid-based approach to measure the congestion of a floorplan. In this method, a floorplan is divided into grids as in global routing. At each grid, the expected number of nets passing through is

recorded as a weight to measure congestion. Although this approach is direct and simple, such kind of routing-oriented estimation is time consuming if it is performed in each iteration of the simulated annealing process. It is not practical for complex circuit designs. Therefore, a new and fast congestion evaluation model using a suitable floorplan representation will be very useful.

In order to provide a simple and efficient congestion evaluation model other than the complicated grid-based approach, an indirect congestion evaluation model, *wire density*, is proposed. Instead of estimating the congestion at each grid using global routing, we evaluate congestion as the wire density on the boundary of different regions in a floorplan. It is because a floorplan that has high wire density on average have a greater chance of having congestion problem. An example is shown in Figure 4.1. We use *twin binary trees (TBT)* as the floorplan representation because the regions to be evaluated can be naturally defined by the TBT representation. For a floorplan with n modules, $n-1$ regions are defined by each tree. In order to provide more regions for evaluation, we have constructed an additional pair of trees, which is the mirror of the original pair of trees. To increase the efficiency of our floorplanner, we have made use of a fast algorithm [41] for the *least common ancestor (LCA)* problem to compute the wire density. Experimental results have shown that an interconnect optimized floorplan of a complex circuit can be obtained in less than three minutes.

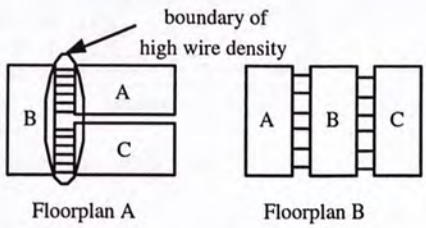


Figure 4.1: Floorplan A is more congested than floorplan B.

This chapter is divided into seven sections. Section 4.2 will give an overview of our floorplanner. In Section 4.3 and 4.4, the ideas and implementation details of the wire density congestion evaluation model will be described and explained. Finally, experimental results will be shown in Section 4.5.

4.2 Overview of Our Floorplanner

In this section, we will give a brief introduction to our routability-driven floorplanner with the new wire density congestion evaluation model. Our floorplanner is based on the TBT floorplan representation and simulated annealing process. Given a candidate floorplan solution, the total wirelength of the nets is estimated by the half-perimeter bounding box approach. The congestion cost is estimated by the wire density which is computed as the number of nets passing per unit length of the boundary of a region. These regions are defined by the TBT representation naturally and hierarchically. The estimation of wire density will start from the leaf nodes and follow the post-order traversal of the tree. Each tree can provide $n-1$ samples, i.e., $n-1$ regions, for wire density estimation. In order to obtain more samples, two additional trees are constructed from the original pair of TBT to provide a total of $4(n-1)$ wire density values.

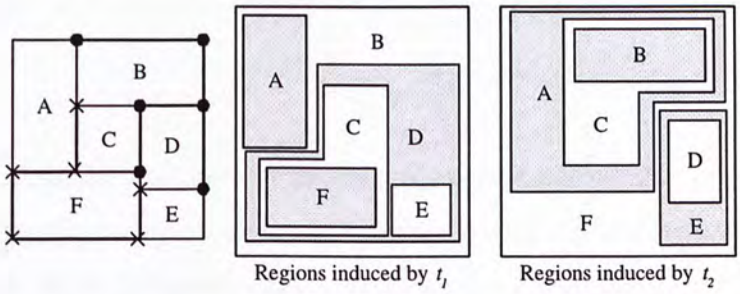


Figure 4.2: Regions induced by t_1 and t_2.

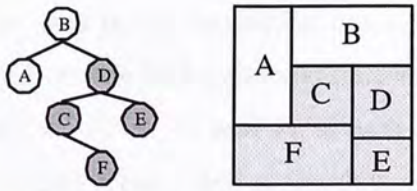


Figure 4.3: Formation of $R(D)$.

4.3 Wire Density Model

In order to improve the routability of a floorplan solution in an efficient way, an indirect but effective congestion evaluation model is used. This model aims at measuring congestion as the wire density (number of nets per unit length) on the boundary of different regions in the floorplan as shown in Figure 4.2.

Definition 4.1 *Given a TBT (t_1, t_2), the* ***region*** *$R(i)$ covered by module i in $t \in \{t_1, t_2\}$ is the rooms occupied by module i and the modules in the subtree rooted at i in t.*

As shown in Figure 4.3, the region $R(D)$ covered by module D in t_1 includes all the rooms occupied by module D and the modules C, F and E in its subtree. We can obtain $n-1$ wire density values for a tree with n nodes. It is because $R(root)$ is the whole packing and there will be no nets passing through the boundary of the packing. The following gives the equation of the *wire density* estimation:

$$WD_i = \frac{N_i}{P_i} \tag{4.1}$$

where WD_i is the wire density of $R(i)$, N_i is the total number of nets passing through the boundary of $R(i)$ and P_i is the normalized half-perimeter of $R(i)$. The details of the computation of N_i and P_i will be given in the coming sections.

We choose TBT as the floorplan representation in our floorplanner because it can define the regions for evaluation naturally. Also, a lot of fast and simple

tree algorithms can be used in our congestion evaluation. We start the estimation of wire density from the leaf nodes and follow the post-order traversal of the tree to compute the terms N_i and P_i at each node i. Using dynamic programming, the information computed at the child nodes is used to compute the wire density of the parent node.

4.3.1 Computation of N_i

The term N_i, which is the total number of nets passing through the boundary of $R(i)$, can be computed as follow:

$$N_i = N'_i + N_{l(i)} + N_{r(i)} - M'_i \tag{4.2}$$

where $l(i)$ is the left child of i, $r(i)$ is the right child of i, N'_i is the number of nets connected to module i, M'_i is the offset for the adjustments due to net merging and net completion, and it is computed as follow:

$$M'_i = \sum_{j=2}^{3}(j-1)m^i_j + \sum_{j=2}^{3} j \cdot c^i_j \tag{4.3}$$

where m^i_j and c^i_j are the number of nets merged and completed when j subnets of a single net meet at i. The value j is the number of subnets of a single net that meet at i. The value of j can be either two or three.

The adjustment for net merging m^i_j is needed because the repeated counting of an identical net in N'_i, $N_{l(i)}$ and $N_{r(i)}$ will over-estimate the number of nets. For j=2, two subnets coming from $R(l(i))$, $R(r(i))$ or module i of a single net are merged. For $j = 3$, three subnets coming from $R(l(i))$, $R(r(i))$ and module i of a single net are merged. The term m^i_j is multiplied by $j-1$ because we need to keep one counting in N_i rather than j counting. In Figure 4.4, we consider the situation when we reach module D during the post-order traversal. We use thick solid lines to represent merged nets. There is one net merged

between module D and $R(C)$, one between module D and $R(E)$, and one between $R(C)$ and $R(E)$, so $m_2^D = 3$. There is also one net merging between module D, $R(C)$ and $R(E)$, so $m_3^D = 1$.

Similarly, the adjustment for net completion c_j^i is needed because the repeated counting of an identical net in N_i', $N_{l(i)}$ and $N_{r(i)}$ will over-estimate the number of nets. The j in c_j^i has the same meaning as that in m_j^i. The term c_j^i is multiplied by j because the net has completed and all the counts should be eliminated. In Figure 4.4, we use thick dotted lines to represent completed nets. There is one net completed between module D and $R(C)$, two nets completed between module D and $R(E)$, three nets completed between $R(C)$ and $R(E)$, so $c_2^D = 1+2+3 = 6$. There is also one net completed between module D, $R(C)$ and $R(E)$, so $c_3^D = 1$. Finally, $M_D' = m_2^D + 2m_3^D + 2c_2^D + 3c_3^D = 3 + 2(1) + 2(6) + 3(1) = 20$

In Figure 4.4, the term N_D is computed as $N_D' + N_C + N_E$ - M_D' where $N_D' = 10$, $N_C = 13$, $N_E = 11$ and $M_D' = 20$. As a result, $N_D = 10 + 13 + 11$ - $20 = 14$. There are 14 nets passing through the boundary of $R(D)$. The value of N_i' can be obtained easily as the net specification is given in the floorplanning phase. However, the term M_i' will vary for different packings, a naive method to compute M_i' will make the time complexity of the congestion model becomes $O(mn)$ where n is the total number of nets and m is the total number of modules. It is not practical for complex circuits. Therefore, we have made use of an efficient algorithm for the least common ancestor (LCA) problem to compute M_i'. The details of the implementation will be given in Section 4.4.

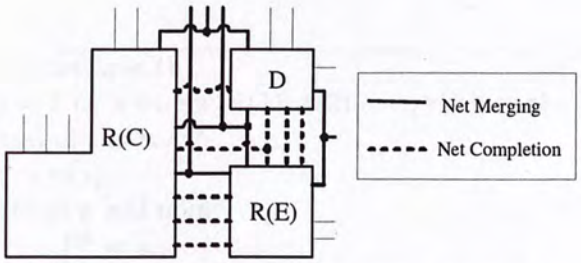


Figure 4.4: An example of computing N_D.

4.3.2 Computation of P_i

The term P_i, which is the normalized half-perimeter of $R(i)$, can also be computed easily by following the post-order traversal of a tree. As the tree edges of TBT represent the width and height of the rooms occupied by the modules, we will separate the half-perimeter P_i of region $R(i)$ into the horizontal (P_i^h) and vertical (P_i^v) portions to make the operation simple. The pseudo code is shown in Figure 4.5.

In the pseudo-code, w_i and h_i are the width and height of the room occupied by module i. The computation of $P(i)$ is divided into four cases. Line 3-6 show the case where module i is a leaf node as in Figure 4.6(a). Figure 4.6(b) shows the case on line 7-10 where module i has a left child $l(i)$ only. Figure 4.6(c) shows the case on line 11-14 where module i has a right child $r(i)$ only. Line 15-18 show the last case where module i has both left child $l(i)$ and right child $r(i)$ in Figure 4.6(d). Finally, on line 19, P_i^h and P_i^v are normalized by the chip width and height respectively to maintain a uniform order of magnitude. As dynamic programming is applied in the computation, the time complexity of *HalfPerimeter(t)* to compute the normalized half-perimeters of all the $(n-1)$ regions is only $O(n)$.

HalfPerimeter(tree t)

1. For $j = 1$ to n where $(\pi(1), \pi(2), \ldots, \pi(n))$ is the post-order traversal of the tree t
2. $\quad i = \pi(j)$
3. $\quad$ If i is a leaf node
4. $\quad\quad P_i^h = w_i$
5. $\quad\quad P_i^v = h_i$
6. $\quad$ Endif
7. $\quad$ If i has left child $l(i)$ only
8. $\quad\quad P_i^h = w_i + P_{l(i)}^h$
9. $\quad\quad P_i^v = max\{h_i, P_{l(i)}^v\}$
10. $\quad$ Endif
11. $\quad$ If i has right child $r(i)$ only
12. $\quad\quad P_i^h = max\{w_i, P_{r(i)}^h\}$
13. $\quad\quad P_i^v = h_i + P_{r(i)}^v$
14. $\quad$ Endif
15. $\quad$ If i has both left $l(i)$ and right $r(i)$ child
16. $\quad\quad P_i^h = max\{(w_i + P_{l(i)}^h), P_{r(i)}^h\}$
17. $\quad\quad P_i^v = max\{(h_i + P_{r(i)}^v), P_{l(i)}^v\}$
18. $\quad$ Endif
19. $\quad P_i = \frac{P_i^h}{chip\_width} + \frac{P_i^v}{chip\_height}$

Figure 4.5: Algorithm to find the normalized half perimeter.

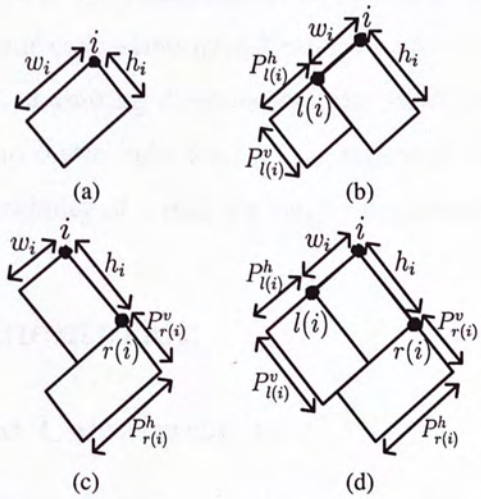


Figure 4.6: Cases in $P(i)$ computation.

4.3.3 Usage of Mirror TBT

After discussing the computation of N_i and P_i, we can evaluate the wire density for t_1 and t_2. By the characteristic of the TBT representation, the WD_i computed from t_1 represent the wire densities of the boundaries facing the upper right direction while those computed from t_2 represent the wire densities of the boundaries facing the lower left direction. Each tree can give $n-1$ statistical samples for the wire density evaluation where n is the number of modules. In order to increase the effectiveness of our congestion model, a pair of *mirror TBT*, which is based on the original pair of TBT, are constructed.

Definition 4.2 *Given a pair of TBT (t_1, t_2), its* ***mirror TBT*** *(t_3, t_4) is a pair of TBT such that:*

(1) *In t_3, $l(i) = j$ if $i = l(j)$ in t_2 and $r(i) = j$ if $i = l(j)$ in t_1.*

(2) *In t_4, $l(i) = j$ if $i = r(j)$ in t_2 and $r(i) = j$ if $i = r(j)$ in t_1.*

where $l(i)$ is the left child of i and $r(i)$ is the right child of i.

Actually, the mirror TBT can be imagined as the TBT constructed from a packing which is rotated 90° counterclockwise as shown in Figure 4.7. Together with mirror TBT, our congestion model can give $4(n-1)$ wire density values which consider in four routing directions (upper right for t_1, lower left for t_2, upper left for t_3 and lower right for t_4). As sufficient statistical samples are considered, the routability of a packing can be estimated accurately.

4.4 Implementation

4.4.1 Efficient Calculation of N_i

In this section, a detailed explanation of using the LCA algorithm to compute N_i will be given. Recall from Section 4.3.1 that the major difficulty of finding

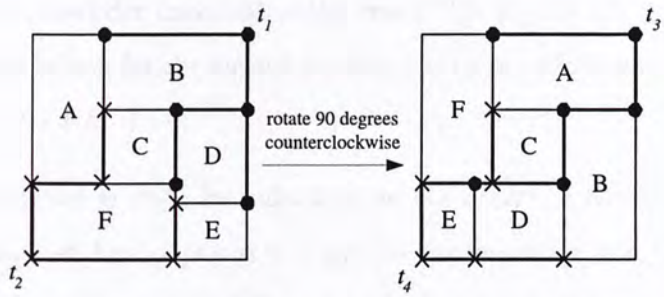


Figure 4.7: Construction of mirror TBT from TBT.

N_i is the high computational cost of finding the term M_i'. Instead of computing M_i' for each module i one by one, we are going to compute all M_i' incrementally by visiting each net one by one. An example is shown in Figure 4.8. In this example, we need to find the nodes B, C and D where adjustments are needed due to net merging and completion for net p. Net p will merge at node B, C and D and finally complete at B. The nodes, where adjustments are needed, are $LCA(u, v)$ where (u, v) are some module pairs in the net. For a net with k modules, $k-1$ LCAs should be found for adjustments. It is observed that we cannot get the correct LCAs where adjustments are needed by just picking the module pairs arbitrarily. For example, the LCAs obtained by simply selecting the three adjacent module pairs from the original net specification of p in Figure 4.8 are $LCA(A, C) = B$, $LCA(C, E) = D$ and $LCA(E, F) = D$ which are not the correct set of LCAs $\{B, C, D\}$ where adjustments are needed. Therefore, the following lemma is used to find the correct set of LCAs where adjustments are needed for a net.

Lemma 4.3 Given a tree t of n nodes (representing n modules) and a net p connecting k modules $(m_1, m_2, \ldots, m_k)$. The set of nodes L_p in t where two or more subnets of p meet (adjustment is needed) is

$$L_p = \{\bigcup_{i=1}^{k-1} LCA(m_{\pi(i)}, m_{\pi(i+1)})\}$$

where $(m_{\pi(1)}, m_{\pi(2)}, \ldots, m_{\pi(k)})$ is a permutation of the k modules obtained by

following the pre-order traversal of the tree t. (In Figure 4.8, $L_p = \{B, C, D\}$ and the permutation for the modules connected by net p following the pre-order traversal is $(ACFE)$.)

Proof: The proof is done by induction on the depth of the tree t. The pre-order traversal of t of depth $n+1$ can be expressed as AB_nC_n where A is the root, B_n and C_n represent the pre-order traversal of the left subtree of A rooted at B and the right subtree of A rooted at C with depth smaller than or equal to n respectively, and n is the larger value of the depths of the left and right subtree of A.

(1) When $n = 1$, the pre-order traversal of t is (AB_1C_1). as shown in Figure 4.9(a). There are three cases for t.

Case 1: If A has both left and right subtrees, i.e., B_1 and C_1 represent B and C respectively. The pre-order traversal is ABC. There are four cases for the net p.

(a) $p = \{C, B, A\}$. The subnet of net p will meet (twice) at node A. The permutated p is $\{A, B, C\}$, and the LCAs found according to the lemma are correct since $LCA(A, B) = A$ and $LCA(B, C) = A$.

(b) $p = \{B, A\}$. The subnet of net p will meet at node A. The permutated p is $\{A, B\}$, and the LCA found according to the lemma is correct since $LCA(A, B) = A$.

(c) $p = \{C, A\}$. The subnet of net p will meet at node A. The permutated p is $\{A, C\}$, and the LCA found according to the lemma is correct since $LCA(A, C) = A$.

(d) $p = \{C, B\}$. The subnet of net p will meet at node A. The permutated p is $\{B, C\}$, and the LCA found according to the lemma is correct since $LCA(B, C) = A$.

Case 2: If A has left subtree only, i.e., B_1 represents B and C_1 is an empty string. The pre-order traversal of t is AB. Similar to case 1(b),

the proposition is true.

Case 3: If A has right subtree only, i.e., B_1 is an empty string and C_1 represents C. The pre-order traversal of t is AC. Similar to case 1(c), the proposition is true.

Hence, the proposition is true when $n = 1$.

(2) Assume that the proposition is true when $n = k$, and the pre-order traversal of t is AB_kC_k as shown in Figure 4.9(b).

When $n = k + 1$, the pre-order traversal of t will be $AB_{k+1}C_{k+1}$. We can re-write it as $A(BD_kE_k)(CF_kG_k)$ as in the example of Figure 4.9(c). Let B^f and B^l be the first and last node of the permutated subnet of net p in BD_kE_k respectively, and C^f be the first node of the permutated subnet of net p in CF_kG_k. There are three cases for t:

Case 1: A has both left(BD_kE_k) and right(CF_kG_k) subtrees. There are five cases for the net p.

(a) p resides in the left(BD_kE_k) or right(CF_kG_k) subtree of A completely. According to the inductive hypothesis, the proposition is true.

(b) p resides in the left subtree of A and node A. According to the inductive hypothesis, the LCAs found from the left subtree(BD_kE_k) are correct. There is one more node that the subnet of net p will meet, which is A, and it will be found correctly according to the lemma since $LCA(A, B^f) = A$.

(c) p resides in the right subtree of A and node A. This case is proved similarly to case 1(b).

(d) p resides in the left and right subtrees of A but not node A. According to the inductive hypothesis, the LCAs found from the left and right subtrees of A are correct. There is one more node that the subnet of net p will meet, which is A, and it will be found correctly according to the lemma since $LCA(B^l, C^f) = A$.

(e) p resides in the left and right subtrees of A and node A. According to the inductive hypothesis, the LCAs found from the left and right subtrees of A are correct. There is one more node that the subnet of net p will meet(twice), which is A, and it will be found correctly according to the lemma since $LCA(A, B^f) = A$ and $LCA(B^l, C^f) = A$.

Case 2: A has left subtree only. This case is proved similarly to case 1(a) and case 1(b).

Case 3: A has right subtree only. This case is proved similarly to case 1(a) and case 1(c).

Hence, the proposition is true for $n = k + 1$.

By the principal of induction, the lemma is true. *Q.E.D.*

After obtaining the set L_p of a net p, we can update the value of the corresponding M'_{lca}. As shown in Figure 4.8, M'_B, M'_C and M'_D will be incremented by 1 because net p will be merged when they are visited. Finally, M'_i of the shallowest module i in the set L_p will be further incremented by 1 because the net is going to be completed there. In Figure 4.8, this shallowest module is B. The same operation will be performed for each net to find all M'_i. Finally, we can apply equation (4.2) to find all N_i values for wire density computation.

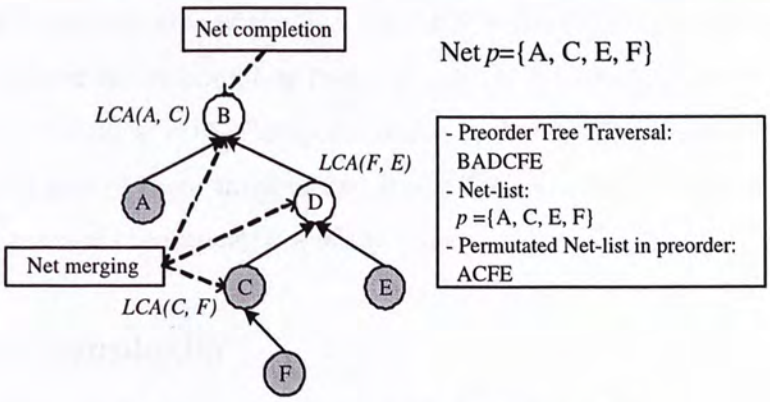


Figure 4.8: Using LCA to compute N_i.

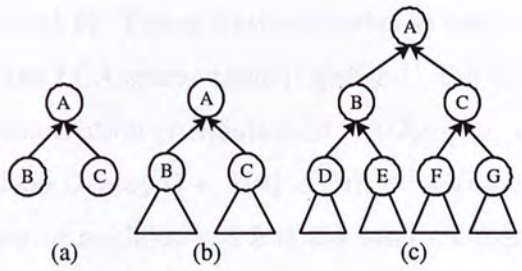


Figure 4.9: Proof of our M_i' computation.

4.4.2 Solving the LCA Problem Efficiently

In paper [41], an efficient and simple LCA algorithm is proposed. It reduces the LCA problem to the *Range Minimum Query (RMQ)* problem. By applying the *Sparse Table (ST)* algorithm for RMQ, the LCA problem can be solved in constant time with pre-processing time of $O(nlogn)$ using dynamic programming. The details of this algorithm is shown in Appendix A.

4.4.3 Cost Function

The cost function for the simulated annealing process of our floorplanner is shown as follow:

$$cost = A + \alpha(HP) + \beta(WD) \tag{4.4}$$

where A is the chip area of the floorplan, HP is the total wirelength estimated by the half-perimeter bounding box approach, WD is the summation of all the wire density values of the floorplan, and α and β are the weights to describe the importance of these three terms. In our floorplanner, α and β are set such that the ratio of the importance of the three terms is $A : HP : WD = 2 : 2 : 1$.

4.4.4 Complexity

Efficiency is one of the major advantages of our wire density congestion model. Recall from equation (4.1), the computation of the wire density WD_i is divided

into two parts, N_i and P_i. The operations needed to compute N_i for all i is the pre-processing of the LCA sparse table ($O(nlogn)$), the computation of all M_i' ($O(k)$) and the computation of equation (4.2) ($O(n)$), so the time complexity of finding N_i will be $O(nlogn) + O(k) + O(n) = O(nlogn) + O(k)$ where n is the total number of modules and k is the total number of pins in all nets. Usually, the magnitude of k is much greater than that of n, so we treat the time complexity of computing N_i as $O(k)$ here. Secondly, the time complexity of computing P_i for all i is $O(n)$. As a result, the time complexity of our congestion estimation method is $O(k)$ only.

4.5 Experimental Results

We have implemented two floorplanners for testing. One is a traditional floorplanner without considering congestion, the other one is a routability-driven floorplanner using our wire density model. Both floorplanners are based on the TBT floorplan representation and simulated annealing process. For the six MCNC benchmarks, only *ami33, ami49* and *playout* are used. We did not use *hp, xerox* and *apte* because it is expected that these simple circuits will not have routability problem. In addition, three more data sets (*n2000, n2500* and *n3000*) are generated to demonstrate the performance of our floorplanner for complex circuits. The detailed specifications of the data sets are shown in Table 4.1. The experiments are performed using a PC with a Pentium IV 1.4GHz processor and 512MB memory. We use a simple global router to evaluate the performance of the floorplanners.

Experimental results are shown in Table 4.2 and 4.3. The term *unroutable wire* are the wire that cannot be routed in the shortest Manhattan distance due to congestion and the wire that cannot have successful buffer insertions with respect to the variable buffer insertion constraint $[low, up]$. We use the

data in paper [9] to compute the parameters of the router. We use the feature values of the $0.18\mu m$ technology for all data sets. In this wire density model, we count the average number of nets crossing a unit grid boundary without performing global routing. The accuracy of this method has been shown in the experimental result. It shows a significant reduction in the number of unroutable wires when comparing with the traditional floorplanner which consider area and wirelength only. The number of unroutable wires have been reduced by about 22% on average. For the efficiency, as we have made use of the tree structure and the fast LCA algorithm, the computation of all the wire density values can be done in linear time. From the experiment, the runtime of our floorplanner for a complex circuit with three thousand nets ($n3000$) is less than three minutes. Therefore, our boundary crossing model is a good evaluation model in terms of accuracy and efficiency.

4.6 Conclusion

In this chapter, we present a new congestion evaluation model using wire density as a measurement. We use TBT as the floorplan representation because the regions for evaluation can be defined by the TBT representation naturally and the fast and simple tree algorithms, for example, the LCA algorithm, can facilitate the efficiency of our congestion model. By using the regions defined by the TBT and the mirror TBT, sufficient samples can be taken for congestion evaluation. The time complexity of the whole congestion estimation method is linear with respect to the number of two-pin nets. Experiments have shown that this congestion evaluation model is efficient and effective when dealing with complex circuit designs. The number of unroutable wires can be greatly reduced in a short time.

Data	Number of Modules	Number of IO Pins	Number of Nets	Number of decomposed two-pin nets
ami33	33	42	123	304
ami49	49	22	408	535
playout	62	192	1611	2122
n2000	60	200	2000	2782
n2500	75	200	2500	3450
n3000	90	200	3000	4139

Table 4.1: Specifications of the data sets.

Floorplanner	Deadspace (%)	Wire Length ($10^3 \mu m$)	Number of Unroutable Wires	Runtime (s)	Time per Iterations (ms)	Iterations
ami33						
Traditional	10.31	21.25	15.03	21.05	0.16	129702
Ours	11.72	21.79	11.44	42.54	0.30	140502
ami49						
Traditional	10.87	386.45	14.53	25.54	0.20	129702
Ours	17.71	402.87	9.96	75.65	0.50	152602
playout						
Traditional	10.25	284.78	170.54	30.64	0.24	129702
Ours	16.13	302.88	134.19	86.22	0.62	139902

Table 4.2: Experimental results of our wire density model on MCNC Benchmark.

Floorplanner	Deadspace (%)	Wire Length ($10^3 \mu m$)	Number of Unroutable Wires	Runtime (s)	Time per Iterations (ms)	Iterations
n2000						
Traditional	11.56	102.60	581.75	35.41	0.27	129702
Ours	15.69	113.28	456.87	95.97	0.76	125594
n2500						
Traditional	14.08	141.09	887.35	37.54	0.29	129702
Ours	17.78	158.44	737.72	121.25	0.95	127020
n3000						
Traditional	16.37	177.77	1299.75	46.95	0.36	129702
Ours	20.16	205.43	1068.89	151.39	1.19	127502

Table 4.3: Experimental results of our wire density model on complex circuit.

Chapter 5

Buffer Planning: Simple Buffer Planning Method

5.1 Introduction

Buffer insertion is usually performed after routing in the VLSI physical design cycle. In deep submicron VLSI designs, it has been shown that the interconnect delay for a wire without buffer insertion will increase quadratically as the wirelength increases, while it will only increase linearly with proper buffer insertions [26, 27, 28]. Also, it is shown that the delay of a 2cm global interconnect can be reduced by a factor of 7× by optimal buffer insertion [29]. Therefore, buffer insertion is very important in improving the circuit performance. However, buffers also consume silicon resources. If buffers are not carefully planned at the beginning of the design cycle, buffer insertions will be difficult due to the insufficiency in space in the final circuit layout. To address this problem, we consider buffer planning earlier in the floorplanning phase. Once buffers are considered in floorplanning, there is a greater chance to have successful buffer insertions after routing.

Some previous buffer planning approaches [36, 37, 38, 39, 40, 17, 18] aim

at computing the best possible buffer locations. However, this kind of computations are time consuming. When efficiency is a major concern, it is not practical, especially for complex circuit designs, to have this kind of detailed buffer planning in the early designing stage. A new buffer planning method that can simplify this complicated process will be very useful. We believe that an approximate buffer planning in the floorplanning phase is already good enough to improve the flexibility of buffer insertion. Also, an approximate planning can lead to a simpler implementation and thus a better performance of the floorplanner.

Our simple buffer planning method aims at improving the feasibility of buffer insertions of the output floorplan solution. The number of nets blocked due to unsuccessful buffer insertion is counted. We assume that buffers are constrained to be inserted for long enough wires such that the distance between adjacent buffers should be within a range of $[low, up]$. This constraint is called the *variable interval buffer insertion constraint* [17, 18]. In our method, we are going to determine whether a net can have all its buffers inserted successfully. In order to achieve an efficient evaluation, we use dynamic programming and a table look-up approach to obtain these routability information in constant time with a linear pre-processing time. In this approach, a floorplan is divided into a two-dimensional grid structure. *Feasible grid* is defined as a grid that contains pin and/or has sufficient space for buffer insertions. Two look-up tables are constructed using dynamic programming to store the routability information between these feasible grids. By accessing the look-up tables, we can decide immediately whether a net is blocked due to unsuccessful buffer insertion.

This chapter is divided into six sections. In Section 5.2, a brief review of the variable interval buffer insertion constraint will be given. Section 5.3 will

give an overview of our floorplanner. In Section 5.4 and 5.5, the ideas and implementation will be described and explained in details. Finally, experimental results and a conclusion will be given in Section 5.6 and 5.7 respectively.

5.2 Variable Interval Buffer Insertion Constraint

In some interconnect-driven floorplanners [17, 18], the variable interval buffer insertion constraint is used to constrain the buffer locations. In this constraint, an interval $[low, up]$ is specified by the user or computed based on the *Elmore delay model* [52]. This interval will be used to constrain the separations between adjacent buffers of a net. Based on the feature values of the $0.18\mu m$ technology as shown in Table 5.1, we can compute this interval using equation 5.1. According to this constraint, the distance between the source of a net and the first buffer, between any two consecutive buffers, and between the last buffer and the sink of a net is required to lie between *low* and *up* inclusively. In this chapter, we simplify the name of the variable interval buffer insertion constraint as *interval constraint.*

$$\begin{cases} low = \sqrt{\frac{4*(R_b*C_b+D_b)}{R_o*(C_o+C_f)}} \times \frac{1}{(2*10*grid\_unit)} \\ up = \sqrt{\frac{4*(R_b*C_b+D_b)}{R_o*(C_o+C_f)}} \times \frac{1}{(10*grid\_unit)} \end{cases} \tag{5.1}$$

Parameters	Values
R_o - unit wire resistance (Ω/mm)	0.075
C_o - unit wire capacitance (fF/mm)	0.118
C_f - unit wire fringing capacitance (fF/mm)	0.0641
D_b - intrinsic repeater delay (ps)	36.4
C_b - load capacitance/buffer capacitance (fF)	23.4
R_b - driver resistance/buffer resistance (Ω)	180

Table 5.1: Feature values in the $0.18\mu m$ technology [9].

Definition 5.1 *A net or a subnet is said to be* ***blocked*** *if none of its shortest Manhattan distance routes can have all its required buffers inserted to satisfy the interval constraint. Otherwise, it is said to be* ***routable****.*

5.3 Overview of Our Floorplanner

In this section, we will give a brief overview of our floorplanner using the simple buffer planning method and the wire density evaluation model. Given the information of the modules and the netlists, a two-stage simulated annealing approach is used to obtain an interconnect-driven floorplan. In the first phase of the annealing process, we consider area, total wirelength (estimated by the half-perimeter bounding box approach) and wire density (as described in Chapter 4) in the cost function. When the temperature is cooled down and the annealing process enters its second stage, we will use the minimum spanning tree (MST) approach to estimate the total wirelength instead of using the half-perimeter bounding box approach. Furthermore, we will consider one more factor, the buffer insertion feasibility, in the cost function. The feasibility of buffer insertion is evaluated by our simple buffer planning method. A floorplan is divided into a two-dimensional grid structure as in global routing. We assume that the wires are routed over-the-cell in their shortest Manhattan distances and multi-bend routing is allowed. Buffers are constrained to be inserted in un-occupied space for long enough wires such that the interval constraint is satisfied. Two look-up tables, which store the routability information between any pair of feasible grids, will be constructed. Finally, we count the number of blocked nets by checking with the look-up tables directly to evaluate the buffer insertion feasibility.

5.4 Buffer Planning

5.4.1 Feasible Grids

In order to improve the feasibility of buffer insertion in the final floorplan solution, a simple buffer planning method is introduced. This method is based on a table look-up approach to compute the number of blocked nets efficiently. In this approach, a floorplan will be divided into a two-dimensional grid structure as in global routing. Buffers are allowed to be inserted in un-occupied space only. Also, we define a list F of feasible grids as follows:

Definition 5.2 *A grid is a* ***feasible grid*** *in F if one or both of the following conditions are satisfied:*

1. *A grid contains a pin of a net.*
2. *A grid can hold a certain number (λ) of buffers, i.e., empty_space $\geq \lambda \times$ buffer_size, where λ is a constant.*

Condition 1 is needed because we need to consider the grids that contain a pin in our table look-up approach. Other feasible grids are required to have enough space to hold at least a certain number (λ) of buffers as required by condition 2. In our method, we will not plan the exact buffer locations for each net. Therefore, we need to tighten up the definition of a feasible grid that allows buffer insertions by introducing the constant λ. In Section 5.6, we will show how the constant λ correlates with the routability of a floorplan by some experimental analysis. In Figure 5.1, we show some examples of feasible grids in a floorplan.

5.4.2 Table Look-up Approach

In our simple buffer planning method, we are not interested in computing the best possible buffer locations. We are only interested in determining whether

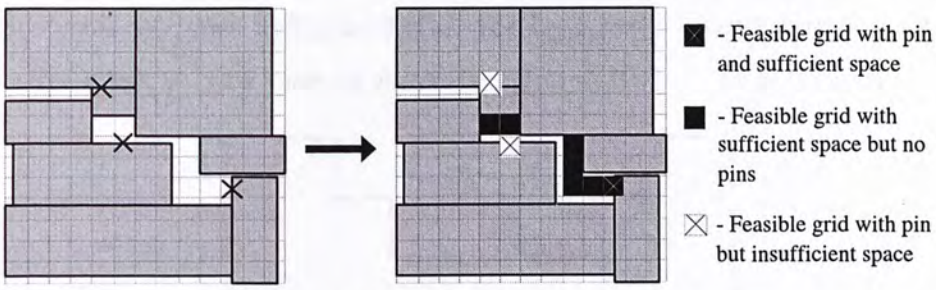


Figure 5.1: Examples of feasible grids.

a net can have all its required buffers inserted successfully. After simplifying the problem, a lot of global routing works can be saved. In order to increase the efficiency of our floorplanner, a look-up table constructed using dynamic programming is used to compute the number of blocked nets. In this approach, we define the grids that contain pin and/or have sufficient space for buffer insertions as feasible grids. It is obvious that the buffers, the source and the sink of a net are all located at these feasible grids. Therefore, a *look-up table* L storing boolean routability information between pairs of feasible grids can be used in the blocked net counting.

Definition 5.3 *A **routability look-up table** L is a two-dimensional table defined as follows:*

$$L[i,j] = \{0 \ or \ 1 \mid 1 \leq i,j \leq |F|\}$$

where i and j are the indices of list F, $|F|$ is the total number of feasible grids, and

$$L[i,j] = \begin{cases} 0 & \textit{if a net or subnet connecting F[i] and F[j] is blocked} \\ 1 & \textit{if a net or subnet connecting F[i] and F[j] is routable} \end{cases} \tag{5.2}$$

As stated before, the pins of the modules are located at feasible grids, we can determine whether a net is blocked instantly by looking at L. For a floorplan, two look-up tables (L_1, L_2) are constructed for the two routing directions as shown in Figure 5.2. For L_1, a net is routed from upper-left to lower-right, or

vice verse as shown in Figure 5.2(a). For L_2, a net is routed from upper-right to lower-left, or vice verse as shown in Figure 5.2(b).

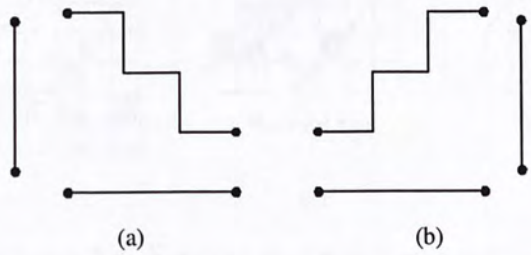


Figure 5.2: Two routing directions.

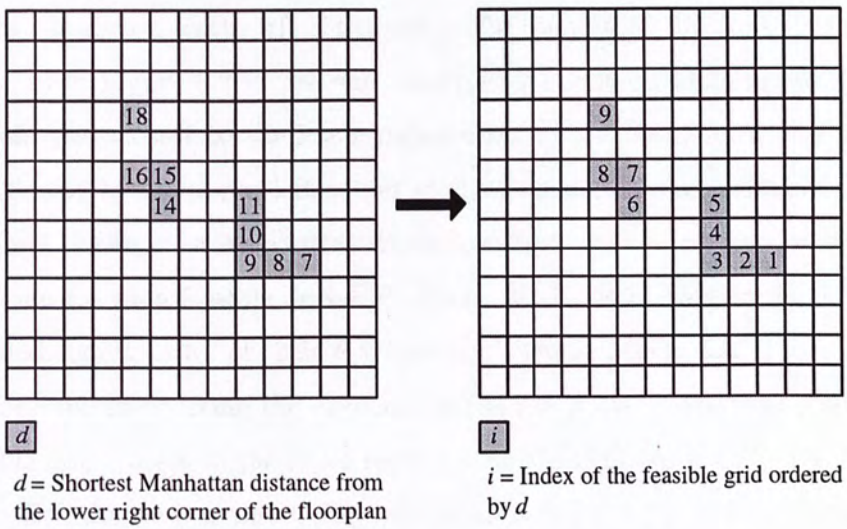


Figure 5.3: Indices of the feasible grids.

5.5 Implementation

5.5.1 Building the Look-up Tables

After locating all the feasible grids, we can start to construct the look-up tables. The look-up tables can be constructed by performing a *forward step* and then a *backward step* in each feasible grid visited as shown in Figure 5.4(a). We will demonstrate the idea of the construction and the usage of L_1 only. The

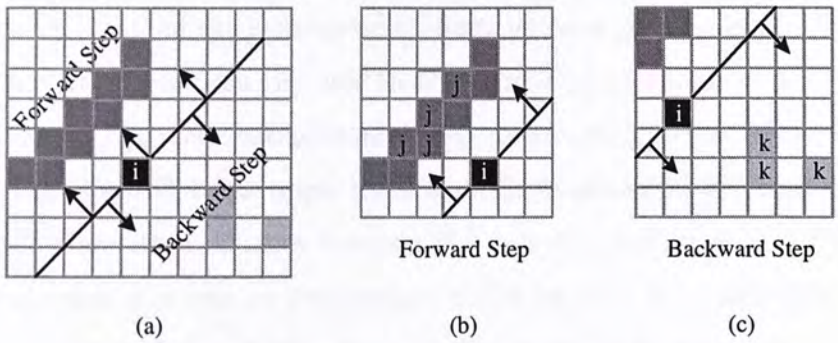


Figure 5.4: Forward step and backward step.

operations on L_2 are similar. To simplify the explanation, we assume that the source s is always on the left of the sink t. For the routing direction represented by L_1 as in Figure 5.2(a), we will construct the look-up table by visiting the feasible grid closest to the lower right corner of the floorplan to the feasible grid closest to the upper left corner of the floorplan. As a result, we need to compute the shortest Manhattan distance d_i from the lower right corner of the floorplan for each feasible grid $F[i]$. Then, the feasible grids are indiced by a non-descending order of their d values as shown in Figure 5.3. These indices will be used in ordering the elements in the list F. It means that $F[1]$ is the feasible grid closest to the lower right corner of the floorplan. For the feasible grids with the same d value, they will be arranged in an non-descending order of their distances from the lower boundary of the floorplan. For each feasible grid $F[i]$, a *reachable list* $R(i)$ will be constructed and maintained.

Definition 5.4 *A* ***reachable list*** $R(i)$ *of a feasible grid* $F[i]$ *is a list containing the set of all feasible grids* $F[k]$ *such that* $F[k]$ *is a* ***possible descendent buffer location*** *of a buffer inserted at* $F[i]$*, i.e., there exists a shortest Manhattan distance path from* $F[i]$ *to* $F[k]$ *such that buffers can be inserted at* $F[i]$*,* $F[k]$ *and some other grids along the path in such a way that the insertion constraint can be satisfied.*

The pseudo-code of the look-up table construction is given in Figure 5.5. The construction starts from the first feasible grid $F[1]$. Suppose we are visiting feasible grid $F[i]$ now. In the forward step, we are going to find all the feasible grids $F[j]$s located at the upper left side of the feasible grid $F[i]$ such that the shortest Manhattan distance between $F[i]$ and $F[j]$ is x where $x \in [low, up]$. It means that $F[j]$ can be the previous buffer location of a buffer inserted at $F[i]$ as shown in Figure 5.4(b). For each $F[j]$ found, its reachable list $R(j)$ will be updated as follows:

$$R(j) = \begin{cases} R(j) \cup R(i) \cup \{F[i]\} & \textit{if F[i] has sufficient space for} \\ & \textit{buffer insertions} \\ R(j) \cup \{F[i]\} & \textit{otherwise} \end{cases} \tag{5.3}$$

It means that if $F[j]$ can be the previous buffer location of a buffer inserted at $F[i]$, a net or subnet connecting $F[i]$ and $F[j]$ is routable. Therefore, we add the feasible grid $F[i]$ to the reachable list $R(j)$. Furthermore, if $F[i]$ have sufficient space for buffer insertions (condition 2 is satisfied), $F[i]$ can act as an intermediate buffer location for a net connecting $F[j]$ and a feasible grid in $R(i)$. It means that the feasible grids in $R(i)$, which are the possible descendent buffer locations of a buffer inserted at $F[i]$, can also be the possible descendent buffer locations of a buffer inserted at $F[j]$. As a result, we append $R(i)$ to $R(j)$. Notice that $R(i)$ has already been found as $F[i]$ is closer to the lower right corner than $F[j]$ and $F[i]$ must be visited earlier than $F[j]$ in the construction.

After the forward step, we will update the routability look-up table by examining the reachable list of $F[i]$ in the backward step. As each feasible grid $F[k]$ in $R(i)$ can be a possible descendent buffer location of a buffer inserted at $F[i]$, i.e., there exists a shortest Manhattan distance route such that buffers can be inserted along the path to satisfy the interval constraint, we will mark the table entry $L[i, k]$ as 1 to indicate that a net or subnet connecting $F[i]$ and

$F[k]$ is routable.

After visiting all the feasible grids, the construction of the look-up table is completed. By reading from the table, we will know which nets are blocked instantly.

```
buildTable(F = {F[1], F[2], ··· , F[n]})
/* Initialization */
All L[i, j] = 0
All R(i) = NULL        /* All reachable lists are empty initially*/
For i = 1 to n where n is the number of feasible grids
    /* Forward Step */
    For all F[j] which can be the previous buffer locations of F[i]
        If F[i] have sufficient space for buffer insertions
            R(j) = R(j) ∪ R(i) ∪ {F[i]}
        Else
            R(j) = R(j) ∪ {F[i]}
        Endif
    Endfor
    /* Backward Step */
    For each F[k] in R(i)
        L[i, k] = 1
    Endfor
Endfor
```

Figure 5.5: Psuedo code to build the look-up table.

5.5.2 An Example of Look-up Table Construction

Based on the feasible grids found in Figure 5.3. An example to construct the routability look-up table is shown in Figure 5.6 - 5.14. Assume that the interval constraint is $[3, 4]$ and we are going to show how the routability look-up table $L[1 \ldots 9, 1 \ldots 9]$ can be constructed.

We first visit the feasible grid $F[1]$ as in Figure 5.6. In the forward step, it is found that $F[4]$ and $F[5]$ are the possible previous buffer locations for a buffer inserted at $F[1]$. As a result, we append $R(1)$ and $\{F[1]\}$ to $R(4)$ and $R(5)$. In the backward step, as $R(1)$ is an empty list, no operations is done.

Then, $F[2]$ is being visited as shown in Figure 5.7. $F[5]$ is found to be a possible previous buffer location for a buffer inserted at $F[2]$. $R(2)$ and $\{F[2]\}$ are appended to $R(5)$, and $R(5)$ now contains $\{F[1], F[2]\}$. It means that both $F[1]$ and $F[2]$ are possible descendent buffer locations for a buffer inserted at $F[5]$. In the backward step, as $R(2)$ is an empty list, no operations is done.

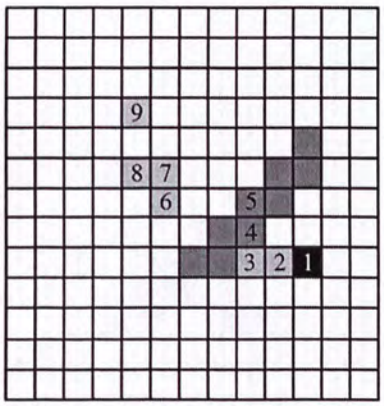
In the third iteration, $F[3]$ is being visited as in Figure 5.8. As no feasible grids is found to be a possible previous buffer location of a buffer inserted at $F[3]$, so there is no operations in the forward step. In the backward step, $R(3)$ is an empty list and no operations is needed.

When we visit $F[4]$ as in Figure 5.9, we found that $F[6]$ can be a possible previous buffer location of a buffer inserted at $F[4]$, so $R(6)$ is updated to contain $\{F[1], F[4]\}$. Since $R(4) = \{F[1]\}$, it means that a net or subnet connecting $F[4]$ and $F[1]$ is routable. Therefore, we mark the table entry $L[4, 1]$ as 1 in the backward step.

When we visit $F[5]$ as in Figure 5.10, $F[6]$ and $F[7]$ are found to be the possible previous buffer locations of a buffer inserted at $F[5]$. As a result, $R(5)$ and $\{F[5]\}$ are appended to $R(6)$ and $R(7)$. As $R(5)$ contains the feasible grids $F[1]$ and $F[2]$, so the table entries $L[5, 1]$ and $L[5, 2]$ are marked as 1 in the backward step.

After visiting $F[5]$, $F[6]$ is being visited as in Figure 5.11. Recall from Figure 5.1 that $F[6]$ is a feasible grid with pin but has insufficient space for buffer insertions. It means that $F[6]$ cannot be an intermediate buffer location. As a result, we only append $\{F[6]\}$ to the reachable list of its possible pervious buffer location $F[9]$ instead of appending $R(6)$ and $\{F[6]\}$ to $R(9)$. This is because a net or subnet connecting $F[9]$ and the feasible grids in $R(6)$ cannot be routed without violating the interval constraint if $F[6]$ must act as an intermediate buffer location. In the backward step, since $R(6) = \{F[1], F[2], F[3], F[4]\}$, the table entries $L[6,1]$, $L[6,2]$, $L[6,3]$ and $L[6,4]$ are marked as 1.

We do the same operations for $F[7]$, $F[8]$ and $F[9]$ as shown in Figure 5.12, 5.13 and 5.14 respectively. After visiting the last feasible grid $F[9]$, the construction of the look-up table is finished. If there is a net connecting $F[1]$ and $F[9]$, we can look-up the table entry L[9, 1] to examine whether it is blocked. As shown in Figure 5.14, L[9, 1] is set to 1, it means that the net is not blocked.



Forward Step:
R(5) = R(5)∪R(1)∪{F[1]} = {F[1]}
R(4) = R(4)∪R(1)∪{F[1]} = {F[1]}

Backward Step:
R(1) = { }

Figure 5.6: Table construction when visiting $F[1]$.

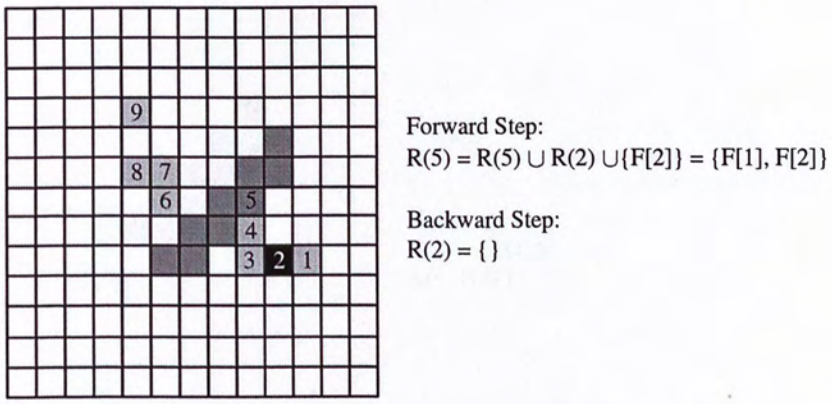


Figure 5.7: Table construction when visiting $F[2]$.

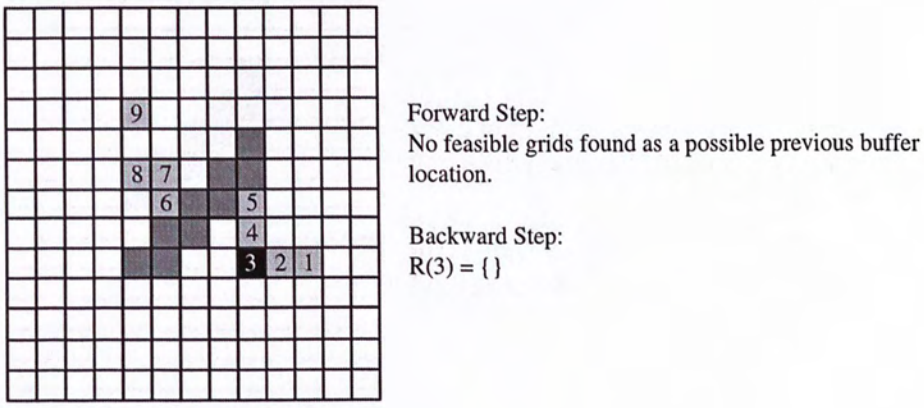
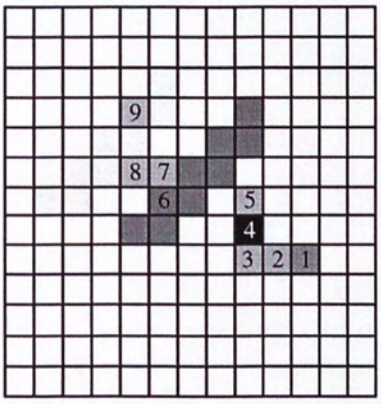


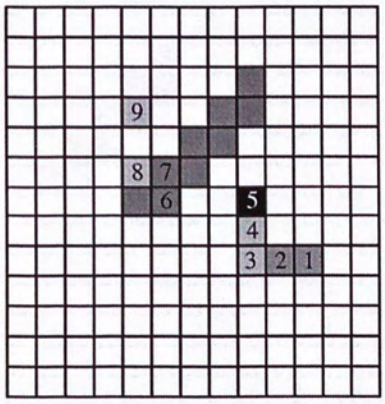
Figure 5.8: Table construction when visiting $F[3]$.



Forward Step:
R(6) = R(6)∪R(4)∪{F[4]} = {F[1], F[4]}

Backward Step:
R(4) = {F[1]}
L[4, 1] = 1

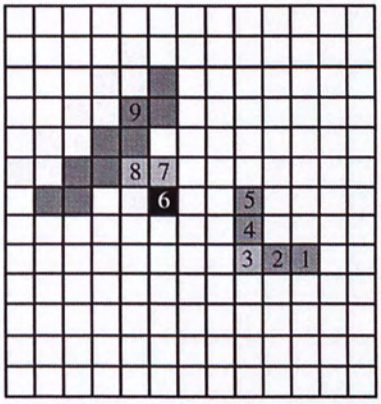
Figure 5.9: Table construction when visiting $F[4]$.



Forward Step:
R(6) = R(6)∪R(5)∪{F[5]} = {F[1], F[2], F[4], F[5]}
R(7) = R(7)∪R(5)∪{F[5]} = {F[1], F[2], F[5]}

Backward Step:
R(5) = {F[1], F[2]}
L[5, 1] = L[5, 2] = 1

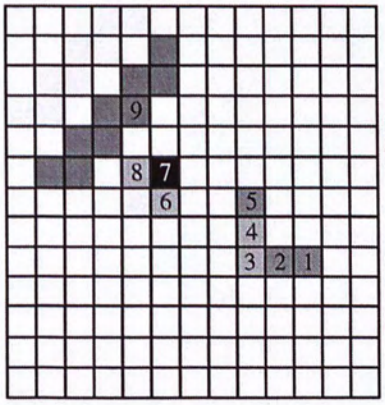
Figure 5.10: Table construction when visiting $F[5]$.



Forward Step:
R(9) = R(9) ∪ {F[6]} = {F[6]}
Grid F[6] has insufficient space for buffer insertion, so R(6) will not be appended to R(9).

Backward Step:
R(6) = {F[1], F[2], F[4], F[5]}
L[6, 1] = L[6, 2] = L[6, 4] = L[6, 5] = 1

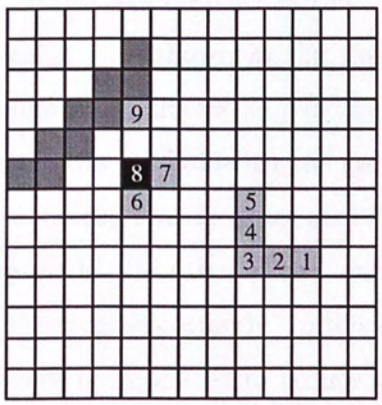
Figure 5.11: Table construction when visiting $F[6]$.



Forward Step:
R(9) = R(9) ∪ R(7) ∪ {F[7]}= {F[1], F[2], F[5], F[6], F[7]}

Backward Step:
R(7) = {F[1], F[2], F[5]}
L[7, 1] = L[7, 2] = L[7, 5] = 1

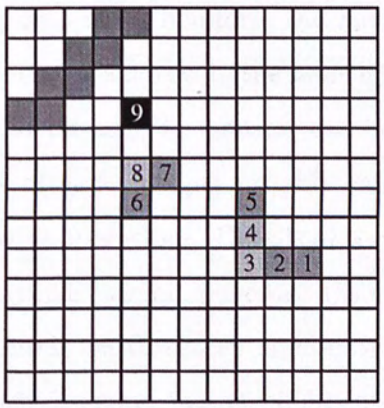
Figure 5.12: Table construction when visiting $F[7]$.



Forward Step:
No feasible grids found as a possible previous buffer location.

Backward Step:
R(8) = { }

Figure 5.13: Table construction when visiting $F[8]$.



Forward Step:
No feasible grids found as a possible previous buffer location.

Backward Step:
R(9) = {F[1], F[2], F[5], F[6], F[7]}
L[9, 1] = L[9, 2] = L[9, 5] = L[9, 6] = L[9, 7] = 1

Figure 5.14: Table construction when visiting $F[9]$.

5.5.3 A Faster Approach for Building the Look-up Tables

The above section has described the basic methodology to construct the look-up table. As shown in the pseudo-code in Figure 5.5, the running time is dominated by the for-loop in line 4, 6 and 14. In line 14, the number of feasible grids in the reachable list can grow rapidly and it will affect the performance of the construction. In order to reduce the runtime, the number of feasible grids in reachable lists should be reduced, while the routability look-up table should be constructed correctly. In this section, a *reduced reachable list* with a smaller size will be proposed to reduce the runtime. Also, we have changed the data structure of the routability look-up table such that the operations on different elements of a reduced reachable list can all be performed simultaneously in one operation. A *page table* with bitwise operations will be introduced to compute the routability information correctly and efficiently. Notice that in the original method, we just need to store a bit '0' or '1' for each table entry, so one bit is enough for one entry. In this faster implementation, we will use a 64-bit long integer variable as one page to store 64 table entries as in Figure 5.15. Therefore, the memory requirement of the floorplanner can be reduced. Each row in the table is refereed as a directory, which contains a set of 64-bit pages. The runtime can be reduced because the operations on different bits of each integer variable can be performed simultaneously by some fast bitwise operations. Details of the optimized algorithm is shown in Figure 5.16. A 3-tuple (*dir_no, page_no, offset*) is used to address a table entry $L[i, j]$ where *dir_no* is the directory number which is equal to i, *page_no* is the page number which is computed as $\lfloor j/64 \rfloor$ and *offset* is the bit position in one page and is computed as $j - 64*page\_no$. In this section, we use $D[i]$ to denote the

directory i where

$$D[i] = \bigcup_{j=1}^{|F|} L[i,j]$$

Definition 5.5 *A* ***reduced reachable list*** $R'(i)$ *of a feasible grid* $F[i]$ *is a list containing the set of feasible grids* $F[k]$ *such that* $F[k]$ *can be a* ***possible next buffer location*** *of a buffer inserted at* $F[i]$.

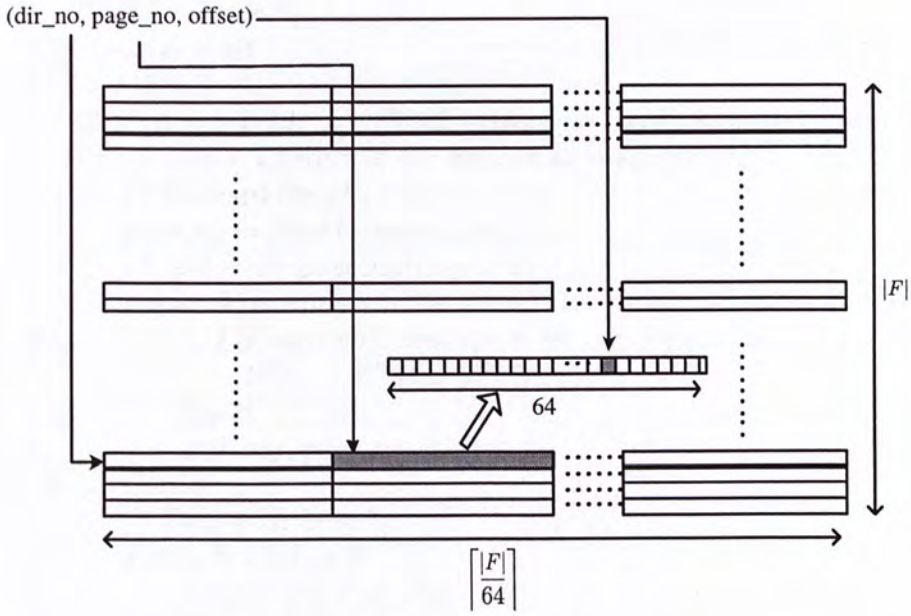


Figure 5.15: A page table structure.

Similar to Section 5.5.1, each feasible grid visit is divided into two phases: forward step and backward step. When we visit $F[i]$ in the forward step, we will first find all the possible previous buffer locations $F[j]$ of a buffer inserted at $F[i]$ in the forward step. We will then update the table entry $L[j,i]$ to 1. The index i will be translated into (*page_no=x*, *offset=y*) and the bit at directory j, page x and offset y will be set to 1 to indicate that there exists a route between $F[i]$ and $F[j]$ which can be routed in its shorest Manhattan distance and satisfy the interval constraint. Then, we will update the reduced

```
fastBuildTable(F = {F[1], F[2], ··· , F[n]})
1. /* Initialization */
2. page_size = 64
3. All L[i] = 0
4. All R'(i) = NULL        /* All reduced reachable lists are empty */
5. For i = 1 to n where n is the number of feasible grids
6.      /* Forward Step */
7.      page_no = floor(i/page_size)
8.      offset = i − page_no*page_size
9.      For all F[j] which can be the previous buffer locations of F[i]
10.         If F[i] have sufficient space for buffer insertions
11.             R'(j) = R'(j) ∪ {F[i]}
12.         Endif
13.         setBit(j, page_no, offset, 1)
14.     Endfor
15.     /* Backward Step */
16.     For each F[k] in R'(i)
17.         D[i] = D[i] OR D[k]
18.     Endfor
19.Endfor
```

Figure 5.16: Psuedo code to build the look-up table based on bitwise operations.

reachable list $R'(j)$ as follows:

$$\begin{cases} R'(j) = R'(j) \cup \{F[i]\} & \textit{if F[i] has sufficient space for buffer insertions} \\ \textit{No update} & \textit{otherwise} \end{cases} \tag{5.4}$$

If $F[i]$ have sufficient space for buffer insertions, $F[i]$ can act as an intermediate buffer location. If a net or subnet connecting $F[i]$ and $F[k]$ is routable ($L[i,k] = 1$), a net or subnet connecting $F[j]$ and $F[k]$ is also routable ($L[j,k] = 1$) with $F[i]$ as an intermediate buffer location. We will update the values of $L[j,k]$ according to the values of $L[i,k]$ when $F[j]$ is being visited. Therefore, we append $\{F[i]\}$ to $R'(j)$, and we can refer back to $F[i]$ when we visit $F[j]$ in some later iteration.

When we visit $F[j]$, we will update the routability look-up table by examining the feasible grids in the reduced reachable list $R'(j)$ in the backward step. Assume that $F[i]$ is in $R'(j)$. Recall from the operations done when $F[i]$ is being visited, $L[j,i]$ has already been updated. Now, we only need to update $L[j,k]$ where $F[k]$ is a possible descendent buffer location of a buffer inserted at $F[i]$. This update can be done by a bitwise *OR* operation to propagate the routability information from $F[i]$ to $F[j]$ as in line 17 of the pseudo code in Figure 5.16. Notice that the routability information of $F[i]$ has already been computed and all its possible descendent buffer locations $F[k]$ are found and marked. Suppose a net or subnet connecting $F[i]$ and $F[k]$ is routable, $L[i,k]$ should have already been marked as 1. If $F[i]$ has sufficient space to insert buffers, a net or subnet connecting $F[j]$ and $F[k]$ is also routable by using $F[i]$ as an intermediate buffer location. The table entry $L[j,k]$ can also be set to 1 by performing an *OR* operation on the directory i and j as:

$$\begin{aligned} D[j] &= D[j] \quad OR \quad D[i] \\ L[j,k] &= L[j,k] \quad OR \quad L[i,k] \quad \forall k = 1, 2, \cdots, |F| \end{aligned}$$

5.5.4 An Example of the Faster Look-up Table Construction

An example to illustrate this faster algorithm is shown in Figure 5.17-5.22. In Figure 5.17, $F[1]$ is being visited. In the forward step of $F[1]$, it is found that $F[4]$ and $F[5]$ can be the possible previous buffer locations of a buffer inserted at $F[1]$. Therefore, we set the table entry $L[4,1]$ and $L[5,1]$ as 1 using the *setBit* operation. Furthermore, we append $\{F[1]\}$ to $R'(4)$ and $R'(5)$. As $R'(1)$ is an empty list, no bits is needed to be set in the backward step.

In the second iteration, $F[2]$ is being visited as in Figure 5.18. $F[5]$ is found to be a possible previous buffer location for a buffer inserted at $F[2]$ in the forward step. The table entry $L[5,2]$ is set to 1 using the *setBit* operation. Also, $\{F[2]\}$ are appended to $R'(5)$, and $R'(5)$ now contains $\{F[1], F[2]\}$. As $R'(2)$ is an empty list, no bits is needed to be set in the backward step.

In the third iteration, $F[3]$ is being visited as in Figure 5.19. As no feasible grids is found to be a possible previous buffer location of a buffer inserted at $F[3]$, there is no operations in the forward step. In the backward step, as $R'(3)$ is an empty list, no bits is set.

After visiting $F[3]$, $F[4]$ is being visited as in Figure 5.20. In the forward step, $F[6]$ is found to be a possible previous buffer location of a buffer inserted at $F[4]$. Therefore, $F[4]$ is appended to the reduced reachable list $R'(6)$ and $L[6,4]$ is marked as 1. In the backward step, as $R'(4)$ contains $\{F[1]\}$, it means that $F[1]$ can be the next possible buffer location of a buffer inserted at $F[4]$. Therefore, we propagate the routability information of $F[1]$ to $F[4]$ using the bitwise *OR* operation.

In the visit of $F[5]$ as shown in Figure 5.21, $F[6]$ and $F[7]$ are found to be the possible previous buffer locations of a buffer inserted at $F[5]$. As a result, $F[5]$ is appended to $R'(6)$ and $R'(7)$. At the same time, the table entries $L[6,5]$ and $L[7,5]$ are marked as 1. In the backward step, $R'(5)$ is found to contain $F[1]$ and $F[2]$, so, we will propagate the routability information of $F[1]$ and $F[2]$ to $F[4]$ using the bitwise *OR* operation.

When we visit $F[6]$ as in Figure 5.22, as $F[6]$ is a feasible grid with pin but have insufficient space for buffer insertions, we will not append $\{F[6]\}$ to $R'(9)$ which is a possible previous buffer location of a buffer of $F[6]$. It is because $F[6]$ cannot be an intermediate buffer location and it is not needed to propagate its routability information to $F[9]$. Therefore, we just set the table entry $L[9,6]$ to 1 in the forward step. In the backward step of $F[6]$, the routability information of $F[4]$ and $F[5]$ are propagated to $F[6]$ by using the bitwise *OR* operation. As shown in Figure 5.22, after the backward step, the bits $L[6,1]$ and $L[6,2]$ are successfully set to 1 (because $L[4,1]$, $L[5,1]$ and $L[5,2]$ are 1). This gives the same result as in the fundamental method in Figure 5.11. The table entities $L[6,1]$, $L[6,2]$, $L[6,4]$ and $L[6,5]$ are all set to 1.

5.5.5 I/O Pin Locations

In our method, multi-pin nets will be first decomposed into two-pin nets. These two-pin nets will be evaluated independently. We use the *minimum spanning tree (MST)* method to decompose the multi-pin nets. Then, we use the *intersection-to-intersection* method to determine the locations of the I/O pins. The grids containing the I/O pins will be the source and sink of the route in our simple buffer planning method.

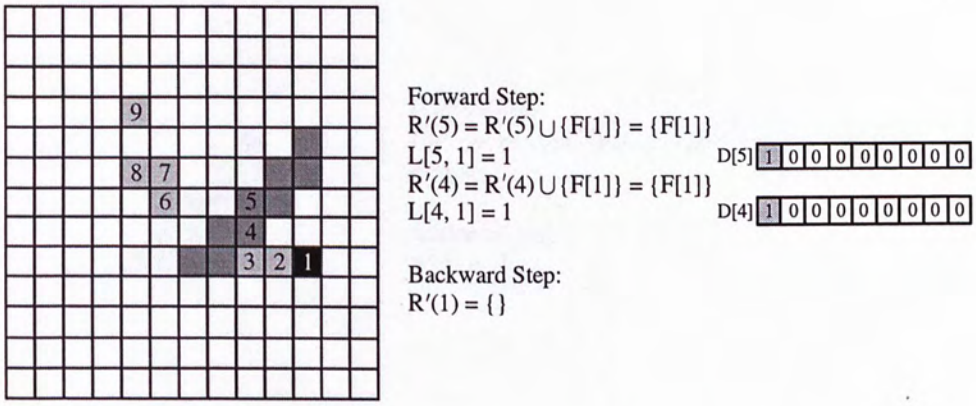


Figure 5.17: A faster table construction when visiting $F[1]$.

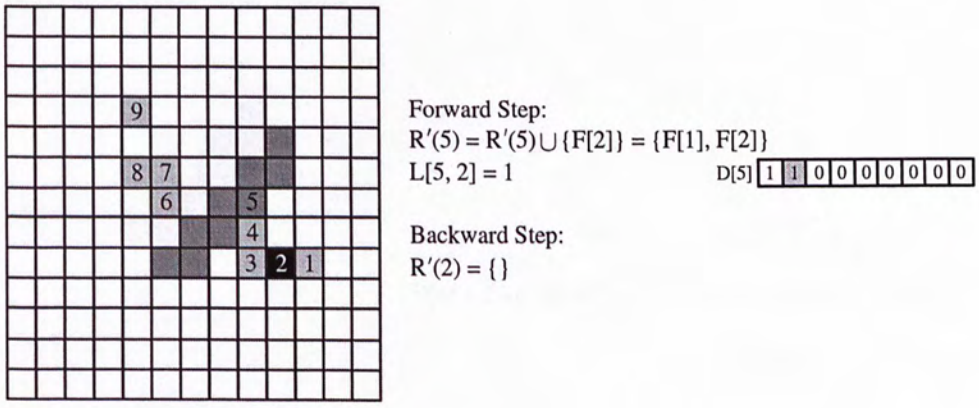
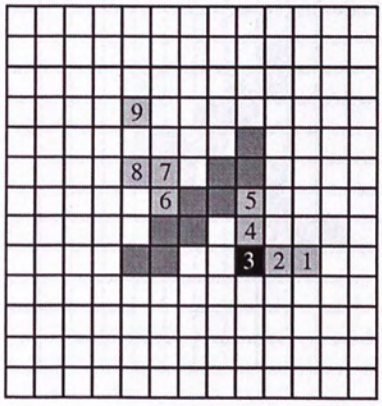


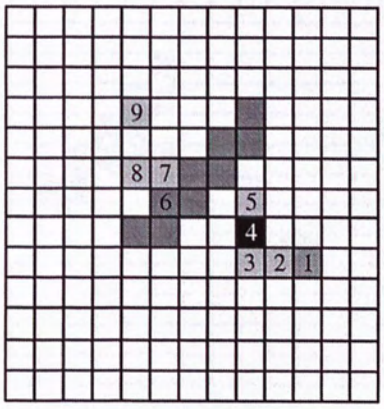
Figure 5.18: A faster table construction when visiting $F[2]$.



Forward Step:
No feasible grids found as a possible previous buffer location.

Backward Step:
$R'(3) = \{\}$

Figure 5.19: A faster table construction when visiting $F[3]$.



Forward Step:
$R'(6) = R'(6) \cup \{F[4]\} = \{F[4]\}$
$L[6, 4] = 1$

Backward Step:
$R'(4) = \{F[1]\}$
$D[4] = D[4]$ OR $D[1]$

D[6]	0	0	0	1	0	0	0	0	0
D[4]	1	0	0	0	0	0	0	0	0
				OR					
D[1]	0	0	0	0	0	0	0	0	0
				||					
D[4]	1	0	0	0	0	0	0	0	0

Figure 5.20: A faster table construction when visiting $F[4]$.

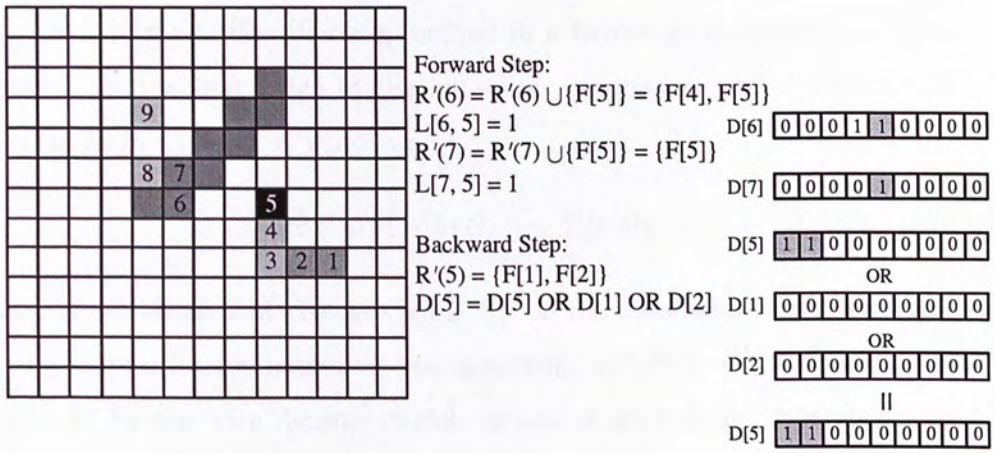


Figure 5.21: A faster table construction when visiting $F[5]$.

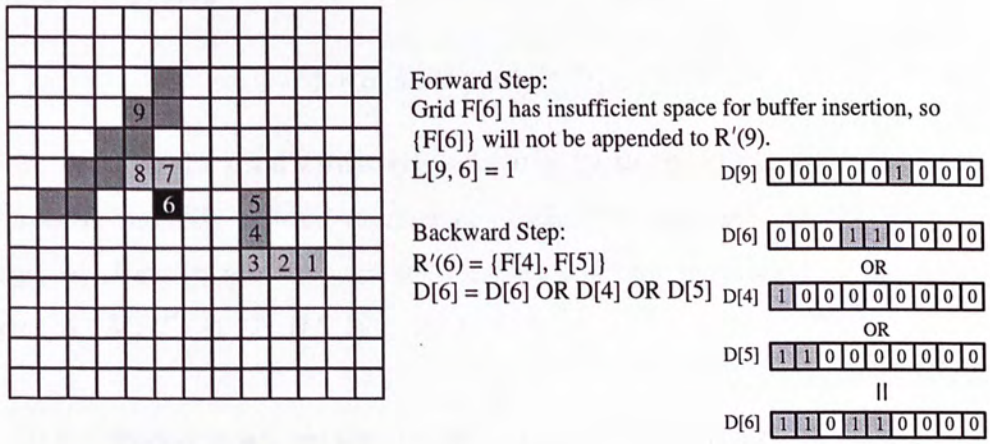


Figure 5.22: A faster table construction when visiting $F[6]$.

5.5.6 Cost Function

In our floorplanner, we combine the wire density evaluation model in Chapter 4 with this simple buffer planning method in a two-stage simulated annealing process. In the first stage of the annealing process, we use the same cost function as in Chapter 4, which is:

$$cost = A + \alpha(HP) + \beta(WD) \tag{5.5}$$

where A is the area of the floorplan, HP is the total wirelength estimated by the half-perimeter bounding box approach, and WD is the wire density estimated by the wire density model. α and β are weights to describe the importance of these three terms. In the first stage, α and β are set such that the ratio of the importance of the three terms $A : HP : WD$ is $2 : 2 : 1$.

After about 40% of the total number of iterations, the second stage of the annealing process is started. In the second stage, the area, wirelength, wiring congestion and buffer insertion feasibility will be considered. The cost function is shown as follows:

$$cost = A + \alpha(MST) + \beta(WD) + \gamma(BN) \tag{5.6}$$

where MST is the total wirelength estimated by the minimum spanning tree approach, and BN is total number of blocked two-pin nets. In the second stage, α, β and γ are set such that the ratio of the importance of the four terms $A : MST : WD : BN$ is $2 : 2 : 1 : 1$.

In the second stage, we use the MST approach to estimate wirelength instead of the half-perimeter bounding box approach. It is because we want to obtain a more accurate estimation of the wirelength when the solution is being fine toned. Besides, the nets have already been decomposed into two-pin nets

using the MST approach in the second phase in order to evaluate the buffer insertion feasibility. Therefore, the MST wirelength can be computed efficiently without spending much extra effort.

5.5.7 Complexity

The goal of our simple buffer planning method is to evaluate a floorplanner on its buffer insertion feasibility efficiently and accurately. By using the table look-up approach, the decision on whether a net is blocked can be made in constant time.

If we construct a look-up table using the original method described in Section 5.5.1, we need to visit all the feasible grids in F once. In each visit, we will examine the feasible grids that can be a possible previous buffer location of a buffer inserted at the visiting grid in the forward step and the feasible grids in the reachable list of the visited grid in the backward step. The number of feasible grids that can be a possible previous buffer location is bounded by the interval $[low, up]$, which is a constant. Therefore, the pre-processing time of the look-up table approach is $O(R_M|F|)$, where R_M is the maximum number of feasible grids in a reachable list. However, if we construct a look-up table using the faster method described in Section 5.5.3, we will examine the feasible grids that can be a possible previous buffer location or a possible next buffer location of a buffer inserted at the visiting grid in each visit. The number of these feasible grids is bounded by the interval $[low, up]$, which is a constant. As a result, the pre-processing time of the faster look-up table approach is $O(|F|)$. After constructing the table using the faster method, we have to examine each decomposed two-pin net one by one to see if it is blocked. As a result, the total time complexity of our simple buffer planning method is $O(|F| + k)$ for

each iteration of the annealing process where k is the total number of two-pin nets.

5.6 Experimental Results

We have implemented a floorplanner with our simple buffer planning method and wire density model for testing. Three MCNC benchmarks (*ami33, ami49* and *playout*) are used. In addition, three more data sets (*n2000, n2500* and *n3000*) are generated randomly to demonstrate the performance of our floorplanner for complex circuit designs. The detailed specifications of the data sets are shown in Table 4.1. The experiments are performed using a PC with a Pentium IV 1.4GHz processor and 512MB memory. We use a simple global router to evaluate the performance of the floorplanner. We have performed two sets of experiments. In Section 5.6.1, we will show an experiment to determine a value for λ in defining feasible grids. In Section 5.6.2, we will compare the results of our floorplanner with other interconnect-driven floorplanners [17, 18].

5.6.1 Selected Value for λ

In Section 5.4.1, we describe the definition of feasible grid. One of the conditions that a grid is a feasible grid is that it can hold at least λ buffers. However, it is difficult to find a suitable value of λ for different data sets.

It is expected that when λ is too small, even a grid with not much space will be treated as a feasible grid. In fact, it has insufficient space to hold the required number of buffers. It results in an increase in the number of unroutable wires reported by the simple global router. On the other hand, when the value of λ is too large, there will be less feasible grids in the floorplan solution. This affects the accuracy of the estimations obtained by the simple buffer planning

method. It is because a grid, that have sufficient space for the required buffer insertions, may not be treated as a feasible grid in the buffer planning process. It is expected that as the number of nets in a circuit increases, a larger λ is needed. It is because more buffers are needed to be inserted to achieve the optimize delay of these complex circuits designs, for example, *n2500* and *n3000*. Therefore, the feasible grids defined by a larger λ can provide more buffer insertions supplies (silicon resources) for the increased buffer insertions demands.

We have performed an experiment to explore the relationship between λ and the number of unroutable wires reduced. The experimental results for the three MCNC benchmarks are shown in Figure 5.23 and that for the three complex circuit designs are shown in Figure 5.24. We choose a value for λ that leads to a maximum reduction in the number of unroutable wires for each data set as shown in Table 5.2. The experimental results show the influences of different λ values on the number of unroutable wires, which is the same as the prediction.

Data Set	ami33	ami49	playout	n2000	n2500	n3000
λ	25	50	250	250	500	750
Number of unroutable wires	6.88	8.72	107.60	430.17	694.46	988.40

Table 5.2: Selected values of λ for different data sets.

5.6.2 Performance of Our Floorplanner

In this section, we compare the results of our floorplanner with the other two interconnect-driven floorplanners [17, 18]. Table 5.3 shows the experimental results of the MCNC benchmarks and Table 5.4 shows the results of the complex circuit designs. As defined before, the term *unroutable wire* refers to the

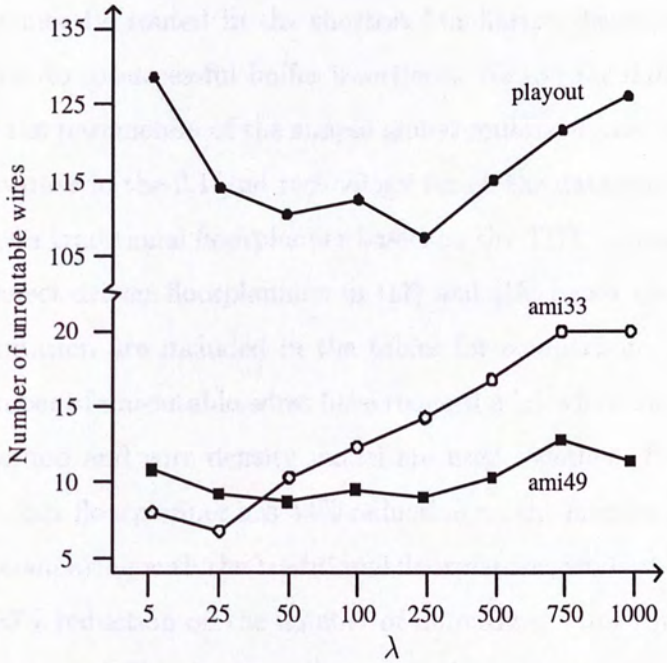


Figure 5.23: Different λ values for the MCNC benchmarks.

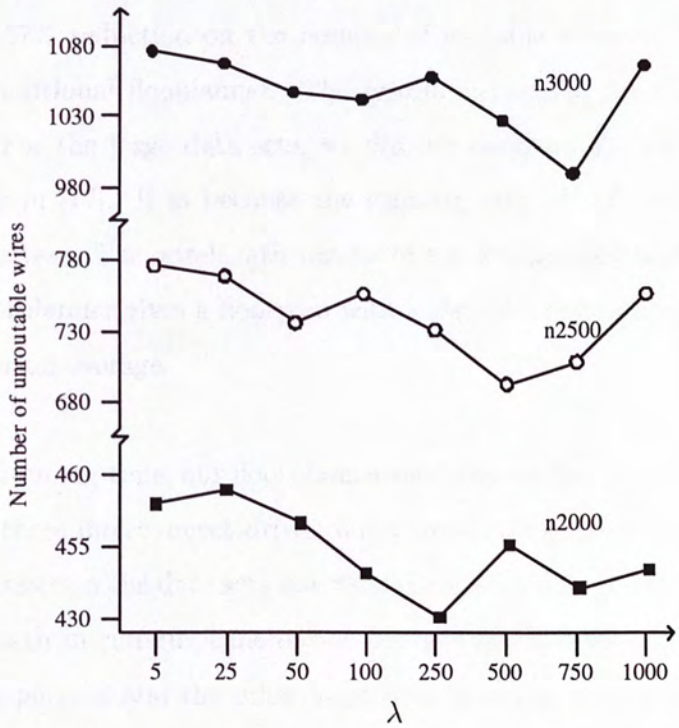


Figure 5.24: Different λ values for the randomly generated complex circuit designs.

wire that cannot be routed in the shortest Manhattan distance due to congestion or due to unsuccessful buffer insertions. We use the data in paper [9] to compute the parameters of the simple global router for evaluation. We use the feature values in the $0.18\mu m$ technology for all the data sets. In addition, the results of a traditional floorplanner based on the TBT representation, and the interconnect-driven floorplanners in [17] and [18] based on the sequence pair representation are included in the tables for comparison. Results show that the number of unroutable wires have reduced a lot when the simple buffer planning method and wire density model are used together. For the MCNC benchmarks, our floorplanner has 44% reduction on the number of unroutable wires when comparing with the traditional floorplanner, while the floorplanner in [17] has 33% reduction on the number of unroutable wires when comparing with the traditional floorplanner. However, when comparing with the floorplaner in [18], [18] shows a better performance in reducing unroutable wires. It shows a 57% reduction on the number of routable wires when comparing with the traditional flooplanner. The results are similar for the three large data sets. For the large data sets, we did not compare the results with the floorplanner in [17]. It is because the running time of [17] will be too long to finish the test. The wirelength results of the four floorplanners are similar but our floorplanner gives a floorplan with a sightly larger deadspace of about 3.92% larger on average.

For the running time, our floorplanner have the shortest time per iterations among the three interconnect-driven floorplanners. The total running time of our floorplanner on the data sets *ami33* and *ami49* are longer than that of [18] but the growth in running time of our floorplanner is much slower than that of [18]. For *playout* and the other large data sets, our running time is faster than that of [18] although the number of iterations is larger in our method. It shows that our table look-up approach is more adaptable for complex circuit

designs. The simple buffer planning method helps in reducing the number of unroutable wires that have unsuccessful buffer insertions, while the wire density evaluation model helps in reducing the wiring congestion. Using these two models together, an interconnect-optimized floorplan can be obtained in less than 14 minutes for a circuit with three thousand nets.

Floorplanner	Deadspace (%)	Wire Length ($10^3 \mu m$)	Number of Unroutable Wires	Runtime (s)	Time per Iterations (ms)	Iterations
ami33						
Traditional	10.31	21.25	15.03	21.05	0.16	129702
[17]	11.80	20.59	9.63	678.45	9.15	74163
[18]	12.39	23.21	3.88	290.69	3.92	74163
Ours	11.63	22.18	6.88	653.75	4.23	154553
ami49						
Traditional	10.87	386.45	14.53	25.54	0.20	129702
[17]	10.80	379.80	10.00	789.46	10.64	74163
[18]	11.24	384.55	6.75	369.18	4.98	74163
Ours	18.67	398.39	8.72	688.73	4.49	153282
playout						
Traditional	10.25	284.78	170.54	30.64	0.24	129702
[17]	10.38	290.56	115.88	3498.23	37.74	92703
[18]	11.74	274.74	94.50	912.21	9.84	92703
Ours	16.83	298.90	107.60	720.44	4.66	154443

Table 5.3: Comparison of our simple buffer planning method with other floorplanners on MCNC benchmark.

5.7 Conclusion

In this chapter, we propose a new buffer planning approach in floorplanning called the simple buffer planning method. This method aims at providing an algorithm to count the number of blocked nets efficiently. It is based on a table look-up approach with bitwise operation and dynamic programming. Feasible grids are defined as grids that contain pin and/or have sufficient space for buffer insertions. The routability information between any pair of feasible grids are stored in the routability look-up table. The construction of the routability look-up table can be done in linear time and the decision on

Floorplanner	Deadspace (%)	Wire Length ($10^3 \mu m$)	Number of Unroutable Wires	Runtime (s)	Time per Iterations (ms)	Iterations
n2000						
Traditional	11.56	102.60	581.75	35.41	0.27	129702
[18]	15.35	114.65	416.47	1316.81	14.20	92703
Ours	17.72	114.42	430.17	761.50	4.95	153988
n2500						
Traditional	14.08	141.09	887.35	37.54	0.29	129702
[18]	16.51	155.37	675.61	1701.46	18.35	92703
Ours	20.26	173.21	694.46	774.83	4.99	155199
n3000						
Traditional	16.37	177.77	1299.75	46.95	0.36	129702
[18]	18.45	194.47	970.45	1916.85	20.68	92703
Ours	20.88	208.80	988.40	832.05	5.34	155909

Table 5.4: Comparison of our simple buffer planning method with other floorplanners on complex circuit designs.

whether a net is blocked can be made in constant time by accessing the table directly. In our floorplanner, we put this simple buffer planning method with the wire density evaluation model together in a two-stage simulated annealing process. Experimental results show that our floorplanner can reduce more unroutable wires than the floorplanner using probabilistic approach [17]. When comparing with the floorplanner in [18], [18] can reduce more unroutable wires. However, the running time of our floorplanner is faster than that of [18] for more complex circuit designs. It is because the running time of our approach grows in linear time with respect to the circuit size. Therefore, our approach is more adaptable to complex circuit designs in the deep submicron technology.

Chapter 6

Conclusion

At the beginning of this thesis, we have given an overview of the VLSI design cycle with emphasis on physical design. In the physical design cycle, we focus on the floorplanning phase. We have reviewed the literatures on floorplan representation and interconnect-driven floorplanning. We are interested in interconnect-driven floorplanning as it is a major concern in the deep submicron VLSI design. We have studied different approaches to estimate wiring congestion and plan the buffer locations in floorplanning.

In our research, we use a mosaic floorplan representation, twin binary trees (TBT), in our floorplanners. We propose two methods, which aim at providing efficient and novel approach to address the interconnect optimization problem. The time complexity of the two methods are linear. The first approach we proposed is the wire density model, which is used to estimate the wiring congestion of a floorplan. We use TBT as the floorplan representation because the tree structures can define the regions for evaluation naturally. We have made use of the fast and simple tree algorithm, the LCA algorithm, to facilitate the efficiency of our congestion estimation process. By using the regions defined by the TBT and the mirror TBT, sufficient samples can be taken for congestion evaluation. The time complexity of the whole congestion estimation method is linear with respect to the number of two-pin nets. Experiments have shown

that this congestion evaluation method is efficient and accurate when dealing with complex circuit designs. For a circuit with three thousand nets, the number of unroutable wires can be greatly reduced by about 18% in less than three minutes when comparing with a traditional floorplanner which considers area and wirelength only.

The second approach we proposed is the simple buffer planning method, which addresses the buffer insertion problem. This method aims at providing an algorithm to count the number of blocked nets efficiently. It is based on a table look-up approach with bitwise operation and dynamic programming. Feasible grids are defined as grids that contain pin and/or have sufficient space for buffer insertions. The routability between all pairs of feasible grids are stored in a routability look-up table. The construction of the look-up table can be done in linear time and the decision on whether a net is blocked can be made in constant time by accessing the table directly. In our floorplanner, we put this simple buffer planning method with the wire density evaluation model together in a two-stage simulated annealing process. Experimental results show that our floorplanner can reduce more unroutable wires than the floorplanner using probabilistic approach [17]. When comparing with the floorplanner in [18] that finds the best buffer locations, [18] can reduce more unroutable wires. However, the running time of our floorplanner is faster than [18] for large curcuits. The running time of our approach grows in linear time with respect to the number of feasible grids and the number of two-pin nets. Therefore, our approach is more adaptable to complex circuit designs.

Appendix A

An Efficient Algorithm for the Least Common Ancestor Problem

In 2000, Bender and Colton proposed a simple and fast algorithm for the *Least Common Ancestor (LCA)* problem in paper [41]. In this chapter, the algorithm is described. This algorithm models the LCA problem to the *range minimum query (RMQ)* problem. A faster algorithm for RMQ can be done in constant time after a pre-processing time of $O(nlgn)$. The definition of LCA and RMQ are as follows:

Definition A.1 *Given a rooted binary tree T with n nodes, the least common ancestor (LCA) of nodes u and v, $LCA(u, v)$, is the node furthest from the root that is an ancestor of both u and v.*

Definition A.2 *Given an array A with n elements, the range minimum query of index i and j, $RMQ_A(i, j)$, is the index of the smallest element in the subarray $A[i \ldots j]$.*

The reduction of the LCA problem to the RMQ problem starts from expressing the rooted tree T with n nodes into three arrays (E, L, R). The array

$E[1, \ldots, 2n-1]$ stores the nodes visited in an *Euler Tour* of T, where $E[i]$ is the label of the ith node visited in the Euler tour. The array $L[1, \ldots, 2n-1]$ stores the level of the nodes in E, where $L[i]$ is the level of node $E[i]$. The level is defined as the distance between the node and the root. Finally, the array $R[1, \ldots, n]$ is the index of the representative of node i, which is defined as the index of the first occurrence of node i in E. The time complexity of this transformation is $O(n)$. An example is shown in Figure A.1.

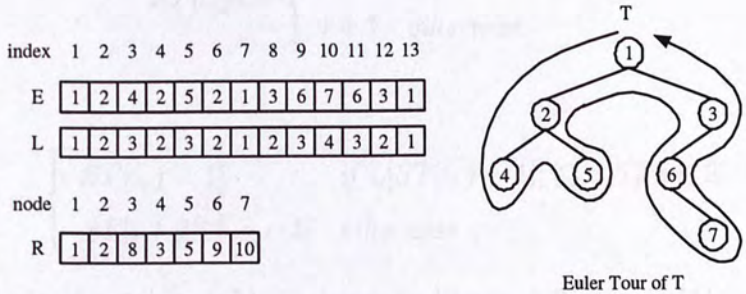


Figure A.1: An example of the three arrays (E, L, R).

After modelling T into the three arrays (E, L, R), we can compute $LCA(u, v)$ by finding the node with lowest level between the first occurrences of u and v in the Euler tour, i.e., $E[R[u], \ldots, R[v]]$ or $E[R[v], \ldots, R[u]]$. The index of this lowest level node can be found by $RMQ_L(R[u], R[v])$. Consequently, $LCA(u, v)$ is computed as $E[RMQ_L(R[u], R[v])]$. It takes $O(n^2)$ running time to build a table $M[1 \ldots n, 1 \ldots n]$ storing the RMQ of all pairs (u, v). It means that we need a $O(n^2)$ pre-processing time to achieve a constant time LCA query. In paper [41], a faster approach is proposed.

Besides building a $n \times n$ size brute-force table, a faster approach builds a sparse table $ST[1 \ldots n, 1 \ldots \lfloor lgn \rfloor]$ with a smaller size, which is $n\lfloor lgn \rfloor$. A sparse table entry $ST[i, j]$ stores the value of $RMQ_L(i, i + 2^j)$, i.e., the index of the lowest level node in the sub-array of L starting with index i and size 2^j.

Instead of storing the RMQ of all nodes pairs in the original approach, this faster approach stores the RMQ of different blocks of nodes with fixed size 2^j systematically where $1 \leq j \leq \lfloor lgn \rfloor$. Therefore, the size of the table can be smaller and the pre-processing time can be faster. Each entry $ST[i,j]$ can be computed by comparing the two minima of its two constituent blocks of size 2^{j-1} using dynamic programming as follow:

if $j = 1$

$$ST[i,j] = \begin{cases} i & \textit{if } L[i] \leq L[i+1] \\ i+1 & \textit{otherwise} \end{cases}$$

if $j > 1$

$$ST[i,j] = \begin{cases} ST[i,j-1] & \textit{if } L[ST[i,j-1]] \leq L[ST[i+2^{j-1}, j-1]] \\ ST[i+2^{j-1}, j-1] & \textit{otherwise} \end{cases}$$

An example of a sparse table is shown in Figure A.2. The $RMQ(i,j)$ can be computed by comparing the minima of the two overlapping blocks of size 2^k, where $k = \lfloor (j-i) \rfloor$ that covers the entire range $[i,j]$.

$$RMQ_L(i,j) = \min\{ST[i,k], ST[i-2^k+1,k]\} \tag{A.1}$$

Finally, the $LCA(u,v)$ can be computed as $E[RMQ_L(R[u], R[v])]$ in constant time with $O(nlgn)$ pre-processing time.

j \ i	1	2	3	4	5	6	7	8	9	10	11	12	13
1	1	2	4	4	6	7	7	8	9	11	12	13	
2	1	2	4	7	7	7	7	8	12	13			
3	1	7	7	7	7	7							

Figure A.2: An sparse table example with size 13.

Bibliography

[1] N. A. Sherwani, *Algorithms for VLSI Physical Design Automation*, Kluwer Academic Publishers, Massachusetts 02061, USA, third edition, 1999.

[2] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, Equation of state calculations by fast computing machines, in *Journal of Chemical Physics*, volume 90, pages 233–241, 1953.

[3] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, Optimization by simulated annealing, in *Science*, volume 220, 4598, pages 671–680, 1983.

[4] S. Koakutsu, M. Kang, and W. W.-M. Dai, Genetic simulated annealing and application to non-slicing floorplan design, in *ACM/SIGDA Physical Design Workshop*, pages 134–141, April 1996.

[5] J. H. Holland, Adaptation in natural and artificial systems, in *Ann Arbor, MI: The University of Michigan Press*, 1975.

[6] X. Hong et al., Corner block list: An effective and efficient topological representation of non-slicing floorplan, in *Proceedings of the IEEE/ACM international conference on Computer-aided design*, pages 8–12, 2000.

[7] B. Yao, H. Chen, C. K. Cheng, and R. Graham, Revisting floorplan representations, in *Proceedings of the international symposium on Physical design*, pages 138–143, 2001.

[8] I. T. R. for Semiconductors., Itrs 2001 edition - interconnect, in *ITRS*, 2001.

[9] J. Cong, Challenges and opportunities for design innovations in nanometer technologies, in *SRC Design Sciences Concept Paper*, 1997.

[10] G. E. Moore, Gramming more components onto integrated circuits, in *Electronics*, volume 38, April 1965.

[11] S. Sutanthavibul and Rosen., An analytical approach to floorplan design and optimization., in *IEEE Transaction on Computer-Aided Design*, pages 761–769, June 1991.

[12] T. Chen and M. K. H. Fan, On convex formulation of the floorplan area minimization problem, in *Proceedings of the 1998 international symposium on Physical design*, pages 124–128, ACM Press, 1998.

[13] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani, Rectangle-packing-based module placement, in *Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, pages 472–479, IEEE Computer Society Press, 1995.

[14] Y.-C. Chang, Y.-W. Chang, G.-M. Wu, and S.-W. Wu, B*-trees: a new representation for non-slicing floorplans, in *Proceedings of the 37th conference on Design automation*, pages 458–463, ACM Press, 2000.

[15] E. F. Y. Young, C. C. N. Chu, and Z. C. Shen, Twin binary sequences: A non-redundant representation for general non-slicing floorplan, in *Proceedings of the international symposium on Physical design*, pages 196–201, 2002.

[16] M. Wang and M. Sarrafzadeh, Modeling and minimization of routing congestion, in *Proceedings on the 2000 conference on Asia and South Pacific design automation*, pages 185–190, ACM Press, 2000.

[17] C. W. Sham and E. F. Y. Young, Routability driven floorplanner with buffer block planning, in *Proceedings of the international symposium on Physical design*, pages 50–55, 2002.

[18] K. K. C. Wong and E. F. Y. Young, Fast buffer planning and congestion optimization in interconnect-driven floorplanning, in *Proceedings of the conference on Asia South Pacific Design Automation Conference*, pages 411–416, 2003.

[19] M. Rebaudengo and M. Reorda, Gallo: A genetic algorithm for floorplan area optimization, in *IEEE Transaction on Computer-Aided Design*, volume 15, pages 943–951, 1996.

[20] D. F. Wong and C. L. Liu, A new algorithm for floorplan design, in *Proceedings of the 23rd ACM/IEEE conference on Design automation*, pages 101–107, ACM Press, 1986.

[21] S. Nakatake, K. Fujiyoshi, H. Murata, and Y. Kajitani, Module placement on bsg-structure and ic layout applications, in *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 484–491, IEEE Computer Society Press, 1996.

[22] P.-N. Guo, C.-K. Cheng, and T. Yoshimura, An o-tree representation of non-slicing floorplan and its applications, in *Proceedings of the 36th ACM/IEEE conference on Design automation conference*, pages 268–273, ACM Press, 1999.

[23] S. Zhou, S. Dong, C.-K. Cheng, and J. Gu, Ecbl: an extended corner block list with solution space including optimum placement, in *Proceedings of the 2001 international symposium on Physical design*, pages 150–155, ACM Press, 2001.

[24] P. Pan and C. L. Liu, Area minimization for floorplans, in *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, volume 14, pages 123–132, January 1995.

[25] F. Y. Young, C. C. N. Chu, W. S. Luk, and Y. C. Wong, Floorplan area minimization using lagrangian relaxation, in *Proceedings of the 2000 international symposium on Physical design*, pages 174–179, ACM Press, 2000.

[26] H. B. Bakoglu, *Circuits, Interconnections and Packageing for VLSI*, Addison-Wesley, 1990.

[27] R. Otten, Global wires harmfuls?, in *Proceedings of the international symposium on Physical design*, pages 104–109, April 1998.

[28] J. Cong and D. Z. Pan, Interconnect delay estimation models for synthesis and design planning, in *Proceedings of the conference on Asia South Pacific Design Automation Conference*, pages 97–100, 1999.

[29] J. Cong, L. He, K. Y. Khoo, C. K. Koh, and D. Z. Pan, Interconnect design for deep submicron ics, in *Proceedings of the IEEE/ACM international conference on Computer-aided design*, pages 478–485, 1997.

[30] H. M. Chen et al., Integrated floorplanning and interconnect planning, in *Proceedings of the IEEE/ACM international conference on Computer-aided design*, pages 354–357, 1999.

[31] M. Wang and M. Sarrafzadeh, On the behavior of congestion minimization during placement, in *Proceedings of the 1999 international symposium on Physical design*, pages 145–150, ACM Press, 1999.

[32] C.-C. Chang, J. Cong, D. Z. Pan, and X. Yuan, Interconnect-driven floorplanning with fast global wiring planning and optimization, in *Proc. SRC Techcon Conference*, 2000.

[33] M. Wang, X. Yang, and M. Sarrafzadeh, Congestion minimization during placement, in *IEEETCAD: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 19, pages 1140–1148, October 2000.

[34] J. Lou, S. Krishnamoorthy, and H. S. Sheng, Estimating routing congestion using probabilistic analysis, in *Proceedings of the international symposium on Physical design*, pages 112–117, 2001.

[35] U. Brenner and A. Rohe, An effective congestion driven placement framework, in *Proceedings of the 2002 international symposium on Physical design*, pages 6–11, ACM Press, 2002.

[36] J. Cong, T. Kong, and D. Z. Pan, Buffer block planning for interconnect-driven floorplanning, in *Proceeding of the 1999 international conference on Computer-aided design*, pages 358–363, 1999.

[37] P. Sarkar, V. Sundararaman, and C.-K. Koh, Routability-driven repeater block planning for interconnect-centric floorplanning, in *Proceedings of the international symposium on Physical design*, pages 186–191, 2000.

[38] X. Tang and D. F. Wong, Planning buffer locations by network flows, in *Proceedings of the international symposium on Physical design*, pages 180–185, 2000.

[39] F. F. Dragan, A. B. Kahng, I. I. M. andoiu, S. Muddu, and A. Zelikovsky, Provably good global buffering using an available buffer block plan, in *Proceedings of the IEEE/ACM international conference on Computer-aided design*, pages 104–109, 2000.

[40] C. J. Alpert, J. Hu, S. S. Sapatnekar, and P. Villarrubia, A practical methodology for early buffer and wire resource allocation, in *Proceedings*

of the 38th conference on Design automation, pages 189–194, ACM Press, 2001.

[41] M. A. Bender and M. Farach-Colton, The lca problem revisited, in *Latin American Theoretical INformatics*, pages 88–94, 2000.

[42] X. Tang and D. F. Wong, Fast-sp: a fast algorithm for block placement based on sequence pair, in *Proceedings of the conference on Asia South Pacific Design Automation Conference*, pages 521–526, ACM Press, 2001.

[43] S. Nakatake, Y. Kubo, and Y. Kajitani, Consistent floorplanning with super hierarchical constraints, in *Proceedings of the 2001 international symposium on Physical design*, pages 144–149, ACM Press, 2001.

[44] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.

[45] C. K. W. J. M. Ho, G. Vijayan, A new approach to the rectilinear steiner tree problem, in *Proceedings of the ACM/IEEE conference on Design automation conference*, 1989.

[46] P. Chong and R. K. Brayton, Estimating and optimizing routing utilization in dsm design, in *Int. Workshop on System Level Interconnect Planning (SLIP)*, April 1999.

[47] C. J. Alpert and A. Devgan, Wire segmenting for improved buffer insertion, in *Proceedings of the ACM/IEEE conference on Design automation conference*, 1997.

[48] J. Cong, A. Kahng, and K. Leung, Efficient algorithm for the minimum shortest path steiner arborescence problem with application to vlsi physical design, in *IEEE Transaction on Computer-Aided Design*, volume 17, pages 24–38, 1998.

[49] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, John Wiley and Sons, 1990.

[50] L. R. Ford and D. R. Fulkerson, *Flows in Network*, Princeton, NJ, 1962.

[51] J. Cong and D. Z. Pan, Interconnect performance estimation models for synthesis and design planning, in *UCLA-CSD-980017*, 1998.

[52] W. C. Elmore, The transient response of damped linear networks with particular regard to wide band amplifiers, in *Journal of Applied Physics*, volume 19, pages 55–63, 1948.

[53] R. Otten, Graphs in floor-plan design, in *International Journal of Circuit Theory and Applications*, volume 16, pages 391–410, Oct. 1988.

[54] P. H. Madden, Partitioning by iteration deletion, in *Proceedings of the international symposium on Physical design*, pages 83–89, 1999.

[55] S. T. W. Lai, E. F. Y. Young, and C. C. N. Chu, A new and efficient congestion evaluation model in floorplanning: Wire density control with twin binary trees, in *Proceedings of Design, Automation and Test in Europe*, pages 856–861, 2003.