2006

An integrated placement and routing approach

Min Pan
Iowa State University

An integrated placement and routing approach

by

Min Pan

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:
Chris Chong-Nuen Chu, Major Professor
Randall Geiger
Morris Chang
Degang Chen
Maria Axenovich

Iowa State University

Ames, Iowa

2006

UMI Number: 3243566

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. GENERAL INTRODUCTION

The semiconductor industry has evolved from the first Integrated Circuits of the early 1970s and experienced a rapid growth since then. The driving force of the VLSI technology has been a constant shrinking of the feature size of VLSI devices. The feature size decreases from about $1\mu m$ in 1990 to $65nm$ in the current advanced technology. Next year, Intel will put $45nm$ technology into mass production in accordance with Moore's law.

The continuous scaling down of the devices has dramatic impact on the VLSI technology in several aspects. First, the device density on integrated circuits grows quadratically as the feature size decreases. The total number of transistors on a single chip has increased from 500K in 1985 to more than a billion transistors today. Second, due to the reduction of device size and increase of chip size, the signal delays due to interconnects have become predominant in today's designs. Third, thanks to the extremely small devices, very complicated systems can be implemented on a single chip. To handle the complexity of modern designs, system on chip (SOC) becomes a common design style today. Large-size pre-designed blocks are used as building blocks for complex systems. All these make the VLSI design, especially the physical design more and more challenging.

In this dissertation, the author collects several works addressing the placement and routing issues in physical design area. A new integrated placement and routing framework is proposed to handle the challenges of large-scale high-quality VLSI designs nowadays.

1.1 VLSI Physical Design

Physical design is a very critical step in the whole VLSI design cycle. It is an art based on the science of establishing interconnections and fulfilling system functions by placing modules and interconnection within a chip or package. In modern designs, the huge number of components and the physics details in fabrication process makes the physical design extremely computationally intensive. The problems in physical design are intractable without the help of computers. As a result, all phases of physical design extensively use Computer Aided Design (CAD) tools, and many phases have already been fully or partially automated.

The physical design algorithms manipulate the geometric objects such as rectangles, polygons and lines to achieve good performance and yield for the design. Hence, those algorithms have close relationship with graph algorithms and combinatorial optimization. Although there are a lot of physical design algorithms, they have some properties in common. First, the objectives of these algorithms are to find optimal arrangement of objects or interconnection scheme try to optimize objectives to achieve the functionality and performance required by the design specification. Second, algorithms for physical design must ensure that the layout generated abide by the design rules so that it can be fabricated correctly. Finally but very importantly, these algorithms must be very efficient to deal with the extremely large size problems brought by current designs in reasonable runtime.

The typical physical design flow includes: Partitioning, Floorplanning, Placement, Routing, Extraction and Verification. Among all these stages, Placement and Routing are the two most important ones.

1.2 Placement and Routing

In modern high-performance designs, up to 75% of the signal delay is due to interconnects. As we know, interconnects are determined mainly by placement (where to put the devices) and routing (how to connect devices by wires). Therefore, placement and routing play key roles to achieve less signal delay, thus high performance.

During the placement, the modules (standard cells and macro blocks) are positioned so that the arrangements of the modules allow completion of interconnections between modules. In addition, the arrangement also need to achieve good timing property.

The objective of routing is to complete the interconnections by wires between modules according to the specified netlist. Normally, it is accomplished in two phases: global routing and detailed routing. In global routing, connections are completed between the proper blocks of the circuit disregarding the exact geometric details of each wire and pin. In other word, global routing specifies the different regions in the routing space through which a wire should be routed. Following the routing paths generated by global routing, detailed routing completes the point-to-point connections between pins considering the design rules.

1.3 Dissertation Overview

1.3.1 Motivation

Placement and Routing are two key steps in the physical design flow. Both of them are very hard problems (NP-hard). Historically, they are divided into two stages to make the problem tractable. Placement first finds positions for every cell, and then routing finishes the interconnections between the fixed cells. Therefore, the routing information is not available during the placement process. Net models such as star-model, HPWL (Half Perimeter Wirelength) are employed to approximate the routing to simplify the placement problem. However, the good placement in terms of these objectives may not be routable at all in the routing stage because different objectives areoptimized in placement and routing stages. This inconsistancy between placement and routing makes the results obtained by the two-step optimization method far from optimal.

Our goal is to integrate placement and routing into the same framework so that the objective optimized in placement is the same as or at least very close to that in routing. Of course, since both placement and routing themselves are very hard problems to handle, we need to have very efficient algorithms so that integrating them together will not lead to intractable complexity. Hence, we develop several efficient and high-quality placement and routing algorithms as the building blocks of the framework. Finally, an integrated placement and routing approach based on these algorithms is proposed to achieve high-quality placement and routing solutions in reasonable runtime.

1.3.2 Components

In Chapter 2, we propose a highly efficient algorithm *FastPlace* 3.0 for large-scale mixed-size placement problem. *FastPlace* 3.0 is a congestion aware multi-level force-directed placement algorithm. It is able to handle the large designs with multi-million standard cells and thousands of big macro blocks. Compared with many state-of-the-art academic placement algorithms, *FastPlace* 3.0 produces competitive results but at a much lesser runtime.

In Chapter 3, we develop an efficient and effective detailed placer - *FastDP*. It improves the placement by moving standard cells to cut down the total wirelength. By applying four major techniques: *Global Wwap*, *Vertical Swap*, *Local Re-ordering* and *Single-segment Clustering*, *FastDP* achieves significant HPWL reduction very efficiently. Experimental results show that *FastDP* outperforms other high-quality academic detailed placers: post-processing in *Fengshui* 5.0 [8], *rowIroning* in the *Capo* package [10], and *Domino* [12] in both wirelength and runtime.

In Chapter 4, we propose a novel performance-driven topology design algorithm. This algorithm focuses on generating good topologies for timing critical nets, especially for those with high fanouts. First, a very fast algorithm constructs an A-tree topology based on table lookup and net-breaking. Then a post-processing technique, not restricted to A-trees anymore, is applied to the obtained A-tree topology to further improve the timing for the net. Experimental results show that our new algorithm can generate topologies with better timing than the timing-driven tree construction in C-tree algorithm [32]. Moreover, our algorithm is 371× faster than C-tree algorithm.

In Chapter 5, we develop an extremely fast global router - *FastRoute*. Different from traditional global routers, it aims at the application of integrating routing into placement. Hence, it has to be fast enough to be integrated in placement process. On the other hand, it needs to generate high-quality routing solutions so that it can be used as real global router. *FastRoute* mainly focuses on determining good Steiner tree topology and Steiner nodes locations according to congestion information. Hence, it alleviates the burden of maze routing which consumes a lot of runtime. This methodology enables both faster runtime and high-quality. Compared with state-of-the-art academic global routers *Labyrinth* [46] and *Chi Dispersion* router [47], *FastRoute* generates better routing solutions and is 132× and 64× faster. It is a dramatic improvement over the previous work.

In Chapter 6, an improved global routing algorithm *FastRoute* 2.0 is proposed to further improve the solution quality over *FastRoute*. It consists of two major techniques: monotonic routing technique and multi-source multi-sink maze routing technique. The monotonic routing can substitute the pattern routing to get better routing solution with similar runtime. The multi-source multi-sink maze routing expands the search space for a good path which results in much less routing congestion. *FastRoute* 2.0 achieves much better solution quality than *FastRoute*, *Labyrinth* and *Chi Dispersion* router. The total overflow is reduced by more than an order of magnitude over *FastRoute*, *Labyrinth* and *Chi Dispersion* router. The runtime is about 73% slower than the extremely fast *FastRoute*, but still 78× and 37× faster than *Labyrinth* and *Chi Dispersion* router.

In Chapter 7, we propose an entirely new placement and routing approach. In this approach, we integrate global routing into placement process to achieve high-quality placement solution and the global routing over it. It is totally different from the traditional sequential placement and routing approach. Having routing information in placement process is always desirable to achieve high-quality placement solutions. However, the major obstacle is the complexity. Since both placement and routing are NP-hard problems, integrating them together using traditional algorithms will become intractable. Benefiting from the very efficient placement and routing algorithms developed in the previous chapters, we are able to integrate the global routing into placement process so that the routing information can direct the generation of placement solution with good routability. The output of this new approach is not only a high-quality placement, but also a global routing solution over it.

CHAPTER 2. HIGHLY EFFICIENT ALGORITHM FOR LARGE-SCALE MIXED-SIZE PLACEMENT PROBLEM

2.1 Introduction

Placement has become a major contributor to the timing closure results of large-scale integrated circuits. The main reason being that placement of circuit modules determines to a large extent interconnect length and hence interconnect delay and routing resource demand.

As semiconductor technology advances into the nanometer regime, the circuit sizes that need to be handled by placement algorithms are steadily increasing to over millions of modules. In addition, placement is often run multiple times during various stages of the physical synthesis flow. Due to these reasons, efficient and scalable placement algorithms are required to produce good quality results in a reasonable amount of time.

Another important constraint that needs to be handled by current placers is that of placement congestion. Placement is typically run in an iterative manner along with timing optimization techniques like buffer insertion and gate sizing. Additionally, it has a major impact on the subsequent routing stage. Hence, placement algorithms should be congestion aware so as to provide space for the subsequent timing optimization and routing stages.

2.2 Key Features of FastPlace 3.0

In this section we briefly describe *FastPlace* 3.0, an efficient congestion aware multilevel force-directed placement algorithm for large-scale mixed-size designs.

The key features of *FastPlace* 3.0 are:

- A *multilevel framework* within the global placement stage to handle large-scale placement circuits. This is achieved by employing a two-level clustering scheme. In the first level, an initial netlist based fine-grain clustering is performed. This is followed by an initial placement of the fine-grain clusters. In the second level, utilizing the initial placement, a combined netlist and physical based coarse-grain clustering is performed.

- A *Hybrid net model* to speed up the quadratic program solver. The Hybrid net model is a combination of the traditional clique and star net models. It results in a substantial decrease in the number of non-zero entries in the connectivity matrix as compared to the clique model thereby resulting in a significant speed-up of the quadratic program solver.

- An efficient *Cell Shifting* technique to spread the modules during the early stages of the placement flow. This technique roughly maintains the relative order of the modules as obtained by solving the quadratic program in both the horizontal and vertical directions.

- An *Iterative Local Refinement* technique to reduce the wirelength based on the half-perimeter measure. This technique is applied once a coarse global placement is obtained and is highly effective in simultaneously reducing the wirelength while spreading the modules. It can also effectively handle placement blockages and placement congestion constraints.

- A robust *macro-block legalization* technique that resolves overlap among the macros by perturbing them by the minimum possible distance from their global placement positions. For any representation specifying the relative positions of the macros, it uses an optimal *Iterative Clustering Algorithm* to place the macros with minimum perturbation from their global placement positions.

- An efficient and robust *standard-cell legalization* technique that operates on the segments created in the placement region due to the presence of placement blockages. This technique satisfies segment capacities and legalizes the standard-cells within the segments.

- A fast and effective *detailed placement algorithm* that can work on both row-based standard-cell placement and placement in the presence of fixed macros.

2.3 Overview of the Algorithm

The multilevel global placement framework used within *FastPlace* 3.0 is summarized in Figure 2.1. It follows the classical hierarchical flow that has been used in many existing placement algorithms [1, 2, 3, 4, 5].

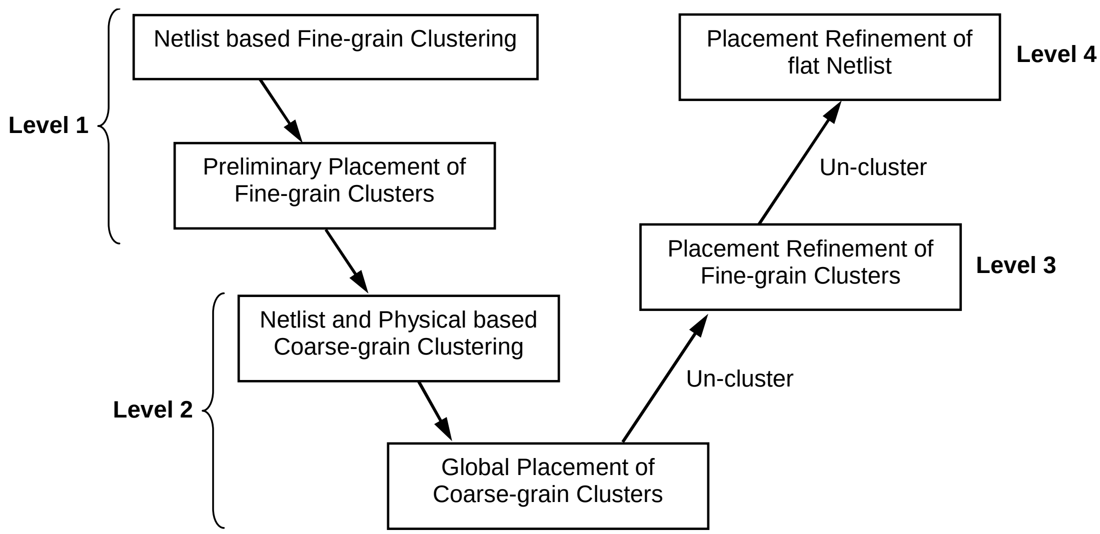

Figure 2.1 Multilevel Global Placement Framework

In Level 1 of the multilevel flow, we create fine-grain clusters using a netlist based connectivity score and perform a fast initial placement of the fine-grain clusters. In Level 2 we perform a second level of clustering in which we use a netlist and physical based clustering score to generate coarse-grain clusters. We then perform global placement on the coarse-grain clustered netlist until the clusters are evenly distributed over the placement region. Since the number of modules at this level are significantly less as compared to the original flat netlist, this step is quite fast and greatly contributes to the overall efficiency of the placement algorithm. After the placement of the coarse-grain clusters, we perform a series of un-clustering and placement refinements in Levels 3 and 4, finally yielding a global placement solution of the original flat netlist.

The entire flow of our placement algorithm is summarized in Figure 2.2. It is divided into three stages: (1) congestion aware global placement using a multilevel framework, (2) legalization of macro-blocks and standard-cells and (3) detailed placement for further improvement.

Experimental results over a broad range of benchmarks show that *FastPlace* 3.0 produces competitive placement results as compared to other state-of-the-art academic placers but at a much lesser runtime. Such an ultra-fast placer is very much needed in present day iterative physical synthesis flows to achieve timing closure without a significant runtime overhead.

Stage 1: Global Placement

Level 1: Initial Placement

1. Construct fine-grain clusters using netlist based clustering
2. Solve initial quadratic program
3. Repeat
   a. perform *regular Iterative Local Refinement* on fine-grain clusters
4. Until the placement is roughly even

Level 2: Coarse Global Placement

5. Construct coarse-grain clusters using netlist and physical based clustering
6. Repeat
   a. Solve the convex quadratic program
   b. Perform *cell-shifting* on coarse-grain clusters and add spreading force
7. Until the placement is roughly even
8. Repeat
   a. Perform *density-based Iterative Local Refinement* on coarse-grain clusters
   b. Perform *regular Iterative Local Refinement* on coarse-grain clusters
9. Until the placement is even

Level 3: Refinement of fine-grain clusters

10. Un-cluster coarse-grain clusters
11. Perform *density-based Iterative Local Refinement* on find-grain clusters
12. Perform *regular Iterative Local Refinement* on fine-grain clusters

Level 4: Refinement of flat netlist

13. Un-cluster fine-grain clusters
14. Perform *density-based Iterative Local Refinement* on flat netslist
15. Perform *regular Iterative Local Refinement* on flat netlist

Stage 2: Legalization

16. Legalize and fix movable macro-blocks using *Iterative Clustering Algorithm*
17. Move standard-cells among segments to satisfy segment capacities
18. Legalize standard-cells within segments

Stage 3: Detailed Placement

Figure 2.2 Outline of the FastPlace 3.0 Algorithm

CHAPTER 3. FASTDP - AN EFFICIENT AND EFFECTIVE DETAILED PLACER

3.1 Introduction

As we mentioned in Chapter 2, placement has become a critical stage in the current physical synthesis flow. High-quality and efficient placement algorithms are in great demand.

Traditionally, placement is separated into two stages, *global* and *detailed* placement. The main purpose of global placement is to distribute the cells evenly over the placement region and optimize certain objectives such as wirelength. As we want to maintain a global view, some approximation has to be made to simplify the problem. Also, the global placement pays more attention to the relative positions among cells globally. Hence, it neglects some local problems. Detailed placement works on the legalized placement to further improve the solution quality. It is more constrained than global placement as it optimizes the objectives by transforming one legal placement solution into another. Because of this nature, more accurate models such as half-perimeter wirelength are used in detailed placement.

Previous literature has mainly focused on the problem of global placement. These algorithms apply various approaches including analytical placement [3, 13, 15, 18, 20, 22, 26], simulated annealing [24, 29], and partitioning / clustering [1, 8, 10]. Recently, there have been significant improvements in terms of both solution quality and runtime. On a set of IBM benchmarks, [3, 20] reported very good wirelength and *FastPlace* [26] achieved runtimes many times faster than other state-of-the-art placement algorithms.

However, compared to global placement, there has been much less work in terms of detailed placement. [7, 9, 11] employed a window-based branch-and-bound method for detailed placement. Alternatively, *Dragon* [29] used a greedy cell exchange algorithm. *Domino* [12] transformed the placement problem into a transportation problem that was solved using a network flow algorithm. Kahng et al. [19] employed combinatorial techniques to perform legalization and detailed placement based on several different objectives. In [21], the single-row problem was solved optimally using a dynamic programming approach. In [17], Hur and Lillis proposed a technique called optimal interleaving and also incorporated the dynamic clustering technique [16].

Current detailed placement techniques are either not very effective or too slow. The window-based technique is very local if the window size is small. If a big window is used, the runtime is not affordable. *Domino* is considered a very good detailed placer but it consumes a lot of runtime. In [27], it was observed that *Domino* can achieve an average wirelength reduction of 5.9% over *FastPlace* on the IBM benchmarks. Hence, we believe that significant improvements in terms of wirelength reduction can be made at the detailed placement stage. It was also observed that the *FastPlace+Domino* flow was on average 7.6× slower than *FastPlace*. Considering that current global placers can generate high-quality solutions in a very short time, it is necessary to have efficient detailed placers to further improve the solution quality of global placement.

In this chapter, we present an efficient and effective detailed placement algorithm - *FastDP* - that can work on both row-based standard cell placement and placement in the presence of fixed macros. The main contributions of this work are:

- An efficient *Global Swap* technique to identify a good pair of cells to swap globally based on their optimal positions while all other cells are fixed.
- A *Vertical Swap* technique that swaps a cell with a nearby cell in the segment above or below so as to move it in the direction of its optimal position.
- A *Local Re-ordering* technique that re-orders consecutive standard cells locally to reduce the wirelength.
- A *Single-Segment Clustering* technique that places standard cells optimally within a segment. It solves the same problem as the Single-Row Problem in [21]. Compared with

the dynamic programming method of [21], this technique can get the optimal solution much faster.

We compare *FastDP* with three state-of-the-art academic detailed placers: postprocessing in *Fengshui5.0* [8], *rowIroning* from the *Capo9.1* package [10] and *Domino* [12] on two benchmark suites: IBM Standard-Cell benchmark suite [6, 26] and ISPD05 benchmark suite [23]. On the IBM benchmarks, on global placements generated by *mPL5* [3] and legalized by the *Placement Utility* from the *Capo9.1* package, *FastDP* can achieve 19.0%, 13.2% and 0.5% more wirelength reduction compared to *Fengshui5.0*, *rowIroning* and *Domino* respectively. Correspondingly we are 3.6×, 2.8× and 15× faster. On the ISPD05 benchmarks, we achieve 8.1% and 9.1% more wirelength reduction compared to *Fengshui5.0* and *rowIroning* respectively. Correspondingly we are 3.1× and 2.3× faster.

The rest of this chapter is organized as follows: Section 3.2 provides an overview of *FastDP* algorithm. Section 3.3 describes the techniques used in *FastDP*. Finally, the experimental results and discussions are presented in Section 3.4.

3.2 Overview

FastDP work on a legalized placement. The placement can be a legalized row-based standard cell placement or a legalized placement with all macros fixed. For standard cell placement, the placeable segments are the rows specified in the placement region. For the placement with macros, the whole placement region is divided into placeable segments based on the macros and placement blockages. In both cases, *FastDP* works on the standard cells in the placeable segments to improve the wirelength.

FastDP consists of four key techniques: *Global Swap*, *Vertical Swap*, *Local Re-ordering* and *Single-Segment Clustering*. *Global Swap* is the technique that gives us the most benefit. For any cell i, it tries to identify a good swap pair, so that i after the swap would be in the position that gives the best wirelength when all other cells are fixed. Because the target position can be close to or far from the current position of i, this technique moves a cell globally to reduce the wirelength. *Vertical Swap* tries to swap a cell i with another nearby cell in the segment

FastDP Algorithm

Perform *Single-Segment Clustering*
Repeat
    Perform *Global Swap*
    Perform *Vertical Swap*
    Perform *Local Re-ordering*
Until no significant improvement in wirelength
Repeat
    Perform *Single-Segment Clustering*
Until no significant improvement in wirelength

Figure 3.1 FastDP Algorithm Flow

above or below so as to move i towards its best position. Although this technique is similar to *Global Swap*, it is more local and faster. It tries to fix some local problems in the vertical direction. In the horizontal direction, we employ a *Local Re-ordering* technique to find a better order for consecutive standard cells within segments. Finally, a *Single-Segment Clustering* technique is developed to optimally place the standard cells within a segment while cells in all other segments are fixed. A near-optimal implementation based on this technique has the time complexity linear to the number of cells in a segment.

The flow of *FastDP* algorithm is summarized in Figure 3.1. We first apply the *Single-Segment Clustering* technique to obtain a relatively good starting solution for the main steps of the algorithm. In the main loop, *Global Swap*, *Vertical Swap* and *Local Re-ordering* are employed to reduce the wirelength until there is no significant improvement. Finally, we re-apply the clustering to get better positions for the cells within the segments without changing their order.

3.3 Detailed Placement Techniques

In this section, we describe the techniques used in *FastDP*.

3.3.1 Global Swap

The basic idea behind *Global Swap* is to find the “optimal region” for a cell i in the placement region and swap i with a cell j or a space s in the “optimal region”. We define the “optimal region” and describe the method to find it in Section 3.3.1.1. In Section 3.3.1.2 we discuss the penalty charged for any overlap created during a swap. Finally, in Section 3.3.1.3 we describe swapping based on the “optimal region” and the penalty for overlap.

3.3.1.1 Optimal Region

Given all other cells in the circuit are fixed, the “optimal region” for a cell i is defined as the region to place i where the wirelength is optimal. This region is determined based on the median idea of [14].

For any cell i, we traverse all the nets connecting to it (noted as N_i) and find their bounding boxes. Here, cell i is excluded from the nets when computing their bounding boxes. For each net $p \in N_i$, we find its bounding box ($x_l[p]$, $x_r[p]$, $y_l[p]$, $y_u[p]$ - the left, right, lower and upper boundaries). From [14], the optimal position for i is given by (x_{opt}, y_{opt}), where x_{opt} and y_{opt} are the medians of the x series ($x_l[1]$, $x_r[1]$, $x_l[2]$, $x_r[2]$, ...) and y series ($y_l[1]$, $y_u[1]$, $y_l[2]$, $y_u[2]$, ...) of bounding boxes. In general, the optimal position is a region rather than a point as the total number of elements in the x and y series are even. This region is the “optimal region” for cell i. In some cases, the “optimal region” can degrade to a point or a line when the two medians of the x and/or the y series carry the same value. Figure 3.2 shows the optimal region for cell 1. There are three nets connecting to cell 1 ($Net1$, $Net2$ and $Net3$). The nets are denoted by closed dashed lines: $Net1$ includes cells 1, 2, 3 and 4; $Net2$ includes cells 1, 5 and 6; $Net3$ includes cells 1, 7 and 8. The bold boundary boxes are the bounding boxes for the nets excluding cell 1. The light lines are the grids constructed by the x series ($x_l[1]$, $x_r[1]$, $x_l[2]$, $x_r[2]$, $x_l[3]$, $x_r[3]$) and y series ($y_l[1]$, $y_u[1]$, $y_l[2]$, $y_u[2]$, $y_l[3]$, $y_u[3]$). The shadowed region is the optimal region for cell 1.

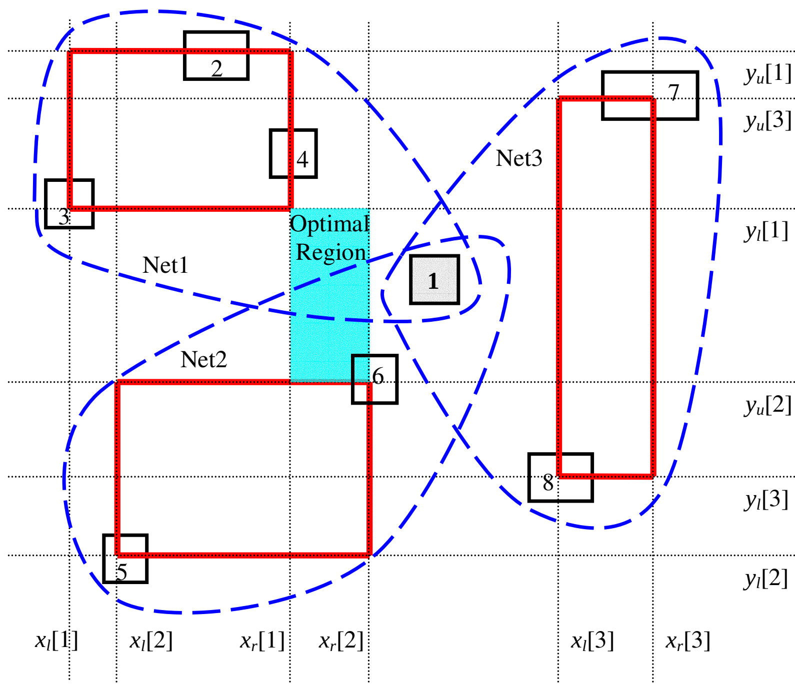

Figure 3.2 Optimal Region

3.3.1.2 Penalty on Overlap

For a cell i, although we find its optimal region, it may not be possible to move it into the optimal region. The reason being that since the detailed placer transforms one legalized placement to another, it is not allowed to have any overlap among cells. Therefore, we need to consider the effect of any resulting overlap among cells when swapping or moving cell i. If a swap causes an overlap, a consequent legalization has to be done to resolve it. Therefore, we need to have a method to model the overlap and consider it when we try to make a swap. We now discuss the method to add a penalty on a swap when it creates an overlap.

If we swap two cells that are not of the same size, the space at the smaller cell may not be enough to hold the bigger cell. Also, If we swap a cell with a space, the space may be smaller than the cell. Both cases may lead to an overlap after swapping. To resolve this overlap, the cells in the segment need to be shifted. We introduce a penalty on this shifting effect. In addition, if the total width of the cells in a segment after swapping is greater than the segment width, we just neglect the swap.

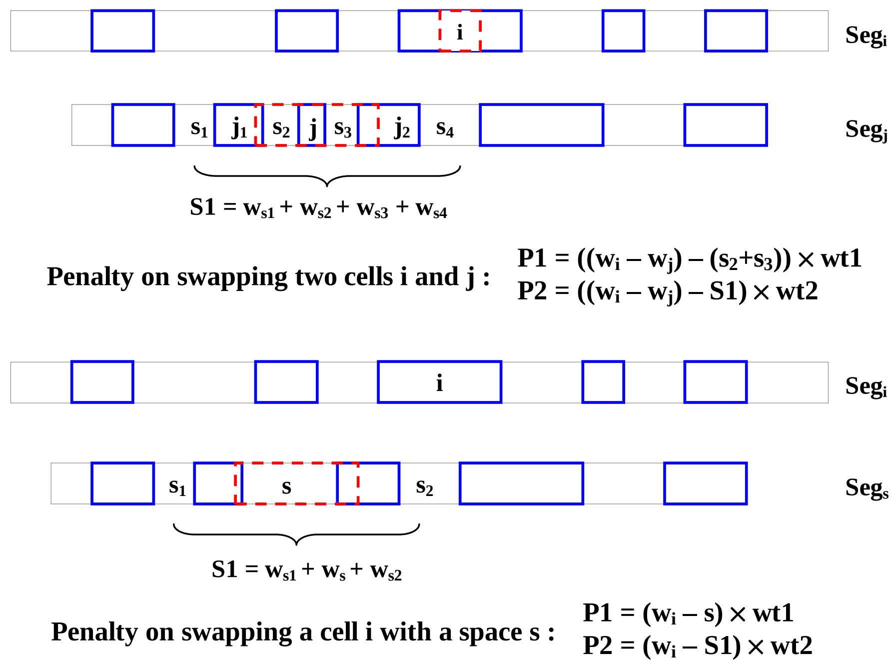

Figure 3.3 Penalty for swapping two cells with different sizes and swapping a cell with a space

For swapping two cells, if there is no overlap after the swap, no penalty is applied; otherwise, a penalty is charged. For swapping a cell with a space, if the space is equal to or bigger than the cell size, no penalty is applied. Otherwise, a penalty is charged. In order to characterize the penalty more accurately, we have two types of penalties: $P1$ and $P2$. $P1$ is the penalty on shifting the closest two cells to resolve overlap. $P2$ is the penalty on shifting cells other than the closest two cells. Figure 3.3 illustrates an example to compute $P1$ and $P2$. The bold boxes are the cells and the light boxes are segments. The dotted lines show the positions after swap for the cells swapped. Consider the case we swap cell i (width w_i) in segment seg_i with another cell j (width w_j) in segment seg_j that is in the optimal region of i. Assume the size of i is larger than j. The two cells left and right to j are j_1 and j_2. The two closest spaces left to j are s_1 and s_2, and the two closest spaces right to j are s_3 and s_4. The total width of spaces s_1, s_2, s_3, s_4 is S1. $P1$ is the wirelength increase caused by shifting j_1 and j_2. If $S1 \geq (w_i - w_j)$, the total shift of j_1 and j_2 to resolve overlap is $(w_i - w_j) - (s_2 + s_3)$. We make $P1$ proportional to this shift. If $S1 \leq (w_i - w_j)$, only shifting j_1 and j_2 cannot resolve the overlap and we need to shift more cells in seg_j. $P2$ is the penalty of shifting cells other than j_1 and j_2 in seg_j. In this case $P2$ is proportional to the shift on cells other than j_1 and j_2, which is $(w_i - w_j) - S1$. Hence, we set $P1$ and $P2$ as follows:

$$P1 = ((w_i - w_j) - (s_2 + s_3)) \times wt1$$
$$P2 = ((w_i - w_j) - S1) \times wt2 \tag{3.1}$$

where $wt1$ and $wt2$ are the two weights on the shift. For the case where we swap i with a space s, the way to get the penalty is similar to that for swapping two cells. The only difference is that the width difference is $w_i - 0 = w_i$ and $S1$ is the sum of the widths of s, the closest space left to s and the closest space right to s.

Since the shifts in $P1$ and $P2$ have the dimension of length, the two weights $wt1$ and $wt2$ are just constants with no dimension. Because we do not want to disturb the original placement too much, large overlap is discouraged by setting $wt2$ much higher than $wt1$.

3.3.1.3 Global Swap Based on Optimal Region

Based on the optimal region and the penalty on overlap, we develop a *Global Swap* technique to swap each cell with a cell or space in its optimal region. Since there could be several cells and spaces in the optimal region, we have many choices. We use a term "benefit" B as a measure for selecting the cell or space in the optimal region. The "benefit" for a swap has two components: one is the the difference between the total wirelength before and after the swap, the other is the penalty charged on the created overlap. If the wirelength before and after the swap are W_1 and W_2, respectively, the "benefit" can be obtained by equation (3.2).

$$B = (W_1 - W_2) - P1 - P2 \tag{3.2}$$

If $B > 0$, it means that we will benefit from the swap. Otherwise, the resulting placement is worse than original. Of course, the "benefit" we compute is not accurate because the real wirelength change due to resolving the overlap is hard to measure. We only use a simple penalty on shifting cells to model this wirelength change. Based on the "benefit", we do the swapping as follows. For each standard cell i, we find its optimal region and try to swap it with every cell j and space s in the optimal region of i. We measure the "benefit" for each swap and pick the j or s with the best "benefit" to perform the swap. If the best "benefit" has a value less than zero we do not make a swap as it would increase the wirelength.

In this technique, we look at the optimal region for a cell to find a good target position. The optimal region can be close to or far from the current position. Hence, compared to the traditional window-based branch-and-bound methods, our *Global Swap* technique has a more global view when repairing the positions of cells. In Table 3.1, we show the distribution of the cells according to the distance of the cells from their respective optimal regions before and after 1 iteration of *Global Swap* for the circuit ibm01. The unit of the distance is the standard row height. Distance 0 means the cell is in its optimal region. It is clear that our technique is very effective in moving cells towards their optimal region.

Table 3.1 Distribution of cells based on the distance from their optimal regions before and after 1 iteration of Global Swap

Distance	0	(0,1]	(1,2]	(2,3]	(3,4]	> 4
before	30.0%	36.8%	18.0%	6.4%	3.4%	5.4%
after	33.0%	39.3%	17.1%	5.3%	2.2%	3.1%

In the actual implementation, to save runtime, for a selected cell, we do not pick the cell with the best "benefit" in its optimal region. Instead, we pick the first "good" cell that can give us certain "benefit". Another issue is that after swapping two cells with different sizes, the placement is no longer legal. Overlaps are created around the bigger cell and spaces are created around the smaller cell. We need to re-legalize the segments containing the two cells. However, legalization after every swap will be very time consuming. In the implementation, we legalize the whole placement after all the segments has been traversed. Of course, we will lose some accuracy on the positions of cells, but experiments show that this inaccuracy does not affect the final wirelength significantly.

3.3.2 Vertical Swap

In *Global Swap* technique, for a cell i, we may not find a good candidate cell or space in its optimal region to swap with it. There could be two reasons for this. First, the size of i is large and the optimal region of i is congested. Hence, the segments that span the optimal region cannot hold i. Second, in order to hold i, many cells have to be shifted to legalize the placement which introduces a high penalty.

To increase the possibility for a good swap and reduce the vertical wirelength locally, we have a *Vertical Swap* technique very similar to the *Global Swap*. The idea of *Vertical Swap* is to move a cell vertically toward its optimal region. This technique is not as greedy as *Global Swap*. Every time it only moves a cell up or down by one row. For a cell i, if the optimal region is above / below the current position, a few nearby cells above / below i are considered to be candidates. We use the same penalty as in *Global Swap* to estimate the effect of overlap and pick the best candidate to swap with i. We observe that if we interleave the *Vertical Swap* with *Global Swap*, the wirelength decrease is faster than only applying *Global Swap*. We believe that this is because the *Vertical Swap* is not very greedy and has more flexibility in moving the cells. At the same time, it may increase the possibility for *Global Swap*. In addition, this technique is much faster than *Global Swap* because for each cell, the number of candidate cells considered for swap are much less than in *Global Swap*.

3.3.3 Local Re-ordering

With *Vertical Swap* fixing local vertical errors, we need a technique to fix local horizontal errors. Although *Global Swap* can also fix horizontal errors, it is quite expensive to use it to fix local problems. Therefore, we propose a very fast *Local Re-ordering* technique to handle this problem. For any n consecutive cells within a segment, we try all possible left-right ordering of cells and pick the order giving the best wirelength. In this technique, we also need to decide the position of the cells in each order. To speed-up the technique, we consider the cells as a group and make the left boundary of the group as the left boundary of the first cell in the original order and the right boundary of the group as the right boundary of the last cell in the original order. Then for each order, we keep the left and right boundaries of the group and evenly distribute the cells inside the group. Since we have the *Single-Segment Clustering* technique to take care of the cell positions, we do not pay much attention to the exact positions of the cells during *Local Re-ordering*.

In *FastDP* implementation, we set $n = 3$. The reason is that $n = 2$ means pairwise swapping and it is too constrained. But if we choose $n = 4$, it will be 4 times slower and the improvement is not so significant. Compared to the conventional window-based technique, *Local Re-ordering* has a 3-cell window in one row and is very local. But since we have the *Global Swap* technique, it is only used to efficiently fix local errors.

3.3.4 Single-Segment Clustering

After the main loop of *FastDP* we fix the segments and the ordering within the segments for all the standard cells. We now want to further reduce the wirelength by moving the cells inside the segments. For a legalized placement, if we fix the order of the cells in one segment and the positions of the cells in all other segments, the problem becomes a fixed-order single segment problem described below.

Fixed-Order Single Segment Placement Problem: Given a segment S in the placement region with n standard cells $C1, C2, ..., Cn$, whose left-to-right order is fixed (C_i is left to C_j if $i < j$). All cells not in S are fixed. Find a non-overlapping placement for the segment S so that the total half-perimeter wirelength is minimized.

This problem is basically the same as the Single-Row Problem in [21]. In [21], the authors proposed a dynamic programming algorithm to solve the problem optimally. In the following part, we describe a more efficient algorithm that can also solve the problem optimally.

First, we define some terms used in our algorithm. A *cluster* is a standard cell or a group of standard cells abutted together (retaining the original order of standard cells). *Clustering* is the operation to abut two clusters to form a new cluster (the width of the new cluster is the sum of the widths of the original clusters). The wirelength function of x-coordinate of a cluster is a convex piecewise linear function W(x) when all other objects are fixed. The slopes for the linear pieces are ..., $-3, -2, -1, 0, 1, 2, 3$, ... The slope 0 part is the optimal region in x-direction for the cluster. The points where the function changes slope are called *bounds*. These bounds are the left and right boundaries of the bounding boxes for the nets connecting to the cluster. *Optimal Region Center* of a cluster is the middle point of the optimal region in x-direction when all the objects (standard cells and macro blocks) not part of the cluster are fixed.

In order to find the optimal region for a cluster C in segment S, we need to fix the positions for all the other objects. But the standard cells in S are not fixed. Therefore, if C has connections to any standard cells in S, the bounds for C cannot be determined. However, since we fix the order of the standard cells in S, we know the left-right orders between the cells. We use this information to get the bounds so that the optimality of the solution will not be affected. The method to get the bounds is as follows. When computing the bounding box for any net N connecting to C, if N is connecting to a standard cell C' in S, we will assume C' at the end of the segment S, i.e., if C' is left to C, we assume C' is at the left end of segment S; otherwise, C' is at the right end of segment S. Although we are not using the real position for C', we will not affect the optimality of the position of C because the left-right order of C and C' has to be maintained. The main idea of the algorithm is to put every cluster at its *Optimal Region Center*. If there is overlap between two clusters, we perform clustering and form a new cluster. The new cluster will not be broken at any later stage. Then we put the new cluster at its *Optimal Region Center*. We iteratively perform clustering until all the cells are put at *Optimal Region Center* without any overlap. If any optimal region boundary is out of the segment range, we will assign it at the closest boundary. In this way, no cell will be put out of the segment. The pseudo-code of the *Single-Segment Clustering* Algorithm is given in Figure 3.4.

Theorem 1 The *Single-Segment Clustering* Algorithm finds the optimal solution for the *Fixed-Order Single Segment Placement Problem*.

Proof It is not hard to see that if the clusters are formed correctly, then the solution obtained by our algorithm is optimal. To show that we will not form wrong clusters, assume on the contrary that the clustering in the optimal solution is different from our solution.

Single-Segment Clustering Algorithm

num_old_cluster ← *n*

Initialize *old_cluster[i]* as standard cell C_i, *i*=1, 2, …, *num_old_cluster*.

do

Find the bounds list and the Optimal Region Center *Xic* for K_i,

and set *X(old_cluster[i])* = *Xic*

newcount ← 1 // the count for the number of new clusters

new_cluster[1] ← *old_cluster[1]* // initialize the first new cluster

j ← 1

while(*j* < *num_old_cluster*)

do

if *new_cluster[newcount]* and *old_cluster[j+1]* has overlap

Cluster *new_cluster[newcount]* and *old_cluster[j+1]* to form the

new *new_cluster[newcount]*

Merge the bounds list for *new_cluster[newcount]* and *old_cluster[j+1]*

to get the new bounds list for *new_cluster[newcount]*

Find the Optimal Region Center *Xc* for *new_cluster[newcount]*

based on the new bounds list

X(new_cluster[newcount]) ← *Xc*

else

newcount ← *newcount* + 1 //begin a new cluster *new_cluster[newcount+1]*

j ← j+1

num_old_cluster ← *newcount*

old_cluster[i] ← *new_cluster[i]* (*i*=1, …, *newcount*)

until no overlap among *old_cluster[i]*, (*i*=1, …, *num_old_cluster*)

Assign the C_i (*i*=1, 2, …, *n*) to the positions according to the positions of the *old_cluster[j]* (*j*=1, 2, …, *num_old_cluster*) they belong to

Figure 3.4 Single-Segment Clustering Algorithm

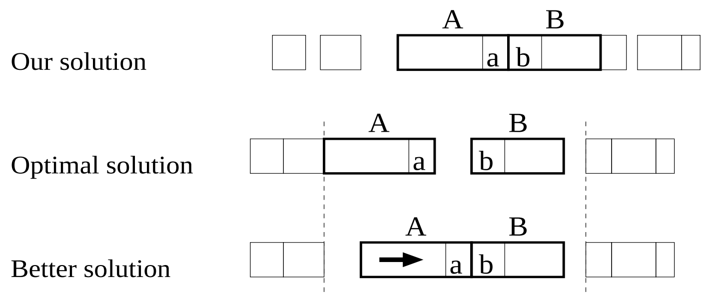

Figure 3.5 Proof of optimality of Single-Segment Clustering Algorithm

Consider a gap in the optimal solution surrounded by a pair of cells a and b that are in the same cluster in our solution. Suppose a and b are clustered together when we merge clusters A and B in some step of our algorithm. See Fig. 3.5 for an illustration. Without loss of generality, we can assume there is no gap within cluster A and within cluster B in the optimal solution. Otherwise, we can consider the gap within cluster A or cluster B instead. Since we merge cluster A and cluster B together at some point, A and B cannot be at the optimal region at the same time if their order is not changed. For any solution, either A wants to move left or B wants to move right (or both) to reduce the wirelength. We can always generate a better wirelength than the optimal solution by moving either A or B towards the gap without creating any overlap. This is a contradiction. Thus, our solution should be optimal. ∎

We now analyze the complexity of the algorithm. There are n cells in total, and the maximum number of clustering is $n-1$. In the clustering operation, every step needs constant time except merging the two bounds lists. The merge takes linear time to the number of bounds m. The complexity of the algorithm is $O(nm)$. However, in practice, it can be much better. In our implementation, we are not keeping all the bounds for the clusters. Instead, we only keep a constant number of bounds for every cluster. Therefore, the merge also takes constant time. The total complexity of the algorithm is $O(n)$. Of course, it will compromise the optimality, but experiments show that even using a small constant will not degrade the solution appreciably. In implementation, the constant we choose is 16. Table 3.2 shows different results when using different constants on the 8^{th} segment of ibm01. It shows that even if a small constant is used, the result can be very close to optimal. Moreover, the segment we choose here is one that has a lot of room to reduce the wirelength. For the 1^{st} to 7^{th} segment in ibm01, just using 8 bounds can achieve the optimal solution.

Table 3.2 The relationship between # bounds and wirelength decrease on the 8th segment of ibm01

#bounds	4	8	12	16	20	opt
WL dec	13600	14060	14210	14377	14425	14425

Although this algorithm can give the optimal solution for a segment, we still need to run it iteratively as it is only optimal when all cells not in the current segment are fixed. Since we change the cell positions segment by segment, we need to run several iterations to find good positions for the cells.

3.4 Experimental Results and Discussion

We consider the ISPD04 IBM Standard-Cell Benchmark suite [6, 26] and the ISPD05 Benchmark suite [23] for our experiments. The placement tools considered are *FastDP*, post-processing in *Fengshui 5.0* [8], *rowIroning* in the *Capo9.1* package [10], and *Domino* [12]. For post-processing in *Fengshui 5.0*, we use the default control string used in the complete flow of *Fengshui 5.0* which is, -reorder "r,4,4:r,4,2:r,4,2:r,4,1:r,4,1:r,4,1". For *rowIroning*, we use the default options on "-ironPasses -ironWindow -ironOverlap -ironTwoDim" used in the complete *Capo9.1* flow.

We run *mPL5* and *Capo9.1* to get the global placements for both IBM and ISPD05 benchmarks. Since we have to disable both the legalizer and detailed placer in *mPL5*, the global placements created by *mPL5* are not legalized. We therefore use the Placement Utilities in the *Capo9.1* package to legalize the *mPL5* global placements. For *Capo9.1*, we disable the greedy swapping and *rowIroning* in the overall flow to get the legalized global placements. All the results are generated on a Linux machine with Intel Pentium 4, 3.00GHz CPU and 2GB memory.

The half-perimeter wirelength and runtime results on IBM benchmarks for *Fengshui5.0*, *rowIroning*, *Domino* and *FastDP* are reported in Tables 3.3 and 3.4. Table 3.3 gives the results for different detailed placers on the global placements generated by *mPL5*+Legalizer. On average *FastDP* gives 19.05% better wirelength with a 3.62× speed-up over *Fengshui5.0*. Compared with *rowIroning*, we are 13.22% better in wirelength and 2.79× faster. Compared with *Domino*, we can achieve 0.54% better wirelength and are around 15× faster. In addition, on average, we can reduce the wirelength of the legalized placement by nearly 30%. This shows that there is a lot of room for the detailed placer to improve the global placement solution. From Table 3.4, for the *Capo9.1* global placements, *FastDP* is 1.17% better than *Fengshui 5.0* in wirelength with a 4.48× speed-up. We are also 1.91% better than *rowIroning* in wirelength and 5.66× faster. Compared with *Domino*, we are 0.55% better in wirelength and 13.45× faster.

Tables 3.5 and 3.6 show the comparison results on the recent ISPD05 benchmarks. This benchmark set has fixed/movable macros with a large number of cells. For bigblue4 we were unable to obtain the global placement solution of *Capo9.1* as the placer ran out of memory on our machine. Also, we were unable to generate feasible solutions using *Domino* on this set of benchmarks. Hence, only *Fengshui 5.0* and *rowIroning* are used for comparison. Table 3.5 gives the results for different detailed placers on the global placements generated by mPL5+Legalizer. On average *FastDP* achieves 8.06% better wirelength with a 3.05× speed-up over *Fengshui5.0*. Compared with *rowIroning*, we are 9.12% better in wirelength and 2.29× faster. From Table 3.6, on the *Capo9.1* global placements, *FastDP* is 2.04% better than *Fengshui 5.0* in wirelength with a 2.81× speed-up. We are also 0.89% better than *rowIroning* in wirelength and 2.18× faster.

From the comparisons made in Tables 3.3–3.6, *FastDP* can achieve better solution quality in much less runtime as compared to other detailed placers. For the ISPD05 benchmarks, *FastDP* has lesser speed-up over *Fengshui5.0* and *rowIroning* because it runs for more iterations to reach the stopping criterion, whereas *Fengshui5.0* and *rowIroning* have fixed number of passes to run the algorithms. On the global placements generated by *Capo9.1*, all the detailed placers get lesser improvement than on the global placements generated by *mPL5*+Legalizer. A possible reason could be that *Capo9.1* has done many local optimizations during partitioning at the lowest level. Therefore, most of the local errors have been fixed. Another interesting observation is that although the wirelengths of the global placements generated by *mPL5*+Legalizer are much higher than that generated by *Capo9.1*, the final results obtained on the *mPL5*+Legalizer global placements are better than that obtained on Capo global placements. The reason may be because the legalizer disturbs the global placements of *mPL5* by a significant amount, but most of these errors can be fixed by the detailed placer.

Table 3.3 Comparison of Detailed Placers on mPL5+Legalizer Global Placement on IBM benchmarks

	mPL+LG	FastDP			Fengshui5.0		RowIroning		Domino	
	WL(1e6)	WL(1e6)	Impv	runtime(s)	Impv	time/FastDP	Impv	time/FastDP	Impv	time/FastDP
ibm01	2.423	1.732	-28.49%	6	-14.99%	2.81	-18.15%	3.87	-29.79%	14.33
ibm02	4.745	3.701	-21.99%	14	-9.05%	2.51	-13.98%	2.72	-21.97%	9.42
ibm03	6.624	4.792	-27.67%	14	-12.36%	2.70	-17.27%	3.13	-28.16%	10.02
ibm04	8.696	5.893	-32.23%	20	-12.47%	2.32	-18.52%	2.73	-32.51%	14.11
ibm05	12.074	10.106	-16.30%	23	-7.21%	2.45	-10.75%	2.54	-16.94%	10.47
ibm06	7.390	5.335	-27.81%	15	-12.04%	4.67	-16.59%	4.19	-29.09%	24.24
ibm07	11.576	8.380	-27.61%	26	-11.47%	3.78	-17.05%	3.32	-27.46%	16.78
ibm08	12.714	9.361	-26.37%	77	NA	NA	-16.90%	1.28	-26.49%	7.98
ibm09	15.267	9.648	-36.80%	35	-14.40%	3.48	-20.79%	3.03	-36.45%	20.81
ibm10	26.403	17.665	-33.09%	53	-12.22%	3.23	-18.23%	2.73	-32.62%	23.22
ibm11	22.128	14.411	-34.87%	46	-12.95%	4.07	-19.71%	3.13	-34.73%	19.44
ibm12	32.378	22.803	-29.57%	61	-10.61%	2.97	-16.34%	2.53	-28.83%	21.12
ibm13	27.249	17.050	-37.43%	62	-13.31%	4.21	-19.69%	2.94	-36.50%	12.77
ibm14	47.110	32.006	-32.06%	117	-11.34%	5.04	-18.23%	2.56	-31.86%	14.86
ibm15	60.133	39.474	-34.35%	146	-11.21%	5.60	-17.27%	2.50	-34.58%	12.42
ibm16	69.489	43.892	-36.84%	199	-12.13%	4.40	-19.54%	2.05	-36.75%	13.62
ibm17	93.186	62.078	-33.38%	191	-10.91%	4.88	-17.04%	2.28	NA	NA
ibm18	67.687	41.759	-38.30%	484	-11.86%	2.50	-21.11%	0.92	-39.69%	7.03
			-30.84%		**-11.79%**[1]	**3.62**[1]	**-17.62%**	**2.79**	**-30.30%**[2]	**14.86**[2]

1. Average over 17 circuits, Fengshui5.0 failed on ibm08, 2. Average over 17 circuits, Domino failed on ibm17

Table 3.4 Comparison of Detailed Placers on Capo9.1 Global Placement on IBM benchmarks

	CAPO	FastDP			Fengshui5.0		RowIroning		Domino	
	WL(1e6)	WL(1e6)	Impv	runtime(s)	Impv ,	time/FastDP	Impv ,	time/FastDP	Impv ,	time/FastDP
ibm01	1.840	1.788	-2.83%	4	-2.44%	4.74	-0.78%	10.79	-2.68%	15.84
ibm02	3.850	3.715	-3.50%	7	-2.35%	5.11	-1.72%	9.02	-4.64%	15.91
ibm03	5.165	4.977	-3.64%	9	-2.19%	4.88	-1.48%	8.83	-2.84%	11.82
ibm04	6.151	5.938	-3.47%	17	-1.89%	2.98	-1.30%	5.54	-3.57%	7.85
ibm05	10.110	9.822	-2.84%	14	-1.07%	4.47	-0.66%	7.04	-2.85%	11.60
ibm06	5.628	5.415	-3.78%	19	-2.36%	3.89	-1.47%	5.60	-3.69%	19.92
ibm07	9.468	9.208	-2.74%	23	-2.01%	4.85	-1.16%	6.60	-2.18%	18.31
ibm08	9.933	9.582	-3.53%	70	NA	NA	-0.98%	2.36	-2.83%	5.87
ibm09	10.483	10.192	-2.77%	23	-2.31%	6.28	-1.48%	8.18	-1.40%	23.52
ibm10	19.271	18.723	-2.84%	50	-1.64%	3.84	-0.95%	4.95	-1.54%	18.53
ibm11	15.540	15.121	-2.69%	33	-1.92%	6.39	-1.27%	7.37	-1.35%	23.69
ibm12	24.833	24.055	-3.14%	90	-1.44%	2.28	-0.93%	2.97	-2.04%	8.87
ibm13	18.561	18.005	-2.99%	44	-2.07%	6.40	-1.31%	6.90	-1.68%	10.95
ibm14	34.573	33.655	-2.66%	131	-1.62%	4.90	-0.91%	3.85	-2.13%	9.59
ibm15	42.702	41.556	-2.68%	149	-1.63%	5.48	-1.05%	4.10	-2.68%	9.58
ibm16	49.597	48.173	-2.87%	192	-1.55%	4.81	-0.85%	3.53	-2.19%	9.25
ibm17	68.990	67.251	-2.52%	167	-1.37%	2.46	-0.81%	2.37	-1.63%	12.71
ibm18	45.020	43.750	-2.82%	218	-1.66%	2.38	-0.94%	1.84	-2.57%	8.34
			-3.02%		**-1.85%[1]**	**4.48[1]**	**-1.11%**	**5.66**	**-2.47%**	**13.45**

1. Average over 17 circuits, Fengshui5.0 failed on ibm08

Table 3.5 Comparison of Detailed Placers on mPL5+Legalizer Global Placement on ISPD05 benchmarks

	mPL+LG	FastDP			Fengshui5.0		RowIroning	
	WL(1e8)	WL(1e8)	Impv	runtime(s)	Impv	runtime/FastDP	Impv	runtime/FastDP
adaptec1	0.925	0.864	-6.53%	96	-3.63%	3.52	-2.85%	3.95
adaptec2	1.139	1.036	-9.05%	177	-4.99%	2.36	-3.81%	2.56
adaptec3	3.206	2.506	-21.84%	427	-6.83%	2.02	-6.65%	1.99
adaptec4	3.040	2.279	-25.02%	481	-9.45%	2.01	-7.71%	1.66
bigblue1	1.221	1.106	-9.38%	152	-6.45%	3.30	-4.25%	3.43
bigblue2	2.183	1.925	-11.84%	620	-6.93%	2.95	-4.33%	1.55
bigblue3	5.024	4.038	-19.63%	1473	-7.90%	2.99	-8.68%	1.43
bigblue4	10.535	9.230	-12.39%	2273	-5.05%	5.23	-4.46%	1.73
			-14.46%		**-6.40%**	**3.05**	**-5.34%**	**2.29**

Table 3.6 Comparison of Detailed Placers on Capo9.1 Global Placement on ISPD05 benchmarks

	CAPO	FastDP			Fengshui5.0		RowIroning	
	WL(1e8)	WL(1e8)	Impv	runtime	Impv	runtime/FastDP	Impv	runtime/FastDP
adaptec1	0.918	0.906	-1.26%	112	0.56%	2.89	-0.78%	3.44
adaptec2	1.027	1.010	-1.61%	171	0.80%	2.30	-0.80%	2.57
adaptec3	2.538	2.509	-1.16%	399	0.37%	2.93	-0.62%	1.97
adaptec4	2.654	2.637	-0.65%	410	0.51%	3.22	-0.60%	2.06
bigblue1	1.167	1.135	-2.72%	214	0.65%	2.27	-0.88%	2.43
bigblue2	1.813	1.783	-1.68%	985	0.07%	1.95	-1.01%	0.94
bigblue3	4.444	4.270	-3.92%	1051	-1.70%	4.11	-2.11%	1.84
			-1.86%		**0.18%**	**2.81**	**-0.97%**	**2.18**

CHAPTER 4. A NOVEL PERFORMANCE-DRIVEN TOPOLOGY DESIGN ALGORITHM

4.1 Introduction

With technology scaling, interconnect delay has become the dominant factor in circuit delay, making effective performance-driven interconnect design vital for the timing closure. Topology design and buffer insertion are two main techniques for performance-driven interconnect design. Alpert et. al. [32] showed that the two-step approach of (1) constructing a Steiner tree, and (2) then running van Ginneken style buffer insertion, can be as good as the slower simultaneous approach. However, topology design, i.e. finding a good Steiner tree, itself is a difficult and time-consuming step. For nets with low degree[1], such as 2-pin or 3-pin nets, finding good topologies is easy. But for high-degree nets, constructing good topologies efficiently is both challenging as well as critical, for they are likely to be the cause of critical paths.

Rectilinear minimum spanning tree (RMST) is a class of topologies widely used in practice because efficient algorithms are available for their solution. However, the wirelength of an RMST can be as much as 1.5 times that of rectilinear Steiner minimal tree (RSMT) [33]. RSMT is another class of well-researched topologies. But RSMT construction being NP-complete [34], no efficient algorithm exists, and a lot of work has focused on approximation algorithms for it. Batched 1-Steiner heuristic [35] and the heuristic proposed by Mandoiu et. al. [36] are two well-known near-optimal algorithms. Recently, FLUTE [37, 38] has been proposed as a very fast and accurate RSMT algorithm for VLSI applications based on a table lookup technique.

In addition to wirelength-driven topologies such as RMST and RSMT, many timing-driven

[1] The *degree* of a net is the number of pins in the net.

topology design techniques have also been proposed. The SERT algorithm of Boese et. al. [41] produces the routing tree for performance. Later, Cong et. al. [31] proposed A-tree algorithm to find a min-area shortest paths tree. In [42], Permutation-constrained routing trees (P-tree) algorithm reported better area objectives than SERT and A-tree. Alpert et al. [43] proposed AHHK trees as a direct trade-off between Prim's MST algorithm and Dijkstra's shortest path tree algorithm, and used in the C-tree algorithm [32] for timing-driven Steiner tree construction. However, all of these algorithms are not very efficient to address large industrial designs with a substantial number of high-degree nets, especially for an integrated route-and-place flow. Although most of the nets in a design are of low degree, there are still a considerable number of high-degree nets (12% nets have degree ≥ 8 [37]). And these high-degree nets are more likely to be timing-critical. Hence, our goal is to develop a fast, performance-driven topology design algorithm applicable to optimizing delay properties of a large-class of nets early-on in the design cycle.

In this chapter, a novel method is proposed to efficiently design performance-driven topology for nets. First, a very fast algorithm constructs an A-tree topology based on table lookup and net-breaking. Then a post-processing technique, not restricted to A-trees anymore, is applied to the obtained A-tree topology to further improve the timing for the net.

The main contributions of this work include the following:

- A very efficient algorithm to construct the A-tree *potentially optimal wirelength vector* (POWV) [37] and topology table for all the nets with degree up to a certain value
- A fast A-tree construction algorithm using table lookup and net-breaking techniques for high-degree nets
- A performance-driven post-processing technique, which modifies the A-tree topology to further improve the timing

Experimental results show that our new algorithm can generate topologies with better timing than the timing-driven tree construction algorithm in C-tree [32], and RSMT algorithm FLUTE [37, 38]. Moreover, our algorithm is 371× faster than C-tree algorithm. Therefore, it is very suitable for performance-driven topology design for a large number of nets, in an integrated physical design flow.

The rest of the chapter is organized as follows. In Section 4.2, we discuss the topologies for the performance-driven interconnect design. Section 4.3 describes the fast algorithm to generate A-tree lookup table. In Section 4.4, we present the algorithm to construct A-tree using table lookup and net-breaking. In Section 4.5, a performance-driven post-processing technique is proposed. Finally, experimental results are shown in Section 4.6.

4.2 Topology for performance

As mentioned earlier, RSMT is a class of widely used topologies with good wirelength metric. However, an RSMT may contain many detours from the source to some sinks resulting in bad timing for them. A simple illustrative example is shown in Figure 4.1. Sink t_4 is the critical sink here. We can see that there is detour from the source s to t_4 which harms the timing result. Therefore, despite its good wirelength, it is not suitable for performance-driven topology design. And we need some other types of topologies for the timing purpose.

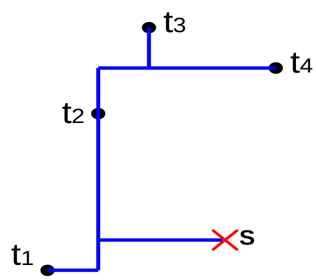

Figure 4.1 Detour in RSMT

A-tree is a class of topologies with good properties for performance-driven interconnect design. First, an A-tree is a shortest path tree (SPT), thus no detours between the source and a sink. In addition, it has been shown in [31] that minimizing total wirelength of an A-tree leads to simultaneous optimization of different components of sink delays. Such a harmony would be impossible to achieve for general routing topologies. Hence we focus on A-trees and their subsequent refinement as our goal. However, finding A-tree with minimum wirelength is an NP-complete problem [40]. Inspired by the table lookup idea of FLUTE [37, 38], we propose a very efficient way to construct A-trees using table lookup techniques to be discussed in detail below.

4.3 A-Tree Lookup Table Generation

In this section, we focus on A-tree lookup table generation. We first discuss how to group infinite number of nets into finite number of groups so that a practical lookup table can be constructed. Then a *Configuration Graph* approach is proposed to generate the lookup table efficiently. Finally, we introduce the concepts of *Abstract Topology* and *Topology Signature* to reduce the complexity in topology table generation.

4.3.1 A-tree Lookup Table Organization

FLUTE [37, 38] is a lookup table based RSMT algorithm. It is shown that the set of all degree d nets can be partitioned into $d!$ groups according to the relative positions of their pins. The relative positions of pins is defined by *vertical sequence*. Consider an d-pin net. Let x_i be the x-coordinate of some vertical Hanan grid line such that $x_1 \leq x_2 \leq ... \leq x_d$. Similarly, let y_j be the y-coordinate of some horizontal Hanan grid line such that $y_1 \leq y_2 \leq ... \leq y_d$. Assume the pins are indexed in ascending order of y-coordinate. Let s_i be the rank of pin i if all pins are sorted in ascending order of x-coordinate. $s_1 s_2 ... s_d$ is the *vertical sequence*. As illustrated in Figure 4.2. All the nets with the same *vertical sequence* fall in one group in the lookup table. For each group, the wirelength of all possibly optimal routing topologies along the Hanan grid [44] can be written as a small number of linear combinations of distances between adjacent Hanan grid lines [37]. Each linear combination can be expressed as a vector of the coefficients which is called a *potentially optimal wirelength vector* (POWV). The few POWVs for each group can be generated once. Each POWV and one corresponding topology are stored into a lookup table. To get the RSMT for a net, the algorithm computes the wirelengths corresponding to the POWVs for the group the net belongs to, and picks the one with the best wirelength.

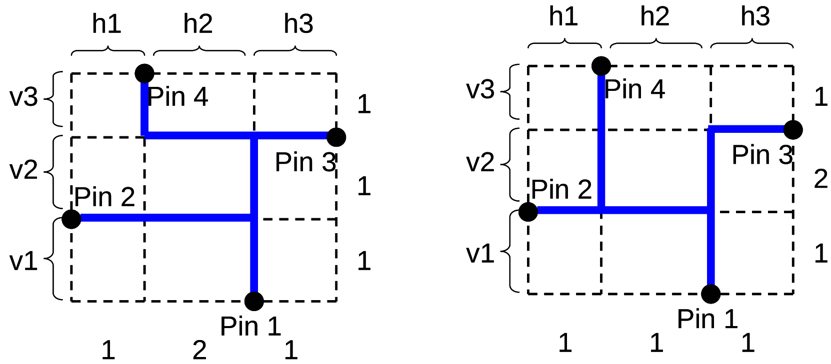

Figure 4.2 Illustration of notions

We also use the *vertical sequence* to group the nets. However, for A-tree, only the *vertical sequence* is not enough to group the nets. The reason is that not only the relative pin positions but also the source pin location define the group of nets sharing the same POWVs (*potentially optimal wirelength vectors*) and topologies. Therefore, we first divide all the nets with degree d into $d!$ groups according to their *vertical sequence*. Then we further divide every group into d subgroups. For subgroup $1, 2, ..., d$, the corresponding source pin is pin $1, 2, ..., d$, respectively. For each subgroup, we will have a set of POWVs and their corresponding topologies stored in the table. Note that in FLUTE, a POWV represents **rectilinear Steiner trees** which can potentially have minimum wirelength. In contrast, in this work a POWV represents **A-trees** that can produce optimal wirelength.

Our POWV table stores POWVs for every subgroup. Moreover, while the FLUTE table contains only one topology for each POWV, in the current work we efficiently store *all* topologies for a POWV. This allows us to explore a very large set of topology alternatives for better timing and good wirelength – although they may all have same wirelength. In this sense, constructing the A-tree table is more sophisticated than constructing RSMT tables of FLUTE.

4.3.2 Configuration Graph

Since there are a lot of possible topologies for each subgroup (defined by *vertical sequence* and the source pin) and the number of subgroups ($d \times d!$ for degree-d nets) are huge, the table generation can be very time-consuming (many days). An efficient way needs to be developed instead of directly enumerating all possible topologies. *Boundary compaction* [37] is a very efficient technique to generate topologies. For a net, the *boundary compaction* technique reduces the Hanan grid size by compacting any one of the four boundaries, i.e., shifting all pins on a boundary to the grid line adjacent to that boundary. The set of routing topologies of the original problem can be generated by expanding the routing topologies of the reduced grid back to the original grid. Figure 4.3 uses the compaction of left boundary to illustrate the idea.

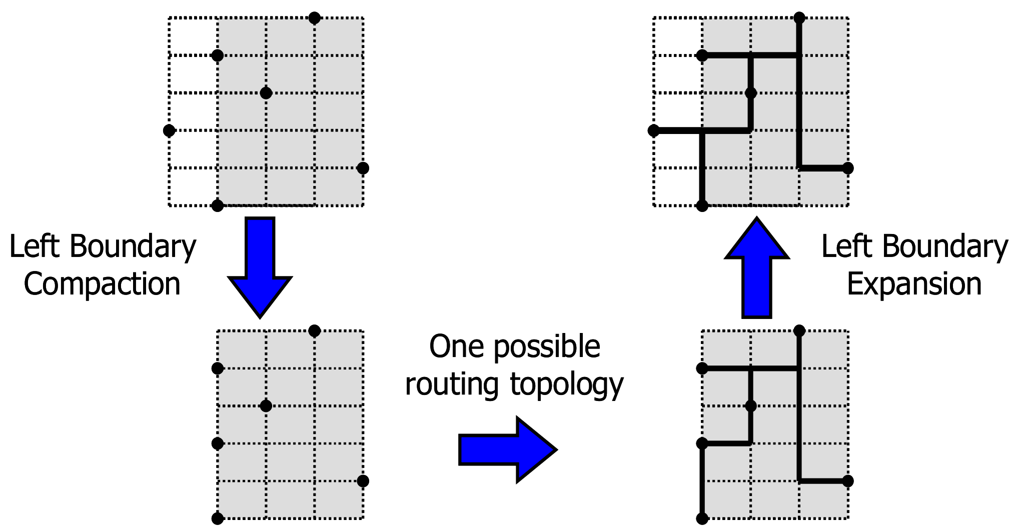

Figure 4.3 Boundary Compaction

We observe that most of the A-tree topologies can be generated by *boundary compaction*. Hence, we employ *boundary compaction* to generate the A-tree topologies. We recursively compact any boundary without the source on it until the grid is compacted into a single node (source). The edges created during the process form an A-tree with the source being the final left node. If we choose different ways for compaction, we will obtain different A-tree topologies. We define the *compacting sequence* as the sequence of compaction operations that reduces the original grid to a single node. Hence, one *compacting sequence* corresponds to one A-tree topology source at the single node left after the compactions. A direct idea for finding different topologies is to look at the different *compacting sequences*. Unfortunately, the number of *compacting sequences* is huge. For each group, the number of different sequences = $\binom{2(d-1)}{d-1} \times 2^{d-1} \times 2^{d-1}$ because we have to perform $d-1$ times of horizontal compactions (left or right) and $d-1$ times of vertical compactions (top or bottom). Therefore, the number of feasible sequences for one group of 9-pin nets = $\binom{16}{8} \times 2^8 \times 2^8 = 843448320$. And this is just for one group, the total # sequences for all 9-pin nets is 9! times this number.

Although the number of *compacting sequences* is huge, we still have hope because we only want to store the different topologies that potentially can result in best wirelength. Therefore, most of the *compacting sequences* can be pruned. But since there are so many sequences, directly generating all sequences and prune them is not practical. Our idea to generate and prune the sequences is using a graph called *Configuration Graph*. We will show that we can generate POWVs for all subgroups in one group (same *vertical sequence* but different source) simultaneously.

First, we define some terms. A *Pin Configuration (PC)* is the configuration of a set of pins on the Hanan grid. This configuration only defines the pin positions on the Hanan grid without considering any real geometry size. If we apply boundary compaction on a *PC*, we will get a new one. The new *PC* has no pin on the compacted boundary and can have the same or less pins than the original because some pins may collapse together.

If the original *PC* is transformed into a new *PC* with a specific bounding box by a sequence of compaction, the new *PC* is independent on the compactions performed, as stated in Lemma 1.

Lemma 1 *The bounding box of a PC in the original Hanan grid defines the PC.*

Proof As shown in Figure 4.4, the whole grid is the Hanan grid of original pin configuration and the center region 3 is the new configuration with bounding box for the center 16 small

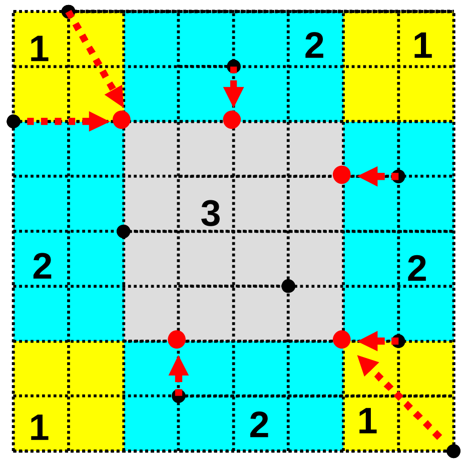

Figure 4.4 Lemma 1 Proof

squares obtained by some *compacting sequence* Q. It is easy to see that all the pins in the four corner region 1 are compacted to the four corners in the new configuration. And the four boundary regions 2 are compacted to the closest boundary with the unique position. The center region 3 is not touched. So every pin has the unique pin position in the compacted configuration. No matter what the *compacting sequence* Q is, every pin has the same position in this configuration. ∎

In *Configuration Graph*, every node corresponds to a *PC*. So we call these nodes *Configuration Nodes (CN)*. There are two kinds of special nodes in the *Configuration Graph*. One is the *CN* corresponding to the original *PC* for a *vertical sequence*. We call it *Start Node* because any boundary compaction operation starts with it. The other type is the *CN* with the *PC* in which all the pins are compacted to a single point on the grid. We call them *End Nodes* because any *compacting sequence* will end with such a *CN*. Note that an A-tree topology is obtained when reaching an *End Node*. A *Partial wirelength Vector* (PWV) is the Wirelength Vector (WV) with undecided entries obtained after a sequence of compactions. For example, if a full WV is $(1221, 1121)$, a PWV could be $(1xx1, 11x1)$ (x means undecided). The undecided part corresponds to the horizontal edges or vertical edges that have not been created by *boundary compaction*. For each *CN*, a set of PWVs are associated with it. They are the PWVs corresponds to the edges created by *compacting sequences* that can result in the the

PC associated with the *CN*. If compacting one boundary of the *PC* associated with a *CN* can get the *PC* of another *CN*, an edge is created from the first *CN* to the second. An example of *Configuration Graph* is shown in Figure 4.5.

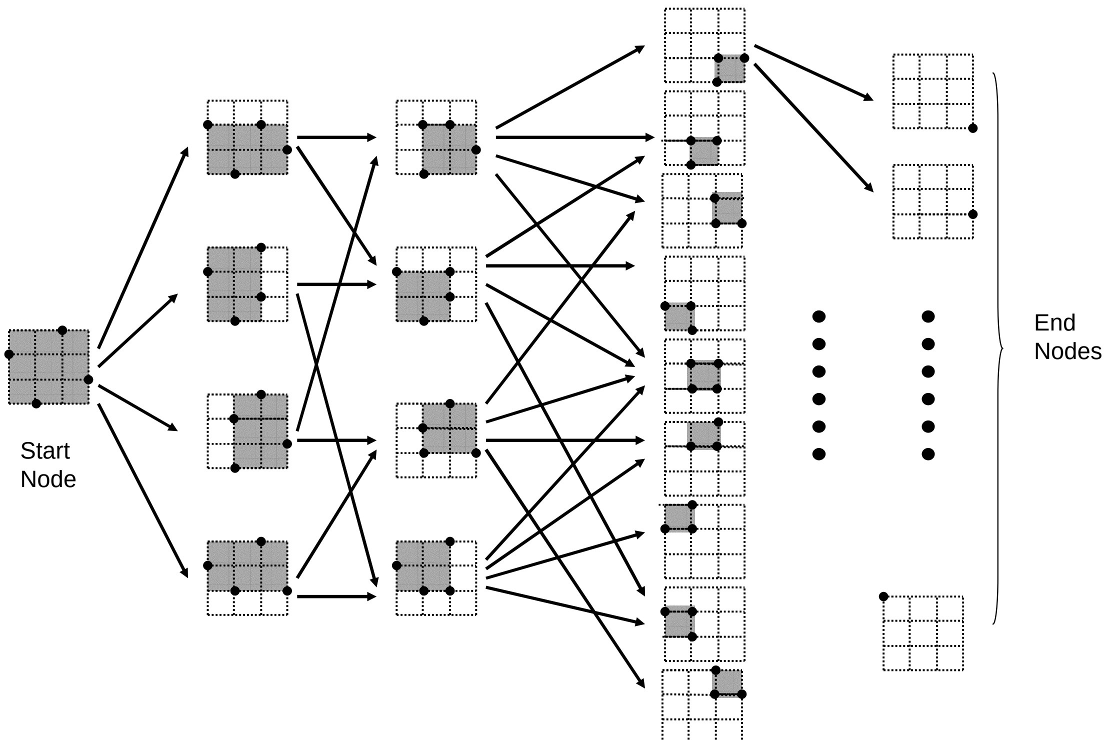

Figure 4.5 Configuration Graph

From Lemma 1, we know the number of *CN*s in *Configuration Graph* is a small number. It is just the number of different bounding boxes we can find in the original Hanan grid. $\#CN = \sum_{i=1}^{d} \sum_{i=1}^{d} (d+1-i)(d+1-j)$, if $d = 9, \#CN = 2025$. Actually, we can even do better. Instead of using the original *PC* as the *Start Node*, we start from a new *Start Node* that is obtained by compacting the original *PC* once in all 4 directions (left, right, top and bottom). We have the following lemma for the new *Start Node*. The proof is similar to the Lemma 2 in [37]. The only difference is that when the source pin is on one of the 4 boundaries, we will treat the new source for the reduced grid at the position of the pin created by compacting the original source.

Lemma 2 *No POWV will be lost by starting boundary compaction at the new Start Node.*

Proof Since we begin from this new *Start Node* with the bounding box size $(d-2)\times(d-2)$, the length of *compacting sequences* is reduced from $2(d-1)$ to $2(d-3)$. And # *CN* can be reduced to $\sum_{i=1}^{d-2}\sum_{i=1}^{d-2}(d-1-i)(d-1-j)$, if $d=9$, $\#CN=784$. ∎

Configuration Graph allows us to do the pruning very efficiently. We have the following lemma.

Lemma 3 *If a PWV at a CN is worse than the other, it cannot be part of any POWV (it can be pruned).*

Proof Prove by contradiction, assume a PWV V_1 at a *CN* is worse than the other V_2, but it is part of a POWV V. From Lemma 1 we know that the undecided part of WV is the same for V_1 and V_2 because of the same *PC*. Let V_b=$V-V_1$. Then V_2+V_b is better than V_1+V_b. A contradiction with V is a POWV. ∎

From Lemma 3, we can use *Configuration Graph* to prune the PWVs using "PWV dominance" at each *CN*s efficiently. We say a PWV is dominated by the other one if it corresponds to more wirelength, i.e., it has the same or bigger value on all entries in WV. This kind of pruning does not wait until the full WV has been generated. It can prune the bad WV as early as possible and accelerates the pruning process a lot. Another advantage for the *Configuration Graph* approach is that if we construct the *Configuration Graph* for a given *vertical sequence*, we already obtain POWVs for all the subgroups corresponding to different source pins. We will see this later.

Hence, we construct *Configuration Graph* for any given group (*vertical sequence*) as following. We start from the new *Start Node* mentioned in Lemma 2. Its corresponding PWV is $(1xx...x1, 1xx...x1)$ because we have four edges due to the first 4 *boundary compactions*. Then, we compact the four boundaries of the current *PC* to get four new *CN*s, their corresponding PWVs, and four edges. Similarly, we just recursively apply boundary compaction on the new created *CN*s and generating more *CN*s. Note that compacting different *CN*s can result in the same *CN* but different PWVs. Therefore, we need to prune the PWVs using "PWV dominance" at each node. Only the PWVs left after pruning associated with a *CN* will be used to generate further PWVs when compacting this *CN* to generate new *CN*. This recursive new *CN* generation will stop when the new generated *CN* is an *End Node*, where no compaction can be applied. After generating all the *CN*s and their corresponding PWV list, we obtain the whole *Configuration Graph* and can easily find the A-tree topologies from it.

It is easy to see that any path from the *Start Node* to an *End Node* corresponds to a *compacting sequence*, hence a tree topology. But our goal is not to find any tree topology but to find A-trees with a specific source pin. We have the following lemma for the generated tree topologies.

Lemma 4 *Any tree topology generated by a compacting sequence corresponding to a path from the Start Node to an End Node is an A-tree with the source at the position corresponding to the End Node.*

Therefore, for any pin as the source, we can easily find the POWVs for the A-trees. We just need to look at the *End Node* corresponding to the position of the source pin and all the POWVs associated with that *End Node* are the POWVs for A-trees with the source pin. Since we have *End Node*s corresponding to every position in the Hanan grid, POWVs for every pin as the source can be obtained simultaneously from the *Configuration Graph*.

4.3.3 Abstract Topology and Topology Signature

By now, we can obtain the POWVs for A-tree topologies from *Configuration Graph*. But unlike FLUTE, we are not satisfied with storing one arbitrary topology corresponding to each POWV. Instead, we want to explore good A-tree topologies for performance. Therefore, we want to find all different A-tree topologies corresponding to each POWV and store them in the table.

We study the topologies generated by different *compacting sequences* corresponding to POWVs and find that most of them are redundant. There are two kinds of redundancy. First, different compacting sequences generate the same topology. Second, different *compacting sequences* generate different but *equivalent* topologies in terms of both wirelength and timing. Two topologies are *equivalent* when they are the same in all node positions (pins and

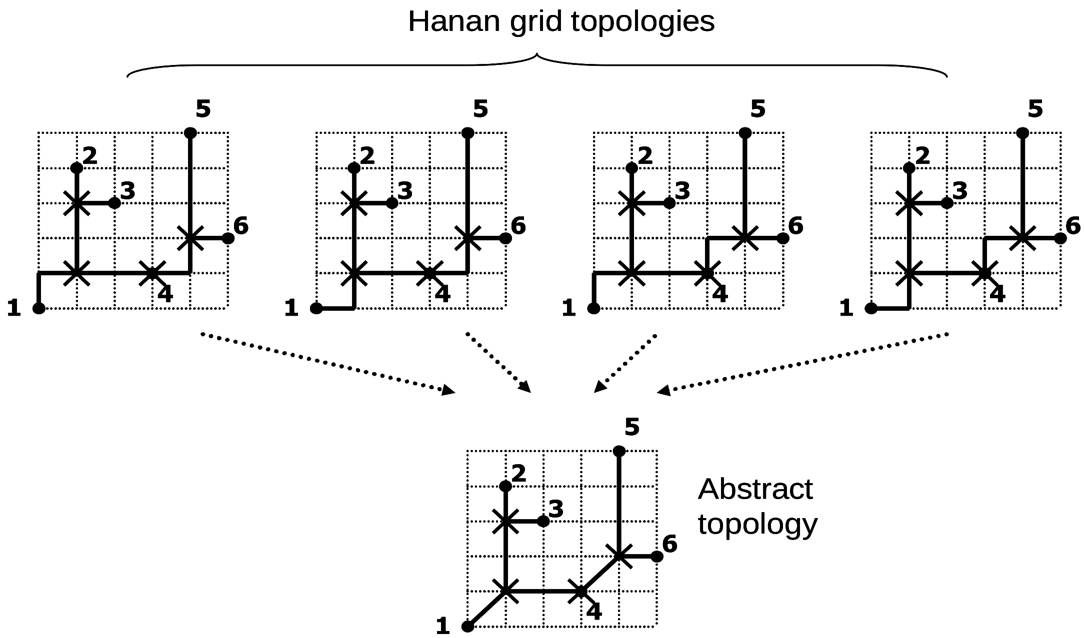

Figure 4.6 Abstract topology

Steiner nodes) on Hanan grid and the connections between nodes. The only difference between *equivalent* topologies is the embedding for the connections. To eliminate these two types of redundancy, we introduce the concept of *Abstract Topology*. An *Abstract Topology* for a net is the topology on the Hanan grid that fixes the positions for all the nodes (pins and Steiner nodes) and the connections between these nodes. The difference between an *Abstract Topology* and a normal topology on the Hanan grid is that the *Abstract Topology* does not specify how the connection is embedded on Hanan grid. If two *compacting sequences* generate the same topology or *equivalent* topologies, their corresponding *Abstract topology* are the same. Therefore, we only need to store the different *Abstract topologies* for POWVs. Figure 4.6 illustrates the concept of *Abstract Topology* for a 6-pin net. Although the concept of *Abstract Topology* is very simple, it can save huge amount of table space. For example, consider a 9-pin *Abstract Topology* with 7 steiner nodes, 15 two-pin connections. If there are 2 different routing on Hanan grid for 10 two-pin connections in 15, # embedded topologies = $2^{10} = 1024$. If we just directly save the different topologies, we may need thousands of times space than just storing *Abstract*

Topologies.

The way we find different *Abstract Topologies* is as following. We start from the *End Node* corresponding to the source pin and for every POWV, trace back in the reverse direction of edges until reaching the *Start Node*. This back trace will form different paths corresponding to different *compacting sequences*. Since each *compacting sequence* corresponds to an A-tree topology, we can get all the possible A-tree topologies for each POWV. Then we can compare their *Abstract Topologies* and just store the different *Abstract Topologies* in the table.

However, there is still one problem in generating and comparing the *Abstract Topologies*. To know whether a topology is redundant, we need to first find its corresponding *Abstract Topology* and compare it to all the other *Abstract Topologies* already found. This topology generation and comparison take a lot of runtime. We want to make it easier and faster. So we introduce the *Topology Signature*. A *Topology Signature* of a Hanan grid topology (for a given *pin configuration*) is the positions of the Steiner nodes in the topology. The following lemma gives us a better way to find whether two tree topologies have the same *Abstract Topology*.

Lemma 5 *For A-trees generated by boundary compaction, two trees A and B has the same Topology Signature if and only if A and B has the same Abstract Topology.*

Lemma 5 tells us that *Abstract Topology* and *Topology Signature* has one-to-one correspondence. So *Topology Signature* is really the "signature" for topologies. Therefore, instead of finding all different *Abstract Topologies*, we only need to find the topologies with different *Topology Signatures*. For the topologies generated by different *compacting sequences*, it is enough to simply compare their Steiner nodes positions on Hanan grid. After we find all the topologies with different *Topology Signatures*, we store their corresponding *Abstract Topologies* in the table.

Till now, we can generate A-tree POWVs and *Abstract Topologies* and store them in the table grouped by the *vertical sequence* and the source pin. Table 4.1 gives the statistics of our POWV table for the nets up to degree 9. And we observed in experiments that all POWVs for the nets up to degree 9 have only **ONE** *Abstract Topology*. With our algorithm based on *Configuration Graph*, it only takes less than 15 minutes to generate the table for all nets up to

Table 4.1 Statistics for POWV

Degree n	# groups n!	# POWVs in a group		
		Max	Avg	Min
2	2	1	1	1
3	6	1	1	1
4	24	8	6	1
5	120	18	12.63	1
6	720	36	25.31	1
7	5040	70	50.69	1
8	40320	144	99.55	1
9	362880	282	193.19	1

degree 9 compared to many hours for generating FLUTE table up to degree 9. The table size for POWV and *Abstract Topologies* up to degree 9 are 21MB and 75MB, respectively. Note that this table only needs to be generated once. And after loaded into the memory, it can be used for as many times as wanted.

4.4 A-tree Topology Construction and Net-breaking

In last section, we construct the A-tree POWV table and corresponding topology table for the nets up to some degree D. Therefore, for any net with degree no more than D, we can find the corresponding group index based on the *vertical sequence* of the net. Having the group index and the source pin, we directly look up the POWV table to find the corresponding POWVs and compute their wirelength based on the real geometric information of the net. Then we pick the POWV with best wirelength and look up the topology table for the A-tree topology corresponding to it.

However, it is not practical to generate table for high-degree nets because of the huge table size. Therefore, a high-degree net will be divided into several sub-nets with degree less than D to which the table lookup can be applied.

The net-breaking method we use is different from that in [38] because we are generating A-tree instead of RSMT. Different heuristics need to be applied and the source needs to be considered when breaking a net.

We can still use the optimal net-breaking algorithm proposed in [38]. If a net satisfies that all the pins in the net can be partitioned into two sets which reside in two diagonal regions, it can be optimally broken into two sub-nets formed by these two sets. An extra pin is introduced in both sub-nets. The pin is positioned at the bounding box corner of one sub-net which is closest to the other sub-net. After the breaking, only one sub-net contains the source. For the other sub-net, we need to specify a source pin. It is very simple in this case that we make the extra pin introduced in both sub-nets as the source for the sub-net without the original source in it. If we construct A-trees for both sub-nets, the combined tree is still an A-tree.

If there is no optimal breaking for a net, we will break the net in x or y direction. However, we cannot directly break the net at some pin and combine the two trees for the two sub-nets to obtain the whole tree as in [38] because it will not result in an A-tree. Therefore, with the A-tree constraint, instead of including the pin where the net is broken in both sub-nets, we introduce an extra pin and include it in both sub-nets. This extra pin will become the source of the subset that does not have the original source in it. The position of this extra pin is found by a "source propagation" technique. Assume the breaking direction and position are known. We first project the source on the breaking line to get the initial position of the new source. However, this position may not be good in some cases. For example, in Figure 4.7, all the pins in the right sub-net have bigger y-coordinates than the source. If we put new source at the position of the projection of the source on the breaking line, it could lead to unnecessary extra wirelength. In order to solve this problem, we slide the source projection along the breaking line until it has the same y-coordinate as the pin with the smallest y-coordinate in the right sub-net. It is easy to see that this operation will not affect the A-tree property of the whole tree. Apparently, this idea can be used no matter in what direction the net is broken and which sub-net the source is in. The new source found is noted as "propagated source".

Lemma 6 *Breaking a net at the "propagated source" generates an A-tree by combining the A-trees of both sub-nets. (The sources of two sub-nets are the original source and the propagated source).*

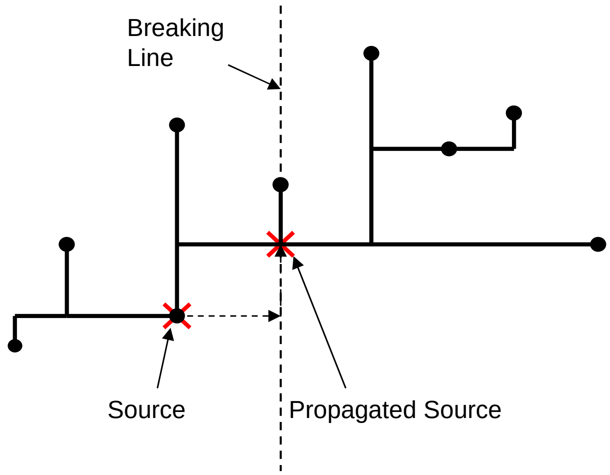

Figure 4.7 Source propagation

Proof Let N, N_1 and N_2 be the original net and two sub-nets after breaking and let S, S_2 be the source of N and the extra pin. Without loss of generality, we assume S is in N_1 after breaking. If T_1 is an A-tree for N_1 with source S and T_2 is an A-tree with source S_2, all the nodes in N_1 have the shortest path to S and all the nodes in N_2 have the shortest path to S_2. Since S_2 is in N_1, S_2 has the shortest path to S (which is a straight line). It is obvious that all the nodes in N_2 has the shortest path to S. Therefore, A_1+A_2 is an A-tree for net N. ∎

For the heuristics of choosing a good direction and position to break the net, we apply a slightly different heuristics from that in [38]. We also compute a score which is a weighted sum of three components. For the first and third component, we follow the way in [38] to find them. But for component two, since the real breaking point is the "propagated source", we will consider the lengths of the segments adjacent to the "propagated source" other than those adjacent to the pin on the candidate breaking line.

After we break the net into sub-nets with degree no more than D, we can look up the table to find out the topologies for them. Finally, we merge these subtrees to form the whole A-tree.

After the A-tree for the net is obtained, we apply a heuristic to repair the errors caused by the nonoptimality of the table and net-breaking. For each node on the tree (pins and Steiner nodes), we try to connect it to the closest point on the tree and in the direction of the source. This will further improve the wirelength and still maintain the A-tree property.

4.5 Performance-driven Post-processing

So far, we can construct a good A-tree topology for any given net by net-breaking and table lookup. However, the topology is still a generic A-tree without consideration of the timing properties of a specific net. In general, A-tree is a good topology in performance-driven routing if there is no difference between all the sinks in criticality. However, for a specific net, different sinks have different capacitive load, required time and distance to the source. This makes it more complicated to find a good topology in terms of performance.

Since we already have the A-tree as a good initial topology, it will be very convenient if we can make improvement on the obtained A-tree to achieve better timing. In addition, we are aiming at some very efficient heuristics so that it can match our fast table lookup based A-tree construction technique. Therefore, we propose a performance-driven post-processing heuristic to modify the tree topology to achieve better timing result.

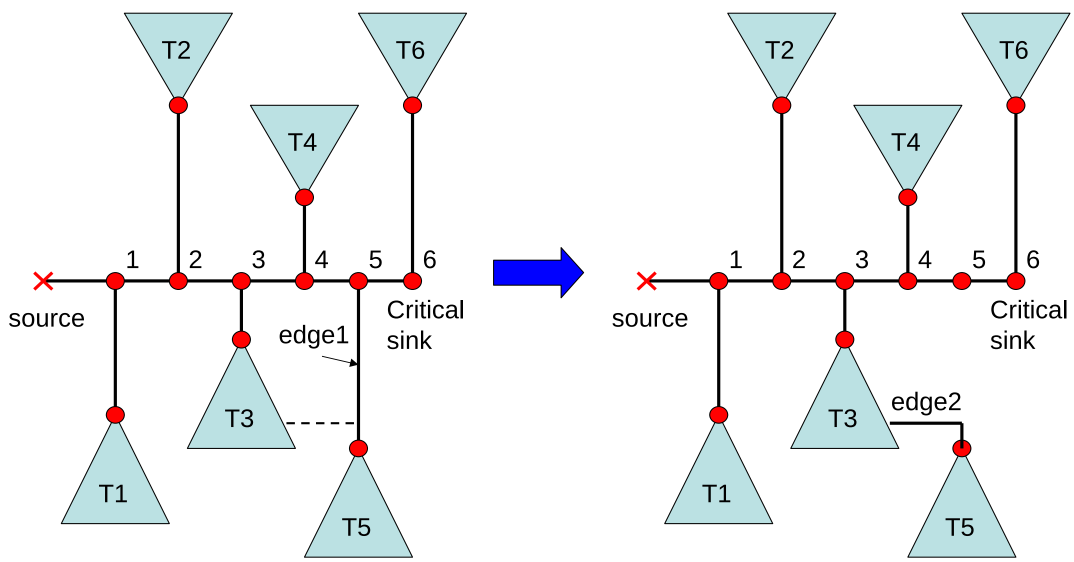

Figure 4.8 Branch Moving

Our heuristic is called "branch moving", which change the tapping point for some branches in the tree. At this stage, we no longer restrict the topology to A-tree. The basic idea is to change the load distribution on the critical path to reduce the delay on critical sinks. To easily understand the technique, let us look at a simple example. As illustrated in Figure 4.8, we have a tree topology (left) and know the critical sink by timing analysis. Now we want to look at the possibility to improve the timing for the critical sink. We first label the tree nodes on the critical path with numbers. These numbers represent the distances from the source to the nodes. The bigger the label on the node, the farther to the source. We use Elmore delay model for our delay computation. Therefore, the delay on the critical sink is the sum of a series of RC terms, $Delay(Critical\ sink\ 6) = \sum_{i=1}^{6} R_i C_i$, where R_i is the path resistance from source to node i, C_i is the downstream capacitance of the subtree T_i rooted at node i, (excluding the critical path). If we change the tapping point of some branch, the delay on the critical sink will be changed as well. Hence, we try to move the branches so that the delay on the critical sink is reduced. For instance, in Figure 4.8 (left), we find a possible edge (dashed line) which connect the branch tapped to node 5 to the subtree tapped to node 3. It is easy to compute the delay change on the critical sink. $\Delta d = R_3(C_5 + C_{edge2}) - R_5(C_5 + C_{edge1})$. Therefore, we can quickly find the delay change on the critical sink when moving a branch.

In fact, this "branch moving" technique is very flexible. You can find an edge (not existing in the original tree) between any two node on the tree and try to connect them. This operation will form a loop. In order to maintain the tree topology, we can break an edge in the formed loop to obtain a new tree. However, there are too many choices for the edge to be connected and broken. In our implementation, we constrain all the edges in the tree on the Hanan grid. Therefore, We find all the edges on the Hanan grid which is not in the tree. Then we measure the "benefit" to connect any of the candidate edges and break another edge in the formed loop. Here, the "benefit" is the delay reduction on the critical sink. Among all the candidates, we simply pick the one with best "benefit" and update the tree. We apply this "branch moving" iteratively until no improvement.

So far, we have introduced the "branch moving" technique to improve the critical sink delay. However, there are several problems that need careful consideration. First, we should not touch the nodes on the critical path. Otherwise, we will create detour from source to the critical sink. Second, although reducing critical sink delay is the major objective here, we do not want to increase the wirelength too much for two reasons: 1. more wirelength corresponds to more routing resources and power, 2. more wirelength could increase the delay for the whole tree for the increased capacitive load. Hence, we add a weighted wirelength part in the "benefit" to discourage the wirelength increase. Finally, moving a branch can reduce the critical sink delay, but it could also cause other sinks to become critical. Therefore, we need to add constraints on "branch moving". When picking the candidate edge, we look at the two nodes that the edge is to connect. If any of the downstream sinks of these two nodes will become critical after "branch moving", we will not consider this edge.

4.6 Experimental results

We test our new method on 2 sets of industrial nets. The first set is from a design in 65nm technology and the second is from a design in projected 45nm technology. Two metal layers are used for routing. These two sets of nets are critical nets extracted from the designs after the timing analysis. The first set has 17 nets and the second set has 12 nets.

We try to find performance-driven topology design algorithms in the public domain. However, most of these algorithms are together with other interconnect optimization techniques such as buffer insertion and wire sizing. Since our focus is topology design, it is very hard to find some direct comparison with these algorithms. Hence, we compare our new algorithm with C-tree [32] and FLUTE [38]. Both of them are downloaded from the GSRC Bookshelf [45]. Since C-tree algorithm is a combination of timing-driven Steiner tree construction and buffer insertion, we turned off the buffer insertion by not specifying any buffer library. In addition, we also turned off the sink polarity by setting all sinks the same polarity as source. FLUTE is used to generate near-optimal rectilinear Steiner minimal trees. All results are generated on a 750MHz Sun Sparc-2 machine.

The result for 6 nets from each set are reported in Table 4.2. We compare the tree wirelength, worst negative slack (WNS), total negative slack (TNS) and runtime for the three algorithms. Note that we report WNS and TNS for A-trees obtained by our algorithm and the final trees obtained after post-processing. We can see the post-processing technqiue is very effective in reduce WNS and TNS. The wirelength and runtime are normalized to our algorithm.

For all the 29 nets, our algorithm always achieves the best WNS and TNS among the three algorithms. We also take the average on all the 29 nets for these measurement. On average, C-tree uses 9.5% more wirelength than ours as FLUTE uses 8.5% less wirelength. And WNS and TNS of trees generated by our algorithm are 82.2% and 57.7% that of C-tree and FLUTE, respectively. From the comparison to FLUTE, we can see that although the tree generated by our algorithm has more wirelength than RSMT, their performance is better. This verifies our proposition that RSMT may not be good for timing. For the runtime, we are just slightly slower than FLUTE and 371 times faster than C-tree. Note that for the nets with degree more than 60, the runtime of our algorithm is about 1ms, which means we can handle 1000 that kind of high-degree nets in one second.

Table 4.2 Comparison between performance-driven interconnect trees generated by different algorithms

	degree	Tree Wirelength			WNS(ps)				TNS(ps)				Runtime		
		Our	C-tree	FLUTE	Our A-tree	Our Final	C-tree	FLUTE	Our A-tree	Our Final	C-tree	FLUTE	Our	C-tree	FLUTE
t1	9	1	1.029	0.914	-0.97	-0.80	-0.97	-0.87	-0.97	-0.80	-0.97	-0.87	1	111	0.11
t2	38	1	1.112	0.936	-5.66	-5.40	-5.71	-5.55	-5.66	-5.40	-5.71	-5.55	1	191	0.57
t3	58	1	1.176	0.809	0.00	0.00	-1.98	-21.61	0.00	0.00	-1.98	-144.3	1	704	1.15
t4	21	1	0.983	0.793	-16.32	-14.33	-15.62	-20.72	-32.34	-28.52	-31.10	-41.03	1	286	0.48
t5	9	1	1.032	0.968	-4.10	-3.81	-3.91	-4.20	-7.95	-7.31	-7.52	-8.07	1	250	0.13
t6	51	1	1.145	0.782	-1.82	0.00	-2.14	-9.76	-1.82	0.00	-2.14	-26.41	1	1255	0.89
n_1885	27	1	1.077	0.860	-4.56	-1.51	-3.73	-6.19	-4.56	-1.51	-3.73	-6.19	1	346	0.73
n_1898	39	1	1.052	0.907	-4.91	-2.73	-4.75	-8.85	-4.91	-2.73	-4.75	-8.85	1	304	0.87
n_2045	54	1	1.181	0.897	-22.71	-22.71	-25.29	-23.28	-126.0	-126.0	-155.4	-147.3	1	455	0.75
n_2049	45	1	1.158	0.924	-2.95	-0.62	-3.55	-5.43	-2.95	-0.62	-5.27	-7.97	1	468	0.84
n_2071	29	1	1.079	0.890	-12.99	-10.66	-14.51	-14.38	-12.99	-10.66	-14.51	-14.38	1	375	0.56
n_2072	69	1	1.180	0.845	-14.72	-12.09	-22.98	-61.55	-48.39	-37.92	-96.73	-1420	1	385	0.74
Avg.[1]	28	1	1.095	0.915	-7.38	-6.09	-7.41	-10.55	-21.96	-18.76	-22.87	-75.27	1	371	0.487

1. Average is over all 29 testcases

CHAPTER 5. FASTROUTE - A STEP TO INTEGRATE GLOBAL ROUTING INTO PLACEMENT

5.1 Introduction

Placement has become a critical step in VLSI design flow. The two major causes are both related to the increasing dominance of interconnect in nanometer-scale IC technologies. First, placement largely determines the performance of a circuit. As feature size in advanced VLSI technology continues to shrink, interconnect delay has become the determining factor of circuit performance. Placement decides the length and hence the delay of interconnect wires to a large extent. Many recent articles reported that interconnect delay can consume as much as 75% of clock cycle in modern designs. Therefore, a good placement solution can substantially improve the performance of a circuit. Second, placement also determines the chip size. Because of the shrinking of device size, the chip area is no longer determined by total cell area, but by the limited routing resources. Extra "white space" is commonly added to provide enough wire tracks to resolve routing congestion. It is typical that more than half of the modern chip is occupied by white space. A good placement needs to allocate white space appropriately to use the chip area effectively.

Because it is very difficult to incorporate circuit delay or routing congestion directly into the placement objective function, timing-driven and congestion-driven placement algorithms typically employ iterative improvement approaches [13][49][50]. First, a placement solution is produced. Next, timing/congestion information are estimated based on the current placement. Then the estimated information are fed back to direct the placer to generate a better placement. This process iterates until there is no significant improvement on timing or congestion objective. To estimate timing, interconnect delay is obtained from very rough interconnect models such as half-perimeter of the bounding box or star-model. Due to lack of routing information, it is impossible to get accurate interconnect topology, wirelength, and possible buffer positions and sizes. Hence, interconnect delay cannot be estimated accurately. To estimate routing congestion, previous works proposed generic estimators which aim at predicting the behavior of all routers consistently. However, as we point out in Section 5.2, routing solutions generated by different routers are very different. Therefore, it is not possible for an estimator to accurately predict congestion of all routers.

In order to get accurate interconnect information during the placement process, it is desirable to incorporate global routing into it. Global routing allocates the routing demand globally over the chip area. It generates interconnect information very close to the final routing implementation and can be used for accurate estimation of interconnect topology, wirelength, delay, congestion, buffering solution, etc. However, due to the high runtime of the traditional global routers, it is impractical to perform global routing repeatedly during placement.

In this work, we develop an extremely fast high-quality global router called *FastRoute*. Experimental results show that *FastRoute* can generate less congested global routing solutions with 132× and 64× speedup over the state-of-the-art academic global routers *Labyrinth* [46] and *Chi Dispersion* router [47], respectively. Very surprisingly, *FastRoute* is even faster than the highly-efficient congestion estimation algorithm *FaDGloR* [48].

The runtime of *FastRoute* is only 1/934 and 1/2229 of the runtime of state-of-the-art academic placers *Capo9.1* [10] and *Dragon3.01* [29], respectively. The promising runtime makes it possible to incorporate global routing directly into the placement process without much runtime penalty. This could dramatically improve the placement solution quality because accurate interconnect information becomes available during the placement process. Note that although we emphasis on the application in placement, we can apply our global router in any early design stage which has the pin locations fixed, e.g., floorplanning and trial placement in physical synthesis loop. We believe that this work will fundamentally change the way the EDA community look at and make use of global routing in the whole design flow.

FastRoute can achieve superior quality and speed because of the following techniques.

- A congestion-driven Steiner tree topology construction method to distribute routing demand according to the congestion map.
- An edge shifting technique to move the horizontal or vertical tree edges in a Steiner tree from congested regions to less congested regions without changing wirelength of the tree.
- A new cost function based on logistic function to direct the maze routing to find less congested paths.

Traditional global routing approaches do not explore the flexibility of tree structures to resolve routing congestion. They just employ spanning tree or Steiner tree algorithms to construct trees for multi-pin nets. Then the tree structure of each net is fixed and broken into into a set of two-pin nets. After that, they rely on the maze routing to route the two-pin nets to resolve the routing congestion. The global routing runtime is dominated by maze routing. Different from them, we shift the burden of maze routing to Steiner tree construction. We focus on determining good Steiner tree topology and Steiner nodes locations according to congestion information. As a result, the maze routing has a good initial solution to work with and less effort is needed.

The remainder of the chapter is organized as follows. In Section 5.2, we review the previous work in global routing, timing estimation and congestion estimation, as well as the problems with current estimation techniques. In Section 5.3, we describe the flow of *FastRoute* and explain its underlying idea. Next in Section 5.4, 5.5 and 5.6, we present the three major steps of *FastRoute* in detail. In Section 5.7, we perform extensive experiments and show the comparison results.

5.2 Previous Work and Some Discussion

5.2.1 Global Router

The grid graph model is widely used in global routing [46][47][51]. In this model, the chip area is partitioned into rectangular regions called global bins and all the pins in a global bin are assumed to be at the center of the bin. Each global bin corresponds to a node in grid graph.

The boundaries of global bins are called global edges, which correpond to the edges in grid graph. The capacity of an edge represents the number of routing tracks for the corresponding boundary. These notions are illustrated in Figure 5.1. The major optimization objective in global routing is to minimize the total overflow on all the edges in the grid graph. The overflow on a global edge e is defined as how much the routing demand d_e exceeds the edge capacity c_e. If $d_e > c_e$, $overflow_e = d_e - c_e$; otherwise $overflow_e = 0$.

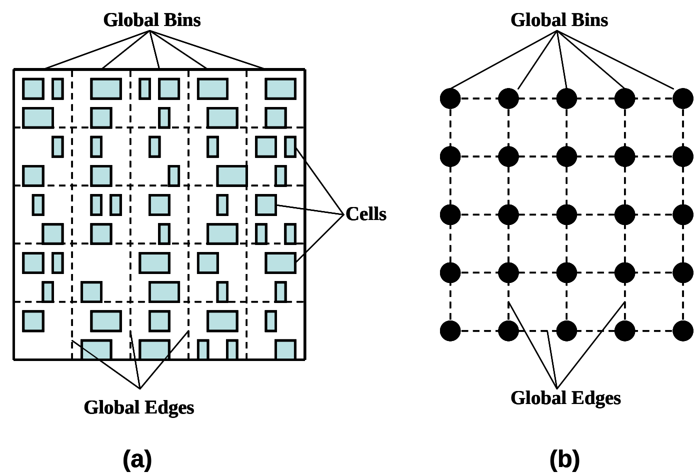

Figure 5.1 (a) Global bins. (b) Corresponding grid graph

Most academic and industrial global routers [46][47] first decompose every multi-pin net into a set of two-pin nets by spanning tree or Steiner tree algorithms. After the decomposition, each two-pin net is routed by maze routing. To further improve the solution quality, those routers utilize rip-up and reroute technique. Albrecht [51] proposed a new multi-commodity flow approximation algorithm to solve the global routing problem. The approximation algorithm uses fractional flows. Hence, it is necessary to perform randomized rounding, followed by traditional rip-up and reroute to complete the process.

5.2.2 Timing Estimation

Previous timing-driven placement algorithms generally employ iterative approaches. For a given placement, the critical path information are obtained by timing analysis. Then the timing information obtained is fed back to the placement engine to generate a new placement solution favoring the critical path. The common methods are adding weight to the critical nets/paths in the objective function [13][52], adding constraints to the critical nets/paths [49], or adding penalty for the critical nets/paths to the simulated annealing cost function [53][54]. The basic assumption here is that the timing information obtained are accurate and can be used to direct the placement process.

As interconnect delay becomes the dominant part of circuit delay, accurate interconnect information is needed for timing analysis. However, since there is no routing information, it is impossible to get accurate estimate for interconnect. Early works neglect the interconnect delay in timing analysis. Many recent works [13][53][54] employ the half-perimeter of the bounding box to estimate the interconnect length. For multi-pin nets, they first lump all sinks of a net together and assume the lumped sink is driven by the driver through a single wire. The wire length is estimated by the half-perimeter of the bounding box. Hence, they can compute the wire capacitance and resistance using this length. In [52][55], authors used a star-model to approximate a multi-pin net. An star point is put at the center of gravity of all pins of the net. However, considering the real implementation, multi-pin nets are typically routed as Steiner trees, and global nets are inserted with buffers to minimize the delay. Hence, both half-perimeter of the bounding box model and star-model is far from accurate for interconnect timing estimation.

5.2.3 Congestion Estimation

Fast congestion estimation is essential for congestion reduction techniques at different stages of the design flow. Post-placement congestion estimation methods try to predict the routing congestion for a given placement. In recent years, a number of probabilistic methods for congestion estimation have been proposed. Lou et al. [56] break mult-pin nets into two-pin wires. Probabilistic usages are then assigned to tiles according to the probability that a two-pin wire will be routed through the tile. Based on the observation that detours are rare, each detour-free path connecting the two pins is assigned an equal probability. Westra et al. [57] and Kahng et al. [58] observed that routes with one or two bends are more likely to occur than multi-bend routes. Consequently, probabilities for the occurence of L-shapes and Z-shapes are emperically derived from industrial designs and are used to improve upon [56]. In [48], Westra et al. presented two congestion estimation tools. The first one, called *pce*, is an implementation of a probabilistic method which is very fast in comparison with other probabilistic methods. The second one, called *FaDGloR*, is new and based on degenerate global routing techniques. Experiments show that *FaDGloR* is about as fast as *pce*. They concluded that global routing based methods are probably more worthwhile than probabilistic methods in congestion estimation. However, unlike the normal global routers, *FaDGloR* does not generating the feasible global routing solutions that minimize overflow.

We notice that for the same circuit, different routers can give very different routing solutions, hence very different congestion distribution. So we have a basic question - is it possible for a generic congestion estimator to accurately predict the routing congestion for all routers? To answer this question, we investigate the routing solutions generated by two global routers - *Labyrinth* and *Chi Dispersion* router, also the routing solutions generated by *Labyrinth* but using different parameters. For a global edge in the grid graph model, if the routing demand on it is greater than its capacity, we say it is congested. Otherwise, it is uncongested. If a global edge is congested in one routing solution and uncongested in the other, we call it a *congestion mismatch*. The total number of *congestion mismatches* gives the similarity of congestion distribution between two routing solutions. Note that *congestion mismatch* is similar to the "wrongly congested" and "wrongly uncongested" notions in [48]. There, congestion is defined as the ratio of routing demand and capacity. The "wrongly congested" happens if the estimated congestion c is greater than 1.1 but real congestion C is lower than 1.1; the "wrongly uncongested" happens if the estimated congestion c is lower than 0.9 but real congestion C is higher than 0.9. We notice that this metric is not proper. Assume that the estimator simply gives the congestion estimation of $c = 1.0$ over the whole grid graph. In this metric, both the number of "wrongly congested" and "wrongly uncongested" edges are 0. Hence, we propose the *congestion mismatch* as the metric.

We perform the experiments as follows. We use the benchmark circuits provided by the authors of [46]. For each circuit, we generate a routing solution using *Labyrinth* (70% shortest nets use pattern routing) and make it as the standard. Then, we also generate two other routing solutions using *Labyrinth* (50% shortest nets use pattern routing) and *Chi Dispersion* router. We find the number of congestion edges for all three routing solutions, as well as the number of *congestion mismatches* between the standard solution and each of the other two solutions. Table 5.1 shows the number of congestion edges and the number of *congestion mismatches*. *Lab (70%)* and *Lab (50%)* means the routing solutions generated by *Labyrinth* with 70% and 50% shortest nets pattern routed, respectively. And #Mismatch in *Lab (50%)* and *Chi Dispersion* columns are the number of *congestion mismatch* compared to (*Lab (70%)*).

Table 5.1 Comparison of number of congestion edges and Congestion Mismatch

	Lab (70%)	Lab (50%)		Chi Dispersion	
	#Con	#Con	#Mismatch	#Con	#Mismatch
ibm01	238	268	398	122	272
ibm02	368	390	580	46	400
ibm03	247	214	367	1	248
ibm04	588	596	662	273	539
ibm06	367	391	596	9	374
ibm07	568	643	887	122	580
ibm08	486	655	865	30	480
ibm09	377	399	638	12	383
ibm10	501	376	691	27	496

From the table we can see that the number of *congestion mismatches* is so significant that it is even more than the number of congestion edges in routing solutions in almost all cases. If we code the congested edge as 1 and uncongested edge as 0, the congestion of a routing solution can be represented as a binary pattern (congestion pattern). The number of *congestion mismatches* of two routing solutions is the Hamming distance [59] between their corresponding congestion patterns. Hamming distance satisfies the triangle inequality: $d_H(x, y) \leq d_H(x, z) + d_H(y, z)$. Assume we use a congestion estimator with the congestion pattern z to estimate the congestion, the numbers of the *congestion mismatches* over the two routing solutions with congestion patterns x and y are $d_H(x, z)$ and $d_H(y, z)$, respectively. From the triangle inequality, we know that the sum of $d_H(x,z)$ and $d_H(y,z)$ is at least $d_H(x,y)$, which is the number of *congestion mismatches* between the two routing solutions. Hence, at least one of $d_H(x,z)$ and $d_H(y,z)$ is bigger than $0.5d_H(x,y)$. Since $d_H(x,y)$ is more than the number of congested edges in either routing solutions, at least for one routing solution, the number of wrongly estimated edges is more than 50% of the number of congested edges in that solution. Therefore, it is impossible for an estimator to claim it can estimate both routing solutions accurately. In fact, the results also show that even using one global router to predict the behavior of another global router (or using one global router with a set of parameters to predict itself with a different set of parameters) is not possible. Therefore, the only possible way to predict congestion accurately is to use the same technique and parameters in both congestion estimation and global routing.

5.3 Outline of FastRoute

Our goal is to develop a very fast high-quality global router which can be used as both interconnect estimator and traditional routing tool. Hence, we care a lot about the runtime of the router. Maze routing is effective in directing routes away from congested region. However as pointed out by many works (e.g, [46]), maze routing is the major contributor of global routing runtime. If we want to achieve orders of magnitude faster runtime, a lot of maze routing has to be cut down.

As far as we know, previous global routers do not consider the effect of routing tree structures on reducing congestion. RSMT or minimum spanning tree is constructed for each net and broken into two-pin nets. Later, every two-pin net is routed independently without touching the original tree structure. In contrast, our approach focuses mainly on the Steiner tree structures to construct good Steiner trees for better congestion results. The routing demand is allocated by these Steiner trees according to congestion map to alleviate the burden of later maze routing phase.

The main flow of *FastRoute* includes three phases:

1. Congestion map generation.
2. Congestion-driven Steiner tree construction.

3. Routing two-pin nets using pattern routing and maze routing.

In the following sections, we will discuss the three phases in detail.

5.4 Congestion Map Generation

In this section, we will describe how to generate the congestion map in the first phase.

We mentioned in Section 5.3 that we will construct the Steiner tree according to routing congestion. Hence, before the congestion-driven Steiner tree construction, we need congestion information. Since we are aiming at a very fast global routing algorithm, we need a very fast but fairly good congestion estimation technique.

First, we generate the Steiner minimal trees for all the nets using *FLUTE* [37]. *FLUTE* is a very fast and accurate rectilinear Steiner minimal tree (RSMT) algorithm. It generates optimal RSMT for nets up to degree 9 and is still very accurate for nets up to degree 100, and is much faster than other RSMT techniques. It is very suitable for our application. Second, after generating the Steiner trees, we break all Steiner trees into two-pin nets. For every two-pin net, we assign the demand to the global edges in the grid graph in the following way. If the two pins of a net have the same x coordinates or y coordinates, we assign demand 1.0 to each global edge on the straight line connecting the two pins. If the two pins of a net have different x and y coordinates, we assume two possible L-shape (sometimes called 1-bend) routings for it - the upper L or lower L. For each edge on the two L-shape routings, we assign demand 0.5 to it. In this way, we get the very first congestion map. Finally, in order to make the congestion map more accurate, we perform a fast rip-up and reroute using L-shaped pattern routing. For each two-pin net, we first remove its routing demand from the congestion map which is added in the second stage. Then we perform routing based on the current congestion map by taking the L-shape which passes through a less congested region. After a full round of L-shaped pattern routing for all the two-pin nets, we get a solution and its corresponding congestion information. We use it as the congestion map for the following congestion-driven Steiner tree construction. Of course, we can use maze routing here, but it will consume a lot of runtime. Since we will change the Steiner tree structures later, it is not worthy to spend the time to perform maze routing in this phase.

5.5 Congestion-driven Steiner Tree Construction

In this section, we focus on the Steiner tree structures to alleviate routing congestion. This is the key phase in the whole flow of *FastRoute*. First, we discribe the two major techniques, *Congestion-driven Topology Generation* in Section 5.5.1, and *Edge Shifting* in Section 5.5.2. Then the flow for the congestion-driven Steiner tree construction phase (phase 2) is given in Section 5.5.3.

5.5.1 Congestion-driven Topology Generation

There is a lot of research on Steiner tree problem. Previous works in global routing apply RSMT algorithms to find Steiner trees to minimize routing tree length. However, our goal is to construct the Steiner tree in favor of congestion reduction.

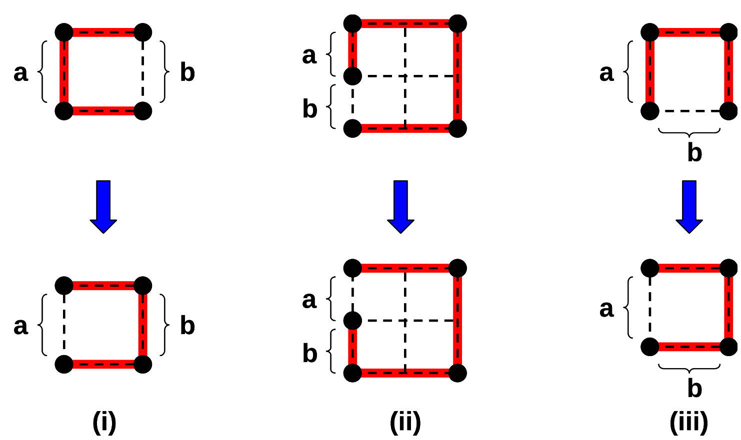

Figure 5.2 Three ways of reallocating routing demand

Routing congestion happens when there is more routing demand than the capacity of global edges. Global routing essentially allocates routing demand over the global edges. The total routing demand of a net is its routing tree length. If a net routed with minimum wirelength uses a congested edge, we have no way to simply eliminate the routing demand on that edge. We have to reallocate it to some other global edges. Without loss of generality, assume a vertical global edge a is congested. There are three ways to reallocate some routing demand on a. (1) Reallocate the demand to another vertical global edge in the same row as a. For example, in Figure 5.2(i), global edge b is used instead of a. (2) Reallocate the demand to another vertical global edge not in the same row as a. For example, in Figure 5.2(ii), global edge b is used instead of a. (3) Reallocate the demand to a horizontal edge. For example, in Figure 5.2(iii), global edge b is used instead of a.

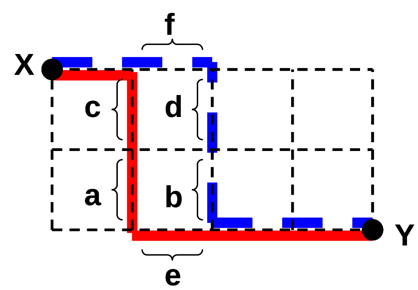

Figure 5.3 Pattern/maze routing example

We observe that the widely used pattern routing and maze routing are applying the first way only. For example, in Figure 5.3, the route from X to Y (solid line) goes through a congested global edge a. To avoid congestion, we can take an alternative route (dashed line) to reallocate the demand to b from a. However, we also need to reallocate the demand from c to d, and from e to f at the same time. Notice that pattern routing and maze routing are not able to reduce the routing demand on any row of vertical global edges or any column of horizontal global edges. On the other hand, ways (2) and (3) can help. Way (2) could move the demand in a specific row (column) of global edges to another row (column) of global edges. Way (3) could transfer the demand from one direction to the other direction.

One important observation we make is that Steiner tree topologies can supply a lot of flexibility in avoiding routing congestion by applying way (2) and (3). For a multi-pin net, there are many different Steiner tree topologies to connect all the pins in the net. Each topology corresponds to some specific routing demand distribution. We notice that different topologies can have very different routing demand in two directions and in different rows/columns of global edges. For example, in Figure 5.4, we show 8 minimal wirelength Steiner tree topologies for a

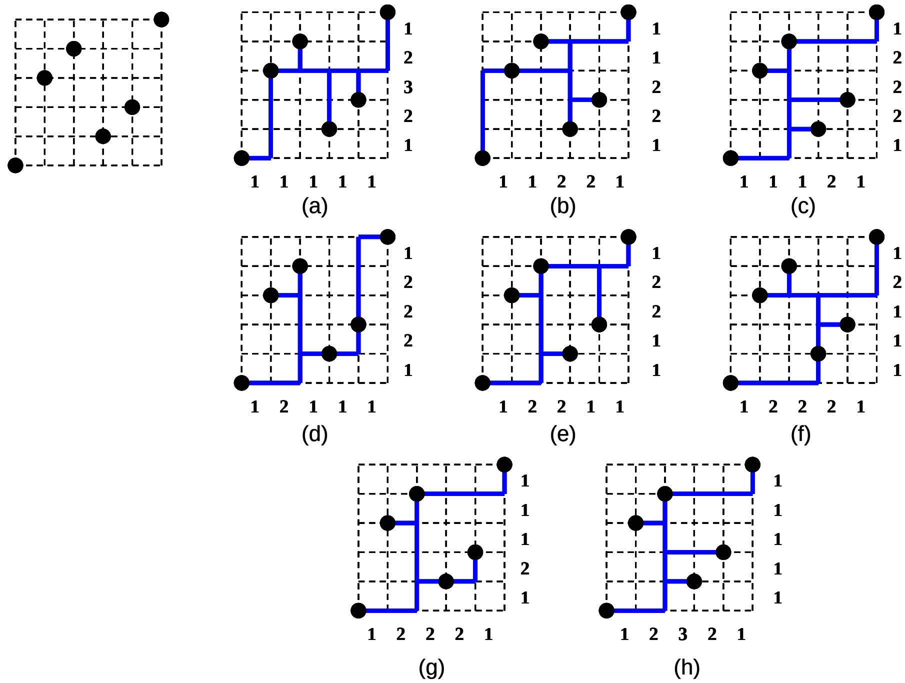

Figure 5.4 Different Steiner trees topologies

6-pin net. For each topology, we only show one of the possible embeddings on the routing grids. The number below each column of global edges is the routing demand over all the horizontal global edges in that column. The number right to each row of global edges is the routing demand over all the vertical global edges in that row. It is clear that although all these Steiner trees have the same wirelength, they have very different routing demand distribution, hence very different congestion results. Therefore, we can make use of this flexibility in topology and try to find good topology for each net in terms of congestion metric. For example, for the net shown in Figure 5.4, if it is congested in horizontal direction, we want to pick topology (a) which has less routing demand in horizontal direction. On the contrary, if it is congested in vertical direction, (h) would be the best choice. This applies way (3) of reallocating demand. In addition to transferring routing demand between two directions, way (2) of reallocating demand is also enabled by changing topology. Comparing topology (b) with (e), instead of having more routing demand in the 2nd row (from left) and 2nd column (from top) of global edges as in (e), topology (b) have more routing demand in the 4th row and 4th column of global edges. So whether use topology (b) or (e) depends on the congestion of these rows and columns of global edges.

With this flexibility of topology in mind, our main idea is to construct good Steiner tree topology for each net according to the congestion map. We encourage to use the topology with less routing demand in the congested direction, and also less routing demand in the congested regions. To achieve this goal, we construct Steiner tree topologies as follows. First, we define the row/column region between two Hanan grid lines [44] for a net as the rectangular region between the two grid lines and the bounding rectangle of the net. As illustrated in Figure 5.5, the shaded region in (a) is the row region between the Hanan grid lines G_{H1} and G_{H2}, and the distance between G_{H1} and G_{H2} is $v2$. Similarly, the shaded region in (b) is the column region between the Hanan grid lines G_{V1} and G_{V2}, and the distance between G_{V1} and G_{V2} is $h2$. For each row/column region between two hanan grid lines of the original net, we compute its corresponding "average congestion" (we will discribe how to compute it in detail later). Then, the distance between the corresponding two hanan grid lines is scaled proportional to the "average congestion". We use these scaled distances instead of their original distances to measure the routing tree wirelength. Hence, we transform the congestion-driven Steiner tree problem into a RSMT problem in scaled wirelength measure. Finally, we apply *FLUTE* to find the RSMT topology in terms of this scaled wirelength. This topology with minimal scaled wirelength leads to the best congestion result. In this way, we maintain a balance between wirelength and congestion when constructing the Steiner tree rather than just minimize wirelength.

So far we have presented the general flow to find a good topology. Now we describe what the "average congestion" for a row/column region is and how to compute it. For a row/column region between two Hanan grid lines, if it is congested in vertical/horizontal global edges, we discourage to use the segments across the region in the direction perpendicular to the two Hanan grid lines. Hence, we scale up the distance between the two Hanan grid lines. The scaling factor we use is the "average congestion". For a row/column region, it is defined as the ratio between the total demand and total capacity on all vertical/horizontal global edges in the

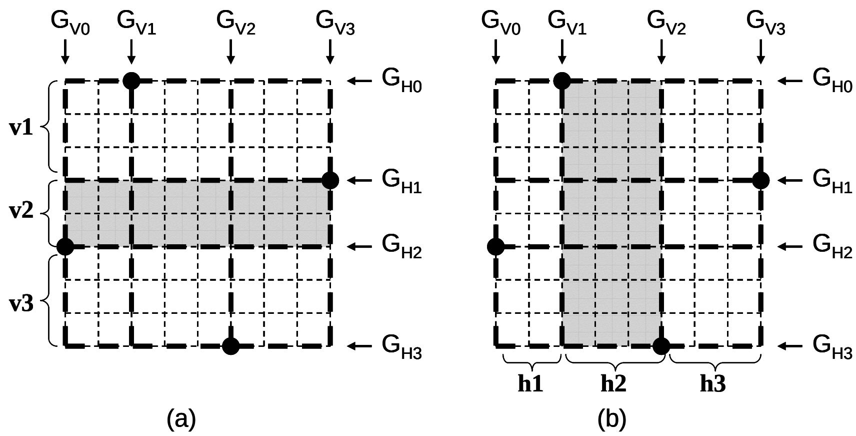

Figure 5.5 (a) The row region between G_{V1} and G_{V2}. (b) The column region between G_{H1} and G_{H2}

corresponding row/column region. "Average congestion" indicates on average how congested a row/column region is. For example, in Figure 5.5 (a), "average congestion" for the shaded row region is computed as the total demand divided by the total capacity on all vertical global edges in the region. Note that we are not just considering the global edges on Hanan grid, but all the global edges in this region because all these global edges are possibly used by our Steiner trees. In this technique, we only try to control the frequency to use different segments between Hanan grid lines in the topology but not the exact position of these segments in the Steiner tree. In fact, it is not necessary to specify the position of segments here. After we fix the Steiner tree topology in this phase, the segments still have a lot of flexibility to change locations. Hence, what we want is the "average" congestion for a row/column region instead of congestion on some specific global edges.

Finally, we want to point out that this congestion-driven Steiner tree construction technique has great impact on the routing solution quality. It explores the solution space out of the scope of pattern routing and maze routing.

5.5.2 Edge Shifting

In Section 5.5.1, we present the congestion-driven Steiner tree topology construction technique. The topology only specify the connections between the pins and Steiner nodes for the net. After fixing the topology, there is still flexibility left for congestion optimization. For example, in Figure 5.6, we focus on the bold edge in the Steiner tree. With different congestion scenarios, the edge should be shifted to different positions to avoid congestion.

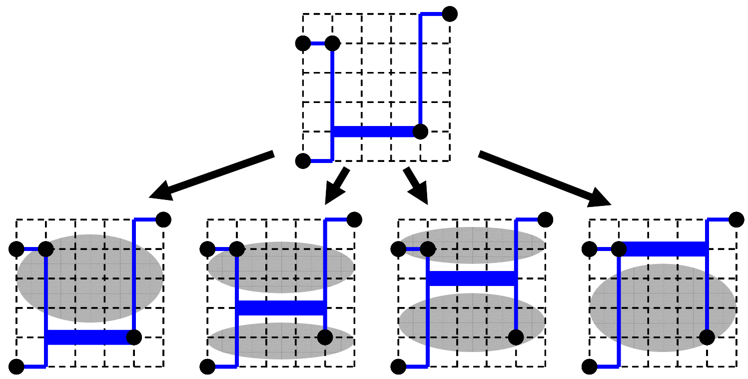

Figure 5.6 Edge Shifting for less Congestion

Bottom four cases: shaded regions are congested

If possible, we want to move tree edges out of congested regions without increasing Steiner tree wirelength. The reason is that the total wirelength is related to the overall congestion. If the total wirelength is more, it is very likely to have more overall congestion. We observe that if the two pins of a horizontal or vertical tree edge are both Steiner nodes, we can shift this edge freely within a "safe range" without increasing the Steiner tree length. In order to find the "safe range", for a horizontal/vertical edge between a pair of Steiner nodes S_1 and S_2, we define the "sliding range" as the range of y/x coordinates so that S_1 and S_2 will not pass any node (including pins and Steiner nodes) in the tree when shifting the tree edge S_1-S_2. As illustrated in Figure 5.7, the "sliding range" of (a) a horizontal edge, or (b) a vertical edge S_1-S_2 is R_{12}. We only consider shifting edge S_1-S_2 when both S_1 and S_2 have degree 3. A Steiner node can only have degree 3 or 4. For any degree 4 Steiner node, we can break it into two connected degree 3 Steiner nodes. The way to get this "sliding range" is as follows. We consider the two neighbors for S_1/S_2 which are not S_2/S_1. If S_1-S_2 is horizontal, the range for safely sliding S_1-S_2 is between the y coordinates of two neighbor nodes (R_1 and R_2 in Figure 5.7(a)). Otherwise, the range for safely sliding S_1-S_2 is between the x coordinates of two neighbor nodes (R_1 and R_2 in Figure 5.7(b)). The "sliding range" of S_1-S_2 is the common part of R_1 and R_2, which is R_{12} in Figure 5.7. In R_{12}, the edge S_1-S_2 can be shifted freely without changing tree length.

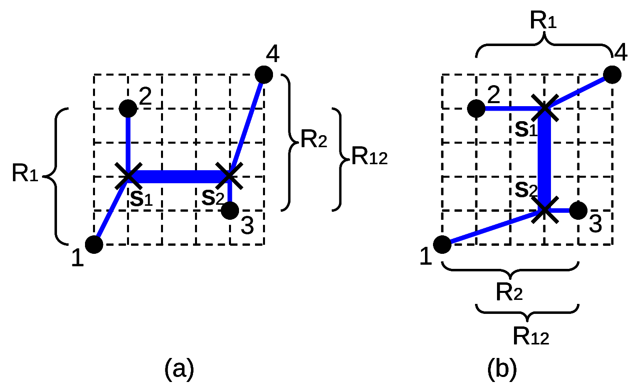

Figure 5.7 "Sliding range" for edge S_1-S_2

We want to point out that the "sliding range" may not always be the "safe range". Sometimes, it is just part of the "safe range". For example, in Figure 5.8(a), the "sliding range" for edge S_1-S_2 is R_{12}. Hence, S_1-S_2 can be shifted at most to the same y grid as Steiner node S_3. But we notice that S_1-S_2 can be shifted higher than S_3 without changing the Steiner tree length. The only problem here is that the tree topology needs to be changed. This happens when two Steiner nodes S_2 and S_3 overlap with each other (as illustrated in Figure 5.8(b)). In this case, we will exchange the two Steiner nodes S_2 and S_3 to enable further shifting, which is shown in Figure 5.8(c). Notice that by exchanging S_2 and S_3, we change the *topology*1 into *topology*2. In Figure 5.8(c), the "sliding range" for *topology*2 is R_{13}. The full "safe range" is R_{123}, which is the sum of R_{12} and R_{13}. Therefore, now we can explore the full "safe range" R_{123} for S_1-S_2.

After we find the "safe range" for an edge S_1-S_2, we need to decide the best position for it within the "safe range". The criterion for the best position is that the total demand of all the global edges on the Steiner tree is minimized. We define this total demand as the cost of the tree. Hence, for every possible position, we can evaluate this cost for the tree. Note that we only need to evaluate the demand on the global edges affected by shifting S_1-S_2, because other global edges will not be affected. So the edges need to be considered are all edges E adjacent

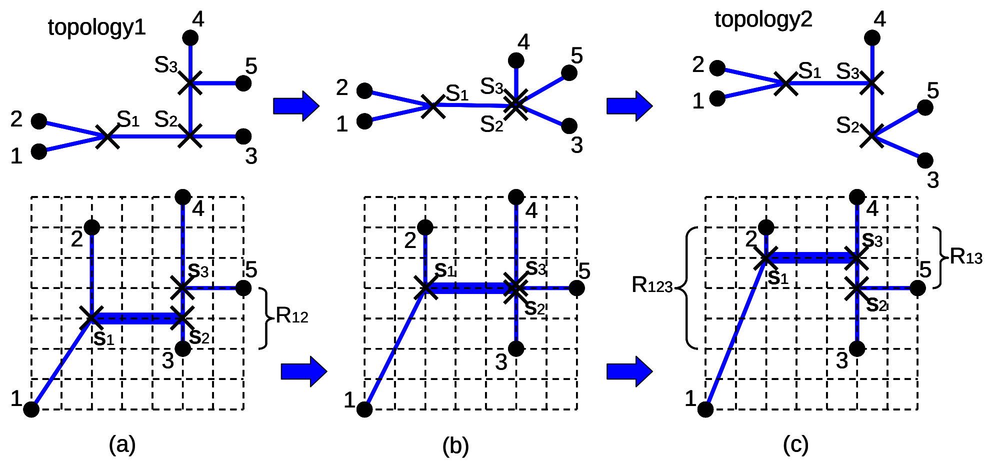

Figure 5.8 Modification of tree topology during edge shifting

to S_1 and S_2. Note that some edge $e \in E$, could be a diagonal edge (e.g., edge 1-S_1 in Figure 5.7(a)). We do not know which global edges this tree edge will use. In this case, we consider the two possible L-shape route for it and pick the one that results in smaller cost. The reason is that for these diagonal edges, later routing stages will try to minimize the total demand of global edges on their routing path. Of course, instead of considering two L-shape routes, we can consider Z-shape route or even maze route. It is a tradeoff between accuracy and runtime. Since the Steiner tree structures keep on changing at this stage, it is not necessary to consider the route too accurately.

The way to shift one tree edge is described above. For each Steiner tree, the algorithm to perform edge shifting is as follows. We find all the horizontal and vertical tree edges between two Steiner nodes in the Steiner tree. Next, we compute the "safe range" R_{12} for each tree edge S_1-S_2 (including the expanded range by exchanging Steiner nodes). Then the cost of the tree is evaluated for every possible position of S_1-S_2 within the "safe range". Finally, the tree edge S_1-S_2 and its position with the minimal cost is chosen and S_1-S_2 is shifted to that position. We iteratively apply this process until we cannot find a tree edge for shifting to further reduce the cost.

After *Edge Shifting*, the positions of Steiner nodes are fixed and the only flexibility left is how to route each tree edges.

5.5.3 Flow for Congestion-driven Steiner Tree Construction

Section 5.5.1 and 5.5.2 give the details of the techniques to construct good Steiner tree structure for a net. In this part, we present the flow of phase 2 to generate Steiner trees for all the nets.

We go through every net in the order in the netlist file. For each net N, we first remove its routing demand from the congestion map. Second, the Steiner tree topoloy for N is constructed as in Section 5.5.1. Then, we apply edge shifting technique in Section 5.5.2 to further reduce the congestion. After Steiner tree structures are fixed, we route all the tree edges using L-shaped pattern routing. Finally, we add new routing demand by N to the congestion map.

5.6 Pattern Routing and Maze Routing

After the congestion-driven Steiner tree construction phase, we find good Steiner tree structures for the nets. Then all routing trees are broken into tree edges (two-pin nets). In the routing phase, we route all two-pin nets by pattern routing and maze routing.

We first apply pure pattern routing to route all the two-pin nets once. Pattern routing is to use predefined patterns to route two-pin nets. The most commonly used are L-shaped (1-bend) or Z-shaped (2-bends) patterns. Pattern routing has much better runtime complexity over maze routing. The effect of pattern routing is investigated extensively in [46]. Here, we use the Z-shaped pattern. It has more flexibility than L-shaped pattern and much faster than maze routing. In fact, in the congestion-driven Steiner tree construction phase, we already perform L-shaped routing when we update the congestion map after constructing Steiner tree for a net.

After the pattern routing, we apply rip-up and reroute using maze routing, which is similar to other works. Many recent global routers [46][51] have routing cost which increases abruptly when the demand on a global edge reaches the edge capacity (Figure 5.9(a)). In [47], the routing cost function for maze routing is discussed and a piece-wise cost function is proposed. A unit cost is assigned to a global edge until it reaches a certain percentage below capacity, and cost is increased linearly until it reaches a certain percentage above capacity (Figure 5.9(b)).

Instead, we employ a logistic function [60] in equation (5.1) as our cost function (Figure 5.9(c)). h and k are function parameters.

$$cost = 1 + \frac{h}{1 + e^{-k(demand - capacity)}} \tag{5.1}$$

The reason for us to use such an function is that we want the cost to increase dramatically around the capacity but mildly in the under-capacity and over-capacity part. The idea behind this is to differentiate the slope of cost function in different parts. If demand on a global edge is much lower than capacity, we do not need to differentiate different demand values, e.g., if the capacity is 10, the difference in cost for demand 2 and 3 should be small. Similarly, if demand on a global edge is much higher than capacity, we do not need to charge very different cost for different demand values, either, e.g., demand 20 or 25 should not make significant difference when capacity is 10. However, if demand on a global edge is close to capacity, the change on demand make significant difference because the edge could become over capacity from within capacity, or from within capacity to over capacity. In this way, we focus more on the global edges with demand close to capacity.

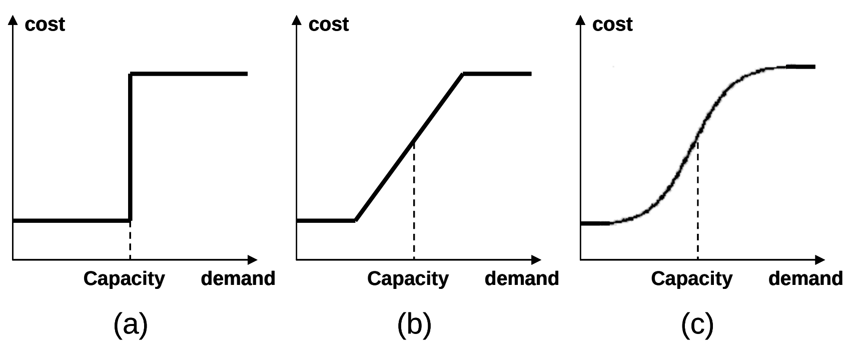

Figure 5.9 (a) Abrupt cost, (b) Linear cost, (c) Logistic cost

For *FastRoute* default mode, we only do one round of maze routing and on average only 2.15% of nets are really routed by maze routing (others use pattern routing). This is the major reason that our algorithm is so fast. Moreover, we can get better total overflow than other global routers although we do much less maze routing. We attribute this to the high-quality Steiner tree structures generated by the second phase. Maze router has a very good starting solution to work with.

5.7 Experimental Results

In this section, we present our experimental results. All experiments were performed on a Linux workstation with Intel Pentium 4 3.0 GHz CPU and 2GB memory.

First, we compare *FastRoute* with two state-of-the-art academic global routers: *Labyrinth* [46] and *Chi Dispersion* router [47]. We use the same benchmarks as in [47] provided by the authors of [46]. For *Labyrinth*, 70% of the shortest connections are routed by pattern routing, which is the same as in [47]. We measure wirelength and total overflow in the manner suggested by the authors of both papers. The results are summarized in Table 5.2. The total overflow and wirelength of *FastRoute* is less than both *Labyrinth* and *Chi Dispersion* router. At the same time, *FastRoute* is 132× and 64× faster than *Labyrinth* and *Chi Dispersion* router, respectively. Because we cannot find a version to duplicate the results in [47], the runtime of *Chi Dispersion* router is scaled from the runtime in [47] based on the information from Standard Performance Evaluation Corporation (SPEC) [61]. In [47], it was claimed that runtime of *Chi Dispersion* router is roughly 2× faster than *Labyrinth*, which coincides with the scaled runtime we obtained. We also get a new version of *Chi Dispersion* router from the authors of [47], the total overflow on the same set of benchmark is 804, but the total runtime is 917 seconds which is close to the runtime of *Labyrinth*. We also have a beaver mode for *FastRoute*. It performs several rounds of rip-up and reroute to achieve lower overflow. It can cut down the total overflow by half with 2.2× runtime of the default mode.

Second, we investigate the effect of three main techniques in *FastRoute*: congestion-driven Steiner tree construction, edge shifting and logistic cost function for maze routing. We disable the three techniques from *FastRoute* one by one and compare the final total overflow with *FastRoute*. For the logistic cost function, we substitute it with a linear cost function proposed in[47] and tried to tune the parameters in the linear cost function to get results as good as possible. From Table 5.3, the total overflow are increased by 51%, 39% and 78% without the three techniques, respectively. It is clear that all of them contribute to the high quality of

Table 5.2 Comparison between FastRoute, Labyrinth and Chi Dispersion router

	FastRoute			FastRoute (Beaver mode)			Labyrinth Predictable router			*Chi* Dispersion router		
	Overflow	Wirelen	Time(s)	Overflow	Wirelen	Time(s)	Overflow	Wirelen	Time(s)	Overflow	Wirelen	Time(s)
ibm01	250	67128	0.21	159	68436	0.72	242	76228	16.99	189	66005	8.63
ibm02	39	179995	0.56	3	180139	1.16	214	202235	26.53	64	178892	26.27
ibm03	1	151023	0.43	1	151023	0.43	117	191500	37.92	10	152392	24.71
ibm04	567	172593	0.50	300	175219	2.30	786	198181	80.95	465	173241	32.94
ibm06	33	285882	0.91	7	287870	1.71	130	339379	72.06	35	289276	53.33
ibm07	18	376835	1.05	2	379989	1.99	407	450855	168.41	309	378994	79.61
ibm08	58	412915	1.16	17	414909	3.17	352	466556	154.82	74	415285	72.94
ibm09	28	426471	1.39	22	428803	2.75	310	481841	229.59	52	427556	86.67
ibm10	18	599433	1.98	1	600321	3.80	288	680113	296.70	73	599937	139.61
Total	1012	2672275	8.19	512	2686709	18.03	2846	3086888	1083.97	1271	2681578	524.71
Norm[1]	**1**	**1**	**1**	**0.506**	**1.005**	**2.201**	**2.812**	**1.155**	**132**	**1.256**	**1.003**	**64**

1. Normalized to FastRoute results.

Table 5.3 Effect of Congestion-driven Steiner tree topology construction, Edge shifting and Logistic cost function

	FastRoute Overflow	w/o StTree Overflow	w/o Edgeshift Overflow	Linear Cost Overflow
ibm01	250	283	323	297
ibm02	39	114	57	108
ibm03	1	5	1	30
ibm04	567	672	666	606
ibm06	33	71	85	129
ibm07	18	178	54	174
ibm08	58	91	89	126
ibm09	28	74	89	113
ibm10	18	39	38	220
Total	1012	1527	1402	1803
Norm[1]	**1**	**1.51**	**1.39**	**1.78**

1. Normalized to FastRoute total overflow

FastRoute.

Third, we show the runtime breakdown for *FastRoute* default mode. As shown in Table 5.4, the three phases in *FastRoute*: congestion map generation, congestion-driven Steiner tree topology construction, and two-pin nets routing account for 14.4%, 27.5% and 58.1% of the total runtime, respectively. In addition, maze routing in two-pin nets routing is still the most time-consuming part (48% of total runtime) although on average only 2.15% two-pin nets are routed using maze routing. Consider that *Labyrinth* apply maze routing on 30% of the nets and do many rounds of rip-up and reroute. That is why *FastRoute* can be two orders faster.

Fourth, we compare the runtime of *FastRoute* and an efficient congestion estimator *FaDGloR*. In [48], the authors claimed *FaDGloR* is as fast as probabilistic congestion estimators. *FaDGloR* reports two runtime, "total runtime" (the total runtime including Steiner tree construction, decomposition, routing, file I/O, and result checking) and "route time" (the actual routing time for all two-pin nets). Hence, we do two type of comparison here. First, we compare the *FastRoute* total runtime with the "total runtime" of *FaDGloR*. Table 5.5 shows that *FastRoute* is about 3.13× faster than *FaDGloR*. Since we should exclude the file I/O

Table 5.4 Runtime breakdown for FastRoute

	Cong Map	Steiner Tree	Route two-pin nets	
			Pattern Route	Maze Route
ibm01	14.3%	23.8%	4.8%	57.1%
ibm02	12.5%	25.0%	7.1%	55.4%
ibm03	13.6%	27.3%	11.4%	47.7%
ibm04	14.0%	22.0%	12.0%	52.0%
ibm06	13.2%	30.8%	8.8%	47.3%
ibm07	16.3%	28.8%	12.5%	42.3%
ibm08	17.2%	36.9%	11.5%	34.4%
ibm09	14.1%	24.6%	12.0%	49.3%
ibm10	14.1%	28.1%	11.1%	46.7%
avg	**14.4%**	**27.5%**	**10.1%**	**48.0%**

and result checking parts from *FaDGloR* "total runtime", the real speedup should be around 3×. Considering their Steiner tree construction algorithm is much slower than FLUTE, we perform a second comparison. We report the *FastRoute* runtime excluding the Steiner tree construction in phase 1 (FastRoute(-rsmt)) and compare it with the "route time" of *FaDGloR* (FaDGloR(-rsmt)). *FastRoute* is still 14% faster than *FaDGloR* for the routing time and has better scalability. But note that *FastRoute* generates high-quality global routing solutions while *FaDGloR* only gives congestion estimation.

Fifth, we also run state-of-the-art placers *Capo9.1* [10] and *Dragon 3.01* [29] on the placement benchmarks from which the global routing benchmarks are generated. Table 5.6 show that *FastRoute* runtime is only about 1/934 and 1/2229 of the runtime of *Capo9.1* and *Dragon3.01*. This means we can run *FastRoute* hundreds of times inside placers without much runtime penalty.

Table 5.5 FastRoute and FaDGloR Runtime Comparison

	FastRoute	FaDGloR[1]	FastRoute(-rsmt)	FaDGloR(-rsmt)
ibm01	0.21	0.71	0.20	0.17
ibm02	0.56	2.18	0.52	0.45
ibm03	0.43	1.36	0.41	0.46
ibm04	0.50	1.54	0.47	0.48
ibm06	0.91	2.27	0.86	0.74
ibm07	1.05	3.08	0.99	1.01
ibm08	1.16	4.35	1.07	1.13
ibm09	1.39	4.31	1.32	1.86
ibm10	1.98	5.86	1.88	2.49
Total	8.19	25.66	7.72	8.79
Norm	**1**	**3.13**[2]	**1**	**1.14**[3]

The unit for all runtime (except Normalized) is second. 1. This runtime include file I/O and result checking time, 2. normalized to full FastRoute runtime, 3. normalized to FastRoute(-rmst) runtime.

Table 5.6 Runtime comparison with Placers

	FastRoute Time(s)	Capo Time(s)	Dragon Time(s)
ibm01	0.21	126	778
ibm02	0.56	280	663
ibm03	0.43	338	633
ibm04	0.50	456	1234
ibm06	0.91	666	1392
ibm07	1.05	1145	1904
ibm08	1.16	1277	4163
ibm09	1.39	1329	3953
ibm10	1.98	2035	3537
Total	8.19	7652	18257
Norm	**1**	**934**	**2229**

CHAPTER 6. FASTROUTE 2.0 - AN IMPROVED GLOBAL ROUTER

6.1 Introduction

As feature size in advanced VLSI technology continues to shrink, interconnect delay has become the dominant factor in circuit delay. Although the scaling of feature size makes the device smaller and faster, interconnect delay is not scaling down as device delay. Many recent articles reported that interconnect delay can consume as much as 75% of clock cycle in modern designs. Hence, the performance of current designs is mainly determined by interconnect instead of device. In addition, because of the shrinking of device size, the chip area is no longer determined by total cell area, but by the limited routing resources. Extra "white space" is commonly added to provide enough wire tracks to resolve routing congestion. It is typical that more than half of the modern chip is occupied by white space.

Although interconnect is not implemented until the routing stage, its importance makes it necessary to be dealt with in early design stages such as floorplanning and placement. One reason is that floorplanning and placement decides the length and hence the delay of interconnect wires to a large extent. The other is that the white space needs to be allocated appropriately before the routing stage to ensure the routability. Generally speaking, the placement obtained by the design stages before routing determines the solution space for the router to explore. For a bad placement, no matter how good the router is, it is impossible to achieve a good design.

In order to consider the interconnect in early design stages without routing information, many interconnect models are employed to estimate timing and routing congestion for interconnect. To estimate timing, interconnect is modeled by half-perimeter of the bounding box [13] [54] or a star [55] to compute the delay from source to sinks. However, considering the real implementation, multi-pin nets are typically routed as Steiner trees. Hence, both half-perimeter of the bounding box and star-model is far from accurate for interconnect timing estimation. For routing congestion, post-placement congestion estimation methods try to predict the routing congestion for a given placement. In recent years, a number of probabilistic methods for congestion estimation have been proposed [56][57][58]. Recently, Westra et al. [48] presented a new technique based on degenerate global routing techniques. All these works proposed generic estimators which aim at predicting the behavior of all routers consistently. However, as pointed out in [62], because routing solutions generated by different routers are very different, it is not possible for an estimator to accurately predict congestion of all routers. Furthermore, even a real global router cannot predict the routing congestion for solutions obtained by another global router. Thus, in both timing and congestion estimation, the interconnect models are far from the real implementation in the routing stage. The interconnect resources required by routing stage are not adequately estimated and reserved during early design stages.

In order to get accurate interconnect information in early design stages, it is desirable to incorporate global routing into them. Global routing allocates the routing demand globally over the chip area. It generates interconnect information very close to the final routing implementation and can be used for accurate estimation of interconnect topology, wirelength, delay, congestion, buffering solution, etc. In addition, if the same global router is used for both early stage interconnect estimation and global routing, the inconsistency between the early design stages and routing can be eliminated.

There are mainly two categories of global routing techniques: rip-up and reroute based techniques, and multicommodity flow based techniques. Many academic routers [46][47] and the majority of the industry routers employ the rip-up and reroute approach. This kind of techniques are essentially sequential routing methods in which each net is routed in a certain order according to the routing congestion from nets already routed. The multicommodity flow based techniques [51][63] can handle simultaneous routing of multiple nets as a multicommodity flow problem. The main idea is to model nets as different commodities that flow through the network of routing resource graph. The flow problem is typically solved by linear programming which results in fractional flow. Therefore, a randomized rounding procedure is used to discretize the solution. Albrecht [51] proposed a method to approximate the LP solution with provable error bounds to speed up the computation.

In order to handle large size problems, multilevel routing approaches [65][66] are proposed to reduce the complexity of the problem. A ”V-shaped” recursive coarsening and refinement process is commonly used.

However, due to the high runtime complexity of the traditional global routers, it is impractical to perform global routing repeatedly in early stages. In Chapter 5 an extremely fast global router, *FastRoute* [62] was proposed to address the runtime issue. Unlike many global routers which rely on maze routing to resolve the congestion, *FastRoute* focuses on determining good Steiner tree topology and Steiner node locations according to congestion information so that much less maze routing is needed. Experimental results show that *FastRoute* can generate less congested global routing solutions with two orders of magnitude speedup over the state-of-the-art academic global routers *Labyrinth* [46] and *Chi Dispersion* router [47]. And it is even faster than the highly-efficient congestion estimation algorithm *FaDGloR* [48].

In this chapter, we propose two major techniques to futher improve *FastRoute* in solution quality.

- A monotonic routing technique to substitute pattern routing.
- A multi-source multi-sink maze routing technique.

The new router is called *FastRoute* 2.0.

On the same set of benchmarks in [62][47], *FastRoute* 2.0 achieves much better solution quality than *FastRoute*, *Labyrinth* and *Chi Dispersion* router . The total overflow is reduced by more than an order of magnitude. The runtime is about 73% slower than the extremely fast *FastRoute*, but still 78× and 37× faster than *Labyrinth* and *Chi Dispersion* router.

The remainder of the chapter is organized as follows. In Section 6.2, we review the framework and techniques of *FastRoute* global router. In Section 6.3, we present the two major techniques in detail. In Section 6.4, experimental results of *FastRoute* 2.0 and comparison with three state-of-the-art global routers are shown.

6.2 Review of FastRoute Global Router

In this section, we briefly review the extremely fast global router, *FastRoute* [62].

Different from traditional global routers, *FastRoute* is a global router aiming at the application in both placement and routing. In placement process, global router may be invoked many times to get the interconnect estimation for intermediate placement. Hence, the runtime is a major concern of the algorithm. As pointed out by many works (e.g, [46]), maze routing is the major contributor of global routing runtime. Therefore, *FastRoute* focuses mainly on the Steiner tree construction to alleviate the burden of maze routing. Because of the good Steiner tree structures obtained, *FastRoute* only runs one round of maze routing and only about 2.15% of 2-pin nets are routed by maze routing. This is the major reason why *FastRoute* can achieve such a significant speedup over other global routers.

FastRoute has three phases:

1. *Congestion map generation:* In this phase, the Steiner trees for all the nets are generated using minimal Steiner tree algorithm *FLUTE* [38]. Then all Steiner trees are broken into 2-pin nets and routed using L-shaped pattern routing. The congestion map is obtained from this rough routing result.

2. *Congestion-driven Steiner tree construction:* In this phase, two major techniques are proposed to construct good Steiner tree structures to reduce the routing congestion. First, a congestion-driven topology generation algorithm generates the Steiner tree topologies to reduce routing congestion according to the congestion map. The algorithm extends the idea of *FLUTE* to handle the congestion by trying to use less wires in the congested region. Second, an edge shifting technique is employed to further reduce the routing congestion after the Steiner tree topology is fixed. It identifies the tree edges that can be shifted without changing the rectilinear wirelength of the tree. By shifting these edges, routing demand can be shifted from congested region to uncongested region so that local congestion can be resolved. Both techniques are applied to every net with more than 4 pins, and the congestion map is updating as each net changes.

3. *Routing of 2-pin nets using pattern routing and maze routing:* In this phase, the Steiner trees obtained from phase 2 are broken into 2-pin nets. Then every 2-pin net is ripped up and rerouted by Z-shaped pattern routing. Finally, the long 2-pin nets over the congested regions is ripped up again and rerouted by maze routing. A cost function based on logistic function [60] is introduced to direct the maze routing to find less congested paths.

FastRoute achieves good global routing solutions with two orders of magnitude faster runtime over other state-of-the-art academic global routers. The extremely high speed makes it possible to incorporate it directly into the early design stages without much runtime penalty. This could dramatically improve the solution quality because accurate interconnect information becomes available in early stages.

6.3 New Routing Techniques

The original *FastRoute* focuses on generating good Steiner tree structures to reduce routing congestion. So we follow the first two phases of *FastRoute* to generate high-quality Steiner tree structures. However, we demonstrate that the pattern routing and maze routing in the third phase can be improved to obtain better routing solutions. In this work, we propose the following two techniques:

- A monotonic routing to substitute the pattern routing in *FastRoute* flow.
- A multi-source multi-sink maze routing technique which is a more powerful maze routing technique to achieve high-quality routing solutions.

In 6.3.1 and 6.3.2, we will describe the two techniques in detail. Finally, the flow of *FastRoute* 2.0, the new global router based on the two techniques is given in 6.3.3.

6.3.1 Monotonic Routing

Pattern routing uses predefined patterns to route 2-pin nets. Usually, the most commonly used are L-shaped (1-bend) or Z-shaped (2-bends) patterns. Because pattern routing limits the pattern of routing path shapes, it can speed up the global routing process. Therefore, pattern routing is typically employed to route a big portion of nets to save runtime. In *FastRoute*, after every Steiner tree broken into 2-pin nets, Z-shaped pattern routing is used to route each 2-pin net.

Although the pattern routing can speed up the routing process, its quality could be much worse than maze routing. The maze router ensures that the least cost route is found, but pattern routing only considers a small portion of possible routes. For a 2-pin net which spans $m \times n$ grids, L-shaped pattern routing only considers 2 different paths, and Z-shaped pattern routing only considers $m + n$ different paths. Hence, pattern routing fails to find good routing paths to avoid the congestion in many cases. We want to find a trade-off between maze routing and pattern routing so that the quality can be better than pattern routing, but the runtime is close to it.

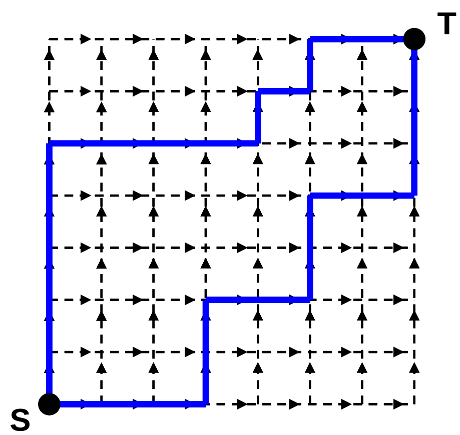

Figure 6.1 Monotonic routing paths

The basic idea is to find the best monotonic routing path for a 2-pin net. Let one pin be the source (S) and the other be sink (T). A monotonic routing path from S to T is a path on the routing grid from S to T which always directs toward T. Figure 6.1 shows two different monotonic routing paths from S to T. Notice that all monotonic routing paths will not go out of the bounding box of S and T. The total number of monotonic routing paths from one corner to the diagonal corner of a $m \times n$ grids is $\binom{m+n-2}{m-1} = \frac{(m+n-2)!}{(m-1)!(n-1)!}$.

One important property of monotonic routing path is that for every grid point within the bounding box of S and T, only one or two grid points can be its predecessor on any monotonic routing path from S to it. As shown in Figure 6.2, all the grid points G (empty circles) with the same x-coordinates or y-coordinates as S have only one predecessor G_1, and all other grid points g (solid dots) have two predecessors g_1 and g_2. Without loss of generality, we assume S is at the lower-left corner of the bounding box and T is at the upper-right corner.

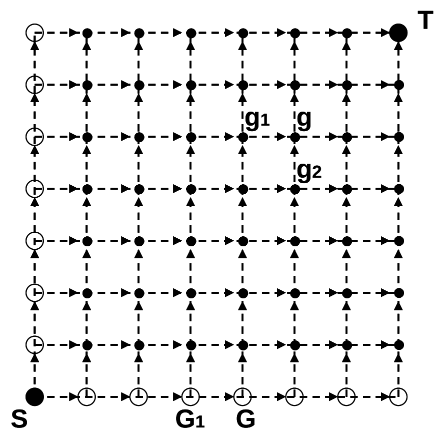

Figure 6.2 Monotonic path property

For the least monotonic path from S to G and S to g, we have the following two lemmas.

Lemma 7 *The least cost monotonic routing path from S to G, $P_{SG} = P_{SG1} + (G_1, G)$, where P_{SG1} is the least cost monotonic routing path from S to G_1.*

Lemma 8 *Let P_1 and P_2 are the least cost monotonic routing paths from S to g_1 and g_2, respectively. The least cost monotonic routing path from S to g, P_{Sg} is one of the following two paths, whichever has less cost. (1) $P_1 + (g_1, g)$, (2) $P_2 + (g_2, g)$.*

These two lemmas follow the truth that every monotonic routing path from S to a grid point must be composed of a monotonic routing path from S to its predecessor(s) and the edge between it and its predecessor(s). The two lemmas told us that if the least cost monotonic routing path(s) for the predecessor(s) is found, the least cost monotonic routing path for the current grid point can be found. Hence, we can use dynamic programming to find the least cost monotonic routing path from S to T. The algorithm is shown in Figure 6.3. Lemma 1 and Lemma 2 ensure the optimality of the algorithm.

Algorithm Monotonic Routing

1. $d(S) = 0$
2. **for** $x = 1$ to m
3. $G = (x, 0)$, $G_1 = (x-1, 0)$
4. $d(G) = d(G_1) + cost(G, G_1)$, $\pi(G) = G_1$
5. **for** $y = 1$ to n
6. $G = (0, y)$, $G_1 = (0, y-1)$
7. $d(G) = d(G_1) + cost(G, G_1)$, $\pi(G) = G_1$
8. **for** $x = 1$ to m
9.   **for** $y = 1$ to n
10. $g = (x, y)$
11. $g_1 = (x-1, y)$, $g_2 = (x, y-1)$
12.   **if** $d(g_1)+cost(g, g_1) < d(g_2)+cost(g, g_2)$
13. $d(g) = d(g_1)+cost(g, g_1)$, $\pi(g) = g_1$
14.   **else**
15. $d(g) = d(g_2)+cost(g, g_2)$, $\pi(g) = g_2$
16. Trace back from T using π to find the least cost monotonic path

Figure 6.3 Monotonic routing algorithm

In the algorithm, $S = (0, 0)$, $T = (m, n)$, and $d()$ is the cost of the least cost monotonic path from S to any grid point in the bounding box of S and T. Lines 2-7 finds the least cost for all grid points represented as empty circles in Figure 6.2. Lines 8-15 finds the least cost for all grid points represented as solid dots in Figure 6.2, including T. Note that whenever we update d for a grid point, d for its predecessor(s) is already available.

Now we analyze the complexity of the algorithm. Lines 2-4 takes $O(m)$ time, lines 5-7 takes $O(n)$ time, lines 8-15 takes $O(mn)$ time, and line 16 takes $O(m + n)$ time. Hence, the runtime complexity of the algorithm is $O(mn)$, which is the same as Z-shaped pattern routing. In Section 6.4, experimental results show that monotonic routing is about $2.3\times$ slower than Z-shaped pattern routing.

6.3.2 Multi-source Multi-sink Maze Routing

Maze routing is the most popular technique used in global routing. Originally, maze routing algorithm is designed to find the shortest path connecting two pins in the presence of routing blockages. Later, it has been extended to find a path connecting two pins in such a way that it favors a path that passes through less congested area according to some cost function. It is a very powerful technique to find paths avoiding congestion.

However, we notice that the application of maze routing in global routing is to find path between two pins. For multi-pin nets, a typical way is to break the routing tree into edges (2-pin nets), and route each edge by maze routing. We find that this kind of independent edge-by-edge routing scheme may cause problems and fail to generate good routing solutions for the multi-pin nets. Figure 6.4 illustrates three different scenarios. The shaded areas denote the congested regions.

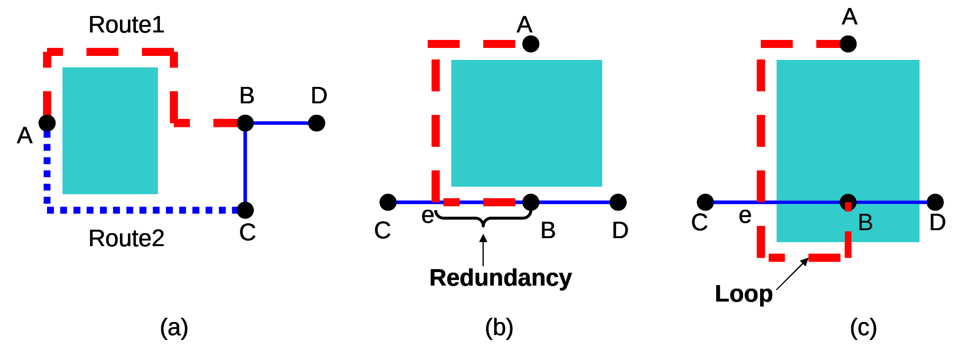

Figure 6.4 Maze routing scenarios

- *Unnecessary detour:* Consider the scenario in Figure 6.4 (a). The dashed route "Route1" is the maze routing result for edge (A, B). However, if the path does not need to go from A to B, "Route2" is a better choice in terms of cost.

- *Redundant routing:* Consider the scenario in Figure 6.4 (b). The dashed route is the maze routing result for edge (A, B). However, the (e, B) part on the path is already part of the routing tree, and it is redundant to repeat it.

- *Unintentionally loop:* Consider the scenario in Figure 6.4 (c). The dashed route is the maze routing result for edge (A, B). A loop is created in the routing tree. It is obvious that this loop is not needed and only the part from A to e is necessary on the path.

As we can see in these three scenarios, unnecessary wires are used in routing the multi-pin nets. This results in using more routing resources than necessary and cause routing congestion. The major defect of this edge-by-edge routing scheme is that the tree information is neglected and every edge is routed independently. When routing an edge in the tree for multi-pin nets, the routing path has to start with one endpoint of the edge and end with the other endpoint. However, this may not be necessary sometimes. It is enough if there is a path created between the two endpoints, no matter it directly go from one endpoint to the other or use part of routing tree already there.

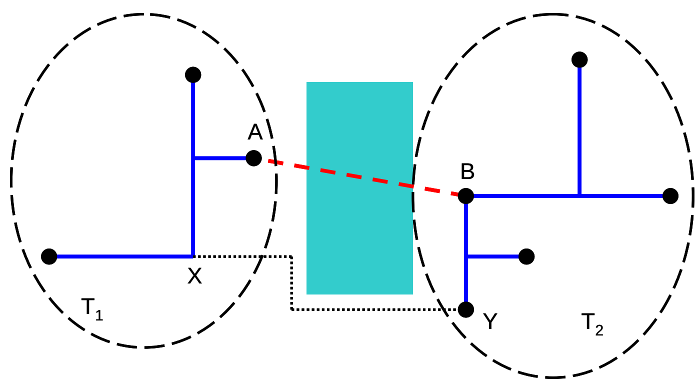

Figure 6.5 Multi-source multi-sink maze routing

Aware of the problem, we propose a multi-source multi-sink maze routing algorithm. The main idea is that the routing tree is respected when we route an edge for a multi-pin net. We do not constrain the two endpoints of the routing path to be the original endpoints of the edge being routed. As illustrated in Figure 6.5, suppose we are routing an edge (A, B) in the routing tree T for a multi-pin net N. We first remove (A, B) from T and obtain two subtrees T_1 and T_2. (Note that T_1 and T_2 can be just a point.) We treat all the grid points on T_1 as sources, and all the grid points on T_2 as sinks. Then, we apply the multi-source multi-sink maze routing to find the best path connecting T_1 and T_2 to form a tree. In Figure 6.5, the dotted line from X to Y is the best path to connect T_1 and T_2.

Our multi-source multi-sink maze routing algorithm is shown in Figure 6.6. In the algorithm, we use the same cost function as in *FastRoute* [62]. $d(g)$ is the distance from T_1 to g, defined as the total cost of all global edges passed by the temporary shortest path from T_1 to g. The algorithm follows the framework of Dijkstra's algorithm [64]. Lines 1-5 initializes the distance d, priority queue Q and destination points. Lines 6-17 is the loop similar to Dijkstra's algorithm. Line 18 just traces back to find the shortest path from T_1 to T_2. Note that Dijkstra's algorithm ensures that when a point u is extracted from Q, $d(u)$ is the shortest distance from T_1 to u. That is why the stopping criterion is when the first destination point is extracted from the priority queue.

Algorithm Multi-source Multi-destination Maze Routing

1. $d(g) = \infty$ for all grid points g
2. Find subtree T_1 (contains A) and T_2 (contains B) after breaking (A, B)
3. Set $d(u) = 0$ and $\pi(u)$ = nil, for all grid points u on T_1
4. Set up a priority queue Q with all grid points on T_1
5. Mark all grid points on T_2 as destination point
6. $u \leftarrow$ Extract-Min(Q)
7. **While** u is not destination point
8.   **do**
9.   **for** each neighbor grid points v of u
10.   **do**
11.   **if** $d(v) > d(u) + cost(u, v)$
12.   **then** $d(v) = d(u) + cost(u, v)$
13. $\pi(v) = u$
14.   **if** v is in Q
15.   **then** update Q
16.   **else** insert v into Q
17. $u \leftarrow$ Extract-Min(Q)
18. Trace back from u using π to find the shortest path from T_1 to T_2

Figure 6.6 Multi-source multi-sink maze routing algorithm

Our algorithm finds the least cost routing path from T_1 to T_2. Theorem 1 gives the optimality of the algorithm.

Theorem 1 The path found by multi-source multi-sink maze routing algorithm is the least cost routing path from T_1 to T_2.

Proof First of all, note that the cost function cost(u, v) is a positive function in our problem. In line 3, $d(u) = 0$ for all the grid points on T_1. Hence, we can assume a supersource which replaces all the grid points on T_1, and all grid points adjacent to T_1 are its neighbor. Similarly, we can assume a supersink which replaces all the grid points on T_2, and all adjacent grid points to T_2. Then the problem is transformed to a single-source, single-sink shortest path problem. The optimality follows the optimality of Dijkstra's algorithm.

The only thing left is to prove the stopping criterion is correct. Recall that we stop when a destination point on T_2 is extracted from Q. Assume u is the first destination point extracted from Q. For the purpose of contradiction, let w be the destination point which is on the shortest path from T_1 to T_2. Hence, we have $d(w) < d(u)$. However, when we extract u from Q, w is still in Q, which means $d(w) \geq d(u)$. Because the cost function is positive, $d(w)$ will never decrease in later updating. Therefore, we obtain a contradiction that $d(w) \geq d(u)$. ∎

Now we analyze the complexity of the algorithm. Assume there are V grid points in the search region. Lines 1-5 takes time $O(V)$. Each Extract-Min operation on the priority queue Q takes time $O(lgV)$. There are at most V iterations for the while loop. For each u, there are at most 4 neighbors adjacent to it. The insertion and updating of Q takes time $O(lgV)$. The total complexity is therefore $O(VlgV)$.

We apply this multi-source multi-sink maze routing algorithm on the tree edges of multi-pin nets. The runtime of maze routing algorithm is highly related to the size of the search region. In order to speed up the algorithm, we do not always search the whole grid graph to find the least cost path. Instead, we expand each boundary of the bounding box by a certain amount, say w rows and h columns, and use this enlarged region as the search region for the maze routing algorithm. By using this kind of search region, the runtime can be reduced significantly but the solution quality is close to optimal. In our implementation, the enlarge value is 10 for all four boundaries in the first maze routing round. If more rounds are needed, the enlarge value is increased by 10 every round.

We want to point out one issue for the multi-source multi-sink maze routing technique. It can totally change the routing tree structure because the endpoints of new routing path do not need to be the endpoints of the edge being routed. For example, in Figure 6.7, the Steiner tree structure is changed from (a) to (b) because of the new routing of edge (A, B). Hence, we need to update the Steiner tree structure accordingly after routing each edge by multi-source multi-sink maze routing.

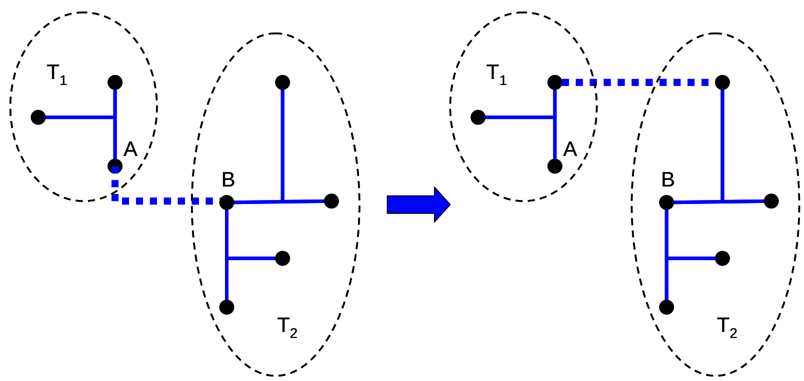

Figure 6.7 Steiner tree topology changed by maze routing

6.3.3 Flow of FastRoute 2.0

In this part, we give the flow of the new router, *FastRoute* 2.0. In general, *FastRoute* 2.0 has the flow similar to *FastRoute*. First, the congestion map is generated. Second, Steiner tree structures are constructed according to the congestion map. Finally, monotonic routing and multi-source multi-sink maze routing are applied to route the tree edges in Steiner routing trees.

The first two phases are the same as *FastRoute*. In the final phase, we first apply the monotonic routing to every edge in every routing tree. Then we run a round of multi-source multi-sink maze routing. However, in this maze routing round, we are not routing every edge by maze routing. Instead, only the edges longer than a threshold and across congested areas will be routed by maze route, and other edges are routed by monotonic routing. The intuition is to avoid routing short nets and long nets not passing congested area by maze routing. Otherwise, unnecessary detours may be created to use more wirelength and cause routing congestion. Of course, another important reason is to cut down the runtime of maze routing. If there is still a lot of overflow, we will run more rounds of maze routing.

6.4 Experimental Results

In this section, we present the experimental results. All experiments were performed on a Linux workstation with Intel Pentium 4 3.0 GHz CPU and 2GB memory.

Table 6.1 Global Routing Benchmark statistics

	Grids	# Nets	# Routed Nets	Max Deg	Avg Deg
ibm01	64x64	11.5k	9.1k	37	3.8
ibm02	80x64	18.4k	14.3k	126	4.4
ibm03	80x64	21.6k	15.3k	49	3.6
ibm04	96x64	26.2k	19.7k	41	3.4
ibm06	128x64	33.4k	25.8k	34	3.8
ibm07	192x64	44.4k	34.4k	22	3.8
ibm08	192x64	47.9k	35.2k	65	4.3
ibm09	256x64	50.4k	39.6k	38	3.8
ibm10	256x64	64.2k	49.5k	32	4.2

First, we compare *FastRoute* 2.0 with three state-of-the-art academic global routers: *FastRoute* [62], *Labyrinth* [46] and *Chi Dispersion* router [47]. We use the same benchmarks as in [47] provided by the authors of [46]. Statistics of the benchmark circuits are shown in Table 6.1. Because several pins in a net may fall in the same grid, the number of routed nets is less than the total number of nets. For *Labyrinth*, 70% of the shortest connections are routed by pattern routing, which is the same as in [47]. We measure wirelength and total overflow in the same manner as [62] and [47]. The results are summarized in Table 6.2. *FastRoute* 2.0 can achieve 0 overflow for 6 circuits out of the total 9 circuits, and the total overflow is reduced by more than an order of magnitude compared to the other three routers. The wirelength of *FastRoute* 2.0 is also the least among all the routers. At the same time, *FastRoute* 2.0 is 73% slower than *FastRoute*, but 78× and 37× faster than *Labyrinth* and *Chi Dispersion* router, respectively. Because we cannot find a version to duplicate the results in [47], the runtime of *Chi*

Table 6.2 Comparion between FastRoute 2.0, FastRoute, Labyrinth and Chi Dispersion router

	FastRoute 2.0			FastRoute			Labyrinth Predictable router			*Chi* Dispersion router		
	Ovflow	Wirelen	Time(s)	Ovflow	Wirelen	Time(s)	Ovflow	Wirelen	Time(s)	Ovflow	Wirelen	Time(s)[1]
ibm01	31	68489	0.72	250	67128	0.21	242	76228	16.99	189	66005	8.63
ibm02	0	178868	0.93	39	179995	0.56	214	202235	26.53	64	178892	26.27
ibm03	0	150393	0.60	1	151023	0.43	117	191500	37.92	10	152392	24.71
ibm04	64	175037	1.88	567	172593	0.50	786	198181	80.95	465	173241	32.94
ibm06	0	284935	1.36	33	285882	0.91	130	339379	72.06	35	289276	53.33
ibm07	0	375185	1.60	18	376835	1.05	407	450855	168.41	309	378994	79.61
ibm08	0	411703	2.36	58	412915	1.16	352	466556	154.82	74	415285	72.94
ibm09	3	424949	1.92	28	426471	1.39	310	481841	229.59	52	427556	86.67
ibm10	0	595622	2.79	18	599433	1.98	288	680113	296.70	73	599937	139.61
Total	98	2665181	14.16	1012	2672275	8.19	2846	3086888	1083.97	1271	2681578	524.71
Norm[2]	**1**	**1**	**1**	**10.327**	**1.003**	**0.578**	**29.041**	**1.158**	**77.552**	**12.969**	**1.006**	**37.056**

1. Scaled runtime on our machine. 2. Normalized to FastRoute 2.0 results.

Dispersion router is scaled from the runtime in [47] based on the information from Standard Performance Evaluation Corporation (SPEC) [61]. In [47], it was claimed that runtime of *Chi Dispersion* router is roughly 2× faster than *Labyrinth*, which coincides with the scaled runtime we obtained. We also get a new version of *Chi Dispersion* router from the authors of [47], the total overflow on the same set of benchmark is 804, but the total runtime is about 65× slower than *FastRoute* 2.0, which is close to the runtime of *Labyrinth*. In [62], a beaver mode of *FastRoute* is also reported. The beaver mode runs more rounds of maze routing to reduce the overflow. The total overflow of the beaver mode is 512 and it is 2.2× slower than *FastRoute* default mode, which is worse than *FastRoute* 2.0 in both total overflow and runtime. This indicates that just applying more maze routing in *FastRoute* cannot achieve the high-quality results of *FastRoute* 2.0.

Second, we investigate the effect of monotonic routing technique. In order to show the effect of monotonic routing, we set up 4 different flows for phase 3.

- Only Z-shaped pattern routing
- Only Monotonic routing
- Z-shaped pattern routing + Maze routing
- Monotonic routing + Maze routing

Table 6.3 shows the comparison results between the first and second flow, as well as between the third and fourth flow. It is clear that monotonic routing can generate less congested solutions before and after the maze routing. And we also measure the runtime for one full round of monotonic routing and one full round Z-shaped pattern routing. The previous one is about 2.3× slower than the latter. But one point we want to mention that sometimes pattern routing may be preferred because it may generate less vias. When there is strict constraint on vias, pattern routing may be a good choice.

Third, we report the total number of tree edges, the percentage of tree edges been maze routed, and the percentage of tree edges whose endpoints are changed during multi-source

Table 6.3 Overflow values of different flows

	Z	Monotonic	Z + maze	Monotonic + maze
ibm01	1435	1280	40	31
ibm02	2711	2569	0	0
ibm03	260	145	0	0
ibm04	1950	1794	112	64
ibm06	1682	1444	0	0
ibm07	1020	853	0	0
ibm08	963	735	1	0
ibm09	1065	626	21	3
ibm10	1834	1532	2	0
Total	12920	10978	176	98

multi-sink maze routing. From Table 6.4, we can see that only 2.34% of edges are maze routed, which is the main reason why the algorithm is very fast. Also, the results show that a significant portion (1 out of 3.5) of the edges been maze routed have their endpoints changed during the multi-source multi-sink maze routing. This indicates the effectiveness of not constraining the endpoints of the routing path for the edges.

Table 6.4 Maze routing Statistics

	Total # of tree edges	Edges being maze routed (%)	Edges w/ endpoints changed (%)
ibm01	28116	3.24%	1.12%
ibm02	55361	4.00%	1.25%
ibm03	45582	1.79%	0.45%
ibm04	53308	4.04%	0.88%
ibm06	82283	2.43%	0.62%
ibm07	109175	1.89%	0.44%
ibm08	133222	1.13%	0.30%
ibm09	128185	1.25%	0.45%
ibm10	181432	1.25%	0.47%
Avg		**2.34%**	**0.66%**

CHAPTER 7. AN INTEGRATED PLACEMENT AND ROUTING APPROACH

7.1 Introduction

As we mentioned in previous chapters, as the technology enters nanometer regime, circuit placement has become a critical step in the VLSI design flow. First, placement largely determines the length and hence the delay of interconnect wires. Second, placement also determines the routing congestion. Because of the shrinking of device size, chip size is no longer determined exclusively by total cell area, but often by the limited routing resources. In today's advanced technologies, it is typical that more than half of the modern chip area is occupied by white space [67]. Hence, a good placement can improve routability and enable the use of a smaller area. Third, as design size keeps on increasing, millions of cells need to be placed. It is crucial to have very efficient algorithms to handle the large problem size.

In placement stage, because it is very difficult to optimize timing or routing congestion directly, wirelength is a commonly used objective during the optimization process. In order to simplify the problem, inaccurate models (e.g., half-perimeter of bounding rectangle, clique-model, star-model) are widely used to approximate the characteristics of interconnects. However, these models are far from the actual implementation of interconnects (i.e., routing). The interconnect resources required by routing stage are not adequately estimated and reserved during placement. This causes the inconsistency between the objectives optimized in placement stage and routing stage. The placed circuit may not be routable and the estimated interconnects timing may not be achievable. Therefore, the placement and routing solutions generated by this sequential approach are far from optimal.

Facing the challenges in placement and routing, in order to achieve good placement solutions for modern large-scale designs and ensure the subsequent routing procedure, we propose an *integrated placement and routing approach*.

As we mentioned in Chapter 5, in order to get accurate interconnect information during the placement process, it is desirable to incorporate global routing into it. Global routing allocates the routing demand globally over the chip area. It generates interconnect information very close to the final routing implementation and can be used for accurate estimation of interconnect topology, wirelength, delay, congestion, buffering solution, etc. If we can integrate it into placement, high-quality placement results in terms of routability and timing property can be achieved.

However, the major obstacle for this integration is complexity. Since both placement and routing are NP-hard problems, integrating them together using traditional algorithms will become intractable. Benefiting from the very efficient placement and routing algorithms developed in the previous chapters, we are able to integrate the global routing into placement process without compromising the speed too much. In this way, the routing information can be obtained during the placement and direct the generation of placement solution with good routability.

Traditional routability-driven placement approaches employ simple wiring models to estimate the interconnects during placement. However, they have no guarantee for the routability of the placement results. In contrast, we repeatedly perform global routing during placement in the new integrated approach. The final output is not only a high-quality placement, but also a global routing solution over it. Therefore, we can guarantee the good routability of the placement.

7.2 Previous Work

It is well-known that a placement with small HPWL may be unroutable due to uneven routing demand and ensuing wiring congestion. A simple example is to compact a placement to obtain another one with less wirelength. After the compaction, the routability becomes worse in spite of the decrease in HPWL. Hence, routability-driven placement algorithms are proposed to explicitly account for routing congestion in order to produce routable placements. In [68], congestion maps are built after global placement, and annealing moves are applied to minimize a congestion metric. Another technique known as WSA [69] is applied after detailed placement. WSA uses congestion maps to identify areas with high congestion and injects whitespace into these areas in a top-down fashion. After all the whitespace allocation and legalization, window based detail placement techniques are applied to reduce wirelength. Cell bloating [70] and cell spreading [69] are used to tie whitespace to specific cells. In [71], single-trunk Steiner tree models is employed to reduce congestion in FPGAs. Recently, the authors of [72] developed a placement technique called ROOSTER to optimize Steiner-tree wirelength in global and detailed placement. ROOSTER improves overall Place-and-Route results over previous works.

All previous work tried to achieve routability by including some congestion measure and use that information as a guide. Instead of routing, Steiner trees and probabilistic congestion prediction are used to get congestion map during placement. However, as pointed out in Chapter 5, it is impossible for a congestion estimator to predict the routing congestion accurately. The only possible way to predict congestion accurately is to use the same technique and parameters in both congestion estimation and global routing. However, although the integrated placement and routing is very desirable, the intractable complexity prevents the practical use of this idea. As far as we know, this is the first work to integrate the placement and routing into the same framework in general ASIC design flow.

7.3 Overview of Integrated Placement and Routing Approach

Our integrated approach is based on FastPlace, FastDP, and FastRoute proposed in previous chapters. In this section, we first discuss some issues for integration, such as routing updating strategy, different routing accuracy. Then, we give the flow of our integrated placement and routing algorithm.

7.3.1 Integration Issues

Although we have efficient placement and routing techniques mentioned in previous chapters, it is not trivial to assemble them together to get an integrated approach. There are several issues for integrated placement and routing to obtain high-quality solutions with affordable runtime.

First, during the iterative placement process, cells are moving all the time. As they change positions, we need to redo the routing. However, although we have very fast global routing algorithms, simply performing global routing after each move of cells is not applicable because each cell will change positions thousands of times during placement and the cell number is huge. Therefore, we perform incremental rip-up and reroute to keep the routing updated. In this method, when we moving a cell, only the nets connecting to this cell will be ripped up and rerouted. This incremental updating strategy may not give the best possible routing. However, it is good enough for routing congestion estimation and will not lead to unaffordable runtime.

Second, during placement process, cell positions are gradually refined from stage to stage. Hence, we adapt routing to this kind of increasing accuracy. Therefore, we apply routing techniques with different accuracy during the whole placement and routing process to balance the accuracy and runtime. In addition, the updating methods also have increasing accuracy. As we will discuss in detail in later sections, the routing information is also becoming more and more accurate as placement approaches final solution.

7.3.2 IPR Flow

We build up the new integrated framework based on the very efficient placement and routing algorithms developed in the previous chapters. The new framework follows the basic flow of *FastPlace*. It also has three stages: (1) global placement, (2) legalization and (3) detailed placement. However, different from *FastPlace*, the objective is no longer HPWL, but good routability. Hence, although we still have these three stages in the flow, many new techniques are introduced due to the new objective.

The flow of this integrated placement and routing approach is summarized in Figure 7.1.

Stage 1: Global Placement

1. Repeat
   a. Solve convex quadratic program
   b. Perform *cell-shifting* and add spreading force
2. Until the placement is roughly even
3. Repeat
   a. perform *Steiner-WL based Iterative Local Refinement*
4. Until the placement is even and no significant improvement on Steiner-WL
5. Run *FastRoute* to get an initial global routing and congestion map
6. Repeat
   a. Perform *Routability Driven Refinement* to reduce routing congestion
7. Until no significant improvement on congestion

Stage 2: Legalization

8. Move standard-cells among segments to satisfy segment capacities
9. Legalize standard-cells within segments

Stage 3: Detailed Placement

10. Run *FastRoute* to get an initial global routing and congestion map
11. Repeat
    a. Apply *Routability Driven Global Swap* to reduce routing congestion
    b. Updating routing and congestion map by rip-up and reroute
12. Until no significant improvement on congestion
13. Run *FastRoute* again to get global routing and congestion map
14. Repeat
    a. Apply *Routability Driven Local Swap* to reduce routing congestion
    b. Updating routing and congestion map by rip-up and reroute
15. Until no significant improvement on congestion

Figure 7.1 Flow of Integrated Placement and Routing Approach

Notice that the new techniques for reducing routing congestion are *Steiner-WL based Interative Local Refinement*, *Routability Driven Refinement*, *Routability Driven Global Swap* and *Routability Driven Local Swap*. In addition, routing is closely integrated in this framework to obtain global routing and congestion map. Finally, the output is a placement with good routability and the global routing over it.

In the following sections, we will introduce these new techniques in this integrated placement and routing framework in detail.

7.4 Global Placement

The first part of global placement (lines 1-2) is very similar to original *FastPlace*. It solves convex quadratic program to optimize quadratic wirelength and employes cell-shifting to even out the placement. The goal is to achieve a good initial placement.

After the initial placement obtained, original *FastPlace* apply *HPWL based Iterative Local Refinement* (*ILR*) to optimizing HPWL and generate a even placement. In the new framework, HPWL is no longer measured and optimized. Instead, the goal is to achieve an even placement with less routing congestion. Hence, the original *ILR* technique is modified to optimize the Steiner tree wirelength. The new technique is called *Steiner-WL based Iterative Local Refinement* (*StWL ILR*) (lines 3-4). The reason to use Steiner-WL as the objective is that it has much higher fidelity with routed wirelength than HPWL. In this step, since we need to evaluate Steiner-WL all the time, the Steiner-WL algorithm has to be very fast. Hence, we employ the extremely fast Steiner-WL estimator - FLUTE [37].

After *StWL ILR* achieves relatively even placement with good ordering, we fixed it as a starting placement for the consideration for congestion reduction. *FastRoute* is invoked (line 5) to generate the global routing and find congestion map over the current placement.

Based on the congestion map, a new *Routability Driven Refinement* (*RDR*) technique is designed to optimize the routability directly. For each cell, we tried to move it in 8 different directions, similar to original *ILR* technique. However, the criterion to move a cell is not the score based on bin cell utilization change and HPWL reduction. Now we measure the congestion before and after moving a cell. Then based on the bin cell utilization change and congestion reduction, we decide whether to move a cell to certain direction. Note that we are not using any congestion estimator to predict the congestion variation, but applying rip-up and reroute to update routing and congestion map. Therefore, we have high confidence for each move to reduce the routing congestion.

After the loop of *RDR* (line 7-8), an even placement with good routability is obtained. Now the cells are in good relative positions with overlaps among each other.

7.5 Legalization

For legalization, the main goal now is to put all cells in legal positions and remove overlaps. At the same time, it needs to maintain the cell at their positions in global placement stage to ensure the routability enabled by global placement.

For placement with high cell utilization, very little whitespace (about 5%) is available and legalization becomes very hard. We add the segment untilization control method to move the cells from segments over capacity to other segments. In this way, we can quickly make all the segments within capacity.

Now each segments can hold all the cells in it. The problem left is to remove overlaps among the standard cells to achieve the legal placement. As we mentioned, it is very important to maintain the cell positions in global placement so that the good routability will not be destroyed. Hence, we developed a *Minimum Movement Legalization MML* algorithm to achieve this. This algorithm is similar to the *Iterative Clustering Algorithm* [28] in *FastPlace*. It can optimally decide the cell positions for a segment with minimum total movement.

7.6 Detailed Placement

In detailed placement stage, *FastRoute* is first employed to generate the initial global routing solution and congestion map. After that, *Routability Driven Global Swap* and *Routability Driven Local Swap* are performed repeatedly to further reduce routing congestion.

7.6.1 Routability Driven Global Swap

First, similar to *Global Swap* in Chapter 3, for each cell i, we find its optimal region. Then we consider each cell j in optimal region as a candidate to swap with i. The major difference between *Routability Driven Global Swap* and *Global Swap* in Chapter 3 is that we no longer care for HPWL, but routing congestion. Hence, we need to keep measuring and optimizing routing congestion instead of HPWL.

Before any tentative swap, we measure the congestion score for current routing. For each candidate cell j, we perform the following operations. (1) Rip-up all the routing trees for all the nets connecting to i and j. (2) Swapping i with j, update the pin locations for the affected nets. (3) Reroute all the nets connecting to i and j for the new pin positions. (4) Measure the routing congestion score. (5) Recover the original routing trees and congestion map before the tentative swap. In this way, we can get the congestion score for each candidate cell j.

Consider the congestion score together with the overlap penalty (as in Chapter 3), the benefit for swapping each candidate cell j with i can be obtained. Based on this benefit, we pick the j resulting the best benefit and swap it with i. After the swap, we update the cell positions, reroute the affected nets according to the new pin locations, and update the congestion map. This procedure involves a lot of routing task and is very time consuming. However, we are able to perform it due to our extremely fast global routing technique.

Note that in this approach, every move is intended for reducing routing congestion. This is achieved by keeping the routing solution and congestion map accurate. And it is the reason why we spend so much runtime on updating routing, as well as the congestion map.

From Figure 7.1, there is a loop for *Routability Driven Global Swap*. We run several iterations and in each iteration we scan through all the cells for good swaps. After the loop, we can reduce the routing congestion significantly.

7.6.2 Routability Driven Local Swap

Routability Driven Global Swap is very effective in reducing routing congestion. However, because of a lot of rip-up and reroute operations, we cannot afford to run many iterations of it. Inspired by the *Vertical Swap* and *Local Re-ordering* techniques in Chapter 3, we develop a *Routability Driven Local Swap* technique to reduce the routing congestion by moveing cells locally.

The main idea for *Routability Driven Local Swap* is as follows. For each cell i, we just look at the cells directly adjacent to i, i.e., the four neighbors of i. Then, we consider swapping i with them one by one. We also pick the one resulting the best benefit and swap it with i, similar to *Routability Driven Global Swap*. But in this technique, we only consider the swaps without creating any overlap. Otherwise, the swap will be neglected.

We have mentioned that the most time consuming part in *Routability Driven Global Swap* is to rip-up and reroute the nets being affected. In order to speed up *Routability Driven Local Swap*, we only update the tree branches being affected by the swap instead of updating all the routing trees. Here, the basic assumption is that the routing tree topologies will not be affected by the swap. Since we only move cells very locally, the routing tree topologies will remain the same in most cases. Notice that in *Routability Driven Global Swap*, this assumption will not hold.

Another reason why this technique is much faster than *Routability Driven Global Swap* is less candidate cells are considered for each cell i. In *Routability Driven Local Swap*, only four candidate cells will be considered. However, in *Routability Driven Global Swap*, there could be hundreds even thousands of candidate cells in the optimal region.

7.7 Experimental Results

In this section, we present our experimental results. All experiments were performed on a Linux workstation with Intel Pentium 4 3.0 GHz CPU and 2GB memory.

We ran experiments on the IBMv2 suite of benchmarks. For the easy case for all the circuits, it is very easy to achieve 0 overflow by different flows. Therefore, we only show the experimental results on the hard cases in IBMv2 suite of benchmarks.

We compare our integrated placement and routing approach with ROOSTER [72], which is so far the best routability driven placement algorithm. The global routing results is obtained by running FastRoute 2.0 on placement solutions generated by ROOSTER.

Table 7.1 Comparison results

Benchmarks	# cells	# nets	IPR		*ROOSTER+FastRoute*	
			Overflow	Time(s)	Overflow	Time(s)
ibm01h	12.0k	11.5k	0	77	0	273
ibm02h	19.1k	18.4k	0	396	463	696
ibm07h	44.8k	44.4k	369	945	736	1469
ibm08h	50.7k	47.9k	4	1275	97	2150
ibm09h	51.4k	50.4k	0	1028	2	1657
ibm10h	66.8k	64.2k	0	1685	11	2498
ibm11h	68.0k	67.0k	1	1395	25	2131
ibm12h	68.7k	67.7k	1152	2104	1046	2881
Total			1526	8906	2380	13755

The results are summarized in Table 7.1. We measure the total overflow and runtime for both flows. From the results we can see that our integrated placement and routing approach is better in both solution quality and runtime. This justifies the effectiveness of our integrated approach.

BIBLIOGRAPHY

[1] T. Chan, J. Cong, T. Kong, and J. Shinnerl. Multilevel optimization for large-scale circuit placement. In *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*, pp. 171-176, 2000.

[2] C. C. Chang, J. Cong, and X. Yuan. Multi-level placement for large-scale mixed-size IC designs. In *Proc. Asia and South Pacific Design Automation Conf.*, pages 325–330, 2003.

[3] T. Chan, J. Cong, K. Sze. Multilevel generalized force-directed method for circuit placement. In *Proc. Intl. Symp. on Physical Design*, pp. 185-192, 2005.

[4] A. B. Kahng, S. Reda, and Q. Wang. Architecture and details of a high quality, large-scale analytical placer. In *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*, pages 890–897, 2005.

[5] B. Hu and M. Marek-Sadowska. Multilevel fixed-point-addition-based VLSI placement. *IEEE Trans. Computer-Aided Design*, 24(8):1188–1203, August 2005.

[6] ISPD04 IBM Standard Cell Benchmarks with Pads. *http://www.public.iastate.edu/~nataraj/ISPD04_Bench.html*.

[7] S. N. Adya, and I. L. Markov. Consistent Placement of Macro-Blocks using Floorplanning and Standard-Cell Placement. In *Proc. Intl. Symp. on Physical Design*, pp. 12-17, 2002.

[8] M. C. Yildiz and P. H. Madden. Global objectives for standard cell placement. In *Proc. 11th Great Lakes Symposium on VLSI*, pp. 68-72, 2001.

[9] A. Agnihotri et al. Fractional Cut: Improved Recursive Bisection Placement. In *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*, pp. 307, 2003.

[10] A. E. Caldwell, A. B. Kahng, and I. L. Markov. Can recursive bisection produce routable placements? In *Proc. ACM/IEEE Design Automation Conf.*, pp. 477-482, 2000.

[11] A. E. Caldwell, A. B. Kahng, and I. L. Markov. Optimal Partitioners and End-Case Placers For Standard-Cell Layout. In *IEEE Trans. on Computer-Aided Design, vol. 19*, pp. 1304-13, 2000.

[12] K. Doll, F. M. Johannes, and K. J. Antreich. Iterative Placement Improvement by Network Flow Methods. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 13(10):1189-1200, 1994.

[13] H. Eisenmann and F. M. Johannes. Generic global placement and floorplanning. In *Proc. ACM/IEEE Design Automation Conf.*, pp. 269-274, 1998.

[14] S. Goto. An Efficient Algorithm for the Two-Dimensional Placement Problem in Electrical Circuit Layout. In *IEEE Transitions on Circuits and Systems, Vol. CAS-28, No. 1*, pp. 12-18, 1981.

[15] B. Hu and M. Marek-Sadowska. FAR: Fixed-points addition and relaxation based placement. In *Proc. Intl. Symp. on Physical Design*, pp. 161-166, 2002.

[16] S.-W. Hur and J. Lillis. Relaxation and Clustering in a Local Search Framework: Application to Linear Placement. In *Proc. ACM/IEEE Design Automation Conf.*, pp.. 360-366, 1999.

[17] S.-W. Hur and J. Lillis. Mongrel: Hybrid Techniques for Standard Cell Placement. In *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*, pp. 165-170, 2000.

[18] S.-W. Hur, et al. Force directed Mongrel with physical net constraints. In *Proc. ACM/IEEE Design Automation Conf.*, pp.. 214-219, 2003.

[19] A. B. Kahng, I. L. Markov and S. Reda. On Legalization of RowBased Placements. In *Proc. Great Lakes Symp. on VLSI*, pp. 214-219, 2004.

[20] A. B. Kahng and Q. Wang. Implementation and extensibility of an analytical placer. In *Proc. Intl. Symp. on Physical Design*, pp. 18-25, 2004.

[21] A. B. Kahng, P. Tucker and A. Zelikovsky. Optimization of Linear Placements for Wirelength Minimization with Free Sites. In *in Asia and South Pacific Design Autom. Conf.*, pp. 241-244, 1999.

[22] J. Kleinhans, G. Sigl, F. Johannes, and K. Antreich. Gordian: VLSI placement by quadratic programming and slicing optimization. *IEEE Trans. Computer-Aided Design*, 10(3):356-365, 1991.

[23] Gi-Joon Nam, Charles J. Alpert, Paul Villarrubia, Bruce Winter and Mehmet Yildiz. The ISPD2005 placement contest and benchmark suite. In *Proc. Intl. Symp. on Physical Design*, pp. 216-220, 2005.

[24] C. Sechen and A. L. S. Vincentelli. Timberwolf 3.2: A new standard cell placement and global routing package. In *Proc. ACM/IEEE Design Automation Conf.*, pp. 432-439, 1986.

[25] R. Varadarajan. Convergence of placement technology in physical synthesis: Is placement really a point tool? In *Proc. Intl. Symp. on Physical Design*, pp. 7-12, 2003.

[26] N. Viswanathan, and Chris C. N. Chu. FastPlace: Efficient Analytical Placement using Cell Shifting, Iterative Local Refinement and a Hybrid Net Model. In *Proc. Intl. Symp. on Physical Design*, pp. 26-33, 2004.

[27] N. Viswanathan, and Chris C. N. Chu. FastPlace: Efficient Analytical Placement using Cell Shifting, Iterative Local Refinement and a Hybrid Net Model. *IEEE Trans. Computer-Aided Design*, 24(5):722-733, 2005.

[28] N. Viswanathan, M. Pan and Chris C. N. Chu. FastPlace 2.0: An Efficient Analytical Placer for Mixed-Mode Designs. In *in Asia and South Pacific Design Autom. Conf.*, pp. 195-200, 2006.

[29] M. Wang, X. Yang, and M. Sarrafzadeh. Dragon2000: Standard-cell placement tool for large industry circuits. In *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*, pp. 260-263, 2000.

[30] Z. Xiu, J. D. Ma, S. M. Fowler, and R. A. Rutenbar. Large-Scale Placement by Grid-Warping. In *Proc. ACM/IEEE Design Automation Conf.*, pp. 351-356, 2004.

[31] J. Cong, K. S. Leung, and D. Zhou. Performance-Driven Interconnect Design Based on Distributed RC Delay Model. *Proc. IEEE/ACM Design Automation Conf.*, 606-611, 1993.

[32] C. J. Alpert, et. al. Buffered Steiner Trees for Difficult Instances. In *Proc. Intl. Symp. on Physical Design*, 4-9, 2001.

[33] F. K. Hwang. On Steiner minimal trees with rectilinear distance. *SIAM Journal of Applied Mathematics*, 30:104-114, 1976.

[34] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, NY, 1979.

[35] J. Griffith, G. Robins, and J. S. Salowe. Closing the gap: Near-optimal Steiner trees in polynomial time. *IEEE Trans. Computer-Aided Design*, 13(11):1351-1365, November 1994.

[36] I. I. Mandoiu, V. V. Vazirani, and J. L. Ganley. A new heuristic for rectilinear Steiner trees. In *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*, 157-162, 1999.

[37] Chris Chu. FLUTE: Fast Lookup Table Based Wirelength Estimation Technique. In *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*, 696-701, 2004.

[38] Chris Chu, Yiu-Chung Wong. Fast and Accurate Rectilinear Steiner Minimal Tree Algorithm for VLSI Design. In *Proc. Intl. Symp. on Physical Design*, 28-35, 2005.

[39] S. Rao, P. Sadayappan, F. Hwang and P. Shor. The Rectilinear Steiner Arborescence Problem. Algorithmica, 277-288, 1992.

[40] W. Shi and C. Su. The Rectilinear Steiner Arborescence Problem is NP-complete. In *Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, 780-787, 2000.

[41] K. D. Boese, A. B. Kahng, G. Robins. High-Performance Routing Trees with Identified Critical Sinks. In *Proc. IEEE/ACM Design Automation Conf.*, 182-187, 1993.

[42] J. Lillis, C.-K.Cheng, T.-T. Y. Lin, and C.-Y. Ho. New Performance Driven Routing Techniques with Explicit Area/Delay Tradeoff and Simultaneous Wire Sizing. In *Proc. IEEE/ACM Design Automation Conf.*, 395-400, 1996.

[43] C. J. Alpert, T. C. Hu, J. H. Huang, and A. B. Kahng. A Direct Combination of the Prim and Dijkstra Constructions for Improved Performance-Driven Global Routing. UCLA CS Dept. TR-920051, 1992.

[44] M. Hanan. On Steiner's problem with rectilinear distance. *SIAM Journal of Applied Mathematics*, 14:255-265, 1966.

[45] A. E. Caldwell, A. B. Kahng, and I. L. Markov. VLSI CAD Bookshelf. http://www.gigascale/org/bookshelf.

[46] R. Kastner, E. Bozogzadeh, and M. Sarrafzadeh. Predictable routing. In *Proc. IEEE/ACM Intl. Conf. Computer-Aided Design*, pp. 110-113, 2000.

[47] R. T. Hadsell and P. H. Madden. Improved global routing through congestion estimation. In *Proc. ACM/IEEE Design Automation Conf.*, pp. 28-31, 2003.

[48] J. Westra and P. Groeneveld. Is probabilistic congestion estimation worthwhile? In *Proc. Intl. Workshop on System-Level Interconnect Prediction*, pp. 99-106, 2005.

[49] B. Halpin, C. Y. R. Chen and N. Sehgal. Timing driven placement using physical net constraints. In *Proc. ACM/IEEE Design Automation Conf.*, pp. 780-783, 2001.

[50] U. Brenner, A. Rohe. An effective congestion-driven placement framework. In *IEEE Trans. on Computer-Aided Design, vol. 22(4)*, pp. 387-394, 2003.

[51] C.Albrecht. Global routing by new approximation algorithms for multicommodity flow. In *IEEE Trans. Computer-Aided Design*, 20:622-631, 2001.

[52] B. M. Riess, and G. G. Ettelt. SPEED: fast and efficient timing driven placement. *Proc. Intl. Symp. on Circuits and Systems*, pp. 377-380, 1995.

[53] W. Swartz and C. Sechen. Timing Driven Placement for Large Standard Cell Circuits. In *Proc. ACM/IEEE Design Automation Conf.*, pp. 211-215, 1995.

[54] X. Yang, B. K. Choi, and M. Sarrafzadeh. Timing-driven placement using design hierarchy guided constraint generation. In *Proc. IEEE/ACM Intl. Conf. Computer-Aided Design*, pp. 177-184, 2002.

[55] G. Stem, B. M. Riess, B. Rohfleisch and F. M. Johannes. Timing driven placement in interaction with netlist transformations. *Proc. Intl. Symp. on Physical Design*, pp. 36-41, 1997.

[56] J. Lou, S. Krishnamoorthy, and H. Sheng. Estimating routing congestion using probabilistic analysis. In *Proc. Intl. Symp. on Physical Design*, pp. 112-117, 2001.

[57] J. Westra, C. Bartels, and P. Groeneveld. Probabilistic congestion prediction. In *Proc. Intl. Symp. on Physical Design*, pp. 204-209, 2004.

[58] A. B. Kahng and X. Xu. Accurate pseudo-constructive wirelength and congestion estimation. In *Proc. Intl. Workshop on System-Level Interconnect Prediction*, pp. 61-68, 2003.

[59] R. W. Hamming. Error-detecting and error-correcting codes. *Bell System Technical Journal*, 29(2):147-160, 1950.

[60] D. von Seggern. CRC Standard Curves and Surfaces. Boca Raton, FL: CRC Press, 1993.

[61] http://www.spec.org/

[62] M. Pan and C. Chu. FastRoute: A step to integrate global routing into placement. To appear. *IEEE/ACM Intl. Conf. Computer-Aided Design*, 2006.

[63] R. Carden, J. Li, and C.-K. Cheng. A global router with a theoretical bound on the optimal solution. In *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 15(2):208-216, 1996.

[64] E. Dijkstra. A note on two problems in connexion with graphs. In *Numerische Mathematik, vol. 1*, pp. 269-271, 1959.

[65] J. Cong, J. Fang and Y. Zhang. Multilevel Approach to Full-Chip Gridless Routing. In *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*, pp. 396-403, 2001.

[66] T.-Y. Ho, Y.-W. Chang, S.-J. Chen, and D.-T. Lee. A fast crosstalk and performance-driven multilevel routing system. In *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*, pp. 382-387, 2003.

[67] Tsu-Wei Ku. Using "empty space" for IC congestion relief. EE Times, June 2003.

[68] X. Yang, B. K. Choi, and M. Sarrafzadeh. Routability Driven White Space Allocation for Fixed-die Standard-cell Placement. In *Proc. Intl. Symp. on Physical Design*, pp. 42-49, 2002.

[69] C. Li, M. Xie, C. K. Koh, J. Cong and P. H. Madden. Routability-driven Placement and White Space Allocation. In *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*, pp. 394-401, 2004.

[70] N. Selvakkumaran, P. Parakh and G. Karypis. Perimeter-degree: A Priori Metric for Directly Measuring and Homogenizing Interconnection Complexity in Multilevel Placement. In *Proc. Intl. Workshop on System-Level Interconnect Prediction*, pp. 53-59, 2003.

[71] D. Jariwala and J. Lillis. On Interactions Between Routing and Detailed Placement. In *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*, pp. 387-393, 2004.

[72] J. A. Roy, J. F. Lu and I. L. Markov. Seeing the Forest and the Trees: Steiner Wirelength Optimization in Placement. In *Proc. Intl. Symp. on Physical Design*, pp. 78-85, 2006.

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this dissertation.

First of all, I would like to thank Dr. Chris C.-N. Chu for his guidance, patience and support throughout this research. He introduced me into the VLSI Physical Design area. As his student, I learned a lot, not just on how to conduct research but more importantly, how to discover, analyze and solve problems in general.

I would also like to thank my committee members for their efforts and contributions to this work: Dr. Randall Geiger, Dr. Morris Chang, Dr. Degang Chen, and Dr. Maria Axenovich.

I also thank Natarajan Viswanathan for his major contribution in FastPlace work, and Dr. Priyadarsan Patra from Intel Corporation for this contribution to the topology design work.

Finally, but very importantly, I am greatly indebted to my dear wife Mingxue, my parents and my parents-in-law, who have supported me throughout the years and done all to help me concentrate on my research work.