UNIVERSITY OF MINNESOTA

This is to certify that I have examined this copy of a doctoral thesis by

Jiang Hu

and have found that it is complete and satisfactory in all aspects,
and that any and all revisions required by the final
examining committee have been made.

Professor Sachin S. Sapatnekar

Name of Faculty Advisor

Signature of Faculty Advisor

Date

GRADUATE SCHOOL

VLSI Interconnect Performance Optimization and Planning

A THESIS

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL

OF THE UNIVERSITY OF MINNESOTA

BY

JIANG HU

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Sachin S. Sapatnekar, Advisor

JANUARY 2001

©Jiang Hu 2001

Abstract

Under the sustained progress in VLSI technology, interconnect wires become increasingly important to system performance. This thesis presents research work on several aspects of VLSI interconnect performance optimization, namely, single-net performance driven routing, routing in the presence of buffer blockages and bays, performance driven multi-net global routing and interconnect planning.

Single net routing is first targeted to improve the performance of multi-pin critical nets where both timing and wire resources are stringent. Buffer insertion and driver sizing are combined separately with non-Hanan optimization, which is a topology optimization technique exploiting Steiner nodes off the Hanan grid, to minimize cost subject to timing constraints. A higher-order AWE model is employed to assure solution quality.

In real scenarios, the routing problem must be solved in an environment that prohibits buffer insertion in certain areas (called blockages), and makes banks of buffers, called bays, available in others. The second problem determines a route for a multi-pin net to avoid(seek) buffer blockages(bays) as much as possible without large wiring detours, so that buffers can be inserted to improve interconnect performance. This problem is solved by iteratively ripping up a sub-path of an existing routing tree and reconnecting the subtrees through maze routing.

The next part of the thesis considers the problem of simultaneous routing of multiple global nets. Two algorithms are proposed: the first optimizes congestion and delay based on hierarchical bisection and a network flow algorithm, while the second optimizes these objectives and in addition, the

number of bends through probability-based gradual refinement. In both algorithms, routing flexibilities under timing constraints are exploited to reduce congestion without violating timing constraints.

The final part of the thesis develops a methodology for early wire and buffer planning. Under modern VLSI technology, optimizing interconnect only at the routing stage is not sufficient and interconnect and buffer locations should be considered in early stages of design. A four-stage heuristic is developed which includes a new wirelength-based optimal buffer site allocation algorithm. Both wires and buffers are planned such that congestion is minimized.

Acknowledgment

I must express my deep gratitude to my advisor Professor Sachin Sapatnekar who has led me towards becoming a researcher in the VLSI CAD area. He has provided me with persistent help, encouragement and guidance on almost all aspects of my research work. His affable personality has also made working with him a pleasure. I am truly indebted to him for his great impact on my career and life.

I also thank my committee members, Professor E. Shragowitz, Professor G. Sobelman and Professor K. Bazargan for their helpful comments.

I am grateful to Dr. Charles Alpert, the mentor of my internship at IBM Austin Research Lab. He has shown me how to solve realistic problems in an industrial environment, and I have learned a lot from him on the theoretical and practical sides of research as well as practical skills. I also thank Steve Quay at IBM Microelectronics for his help during my internship, and IBM Austin Research Lab for providing me with the opportunity to work as an intern.

I owe many thanks to fellow graduate students for making my stay pleasant and fruitful: Naresh Maheshari, Yanbin Jiang, Min Zhao, Kishore V. Kasamsetty, Kaushik Gala, Jatuchai Pangjun, Mahesh Ketkar, Haihua Su, Suresh Raman, Arvind Karandikar, Raza ul Mustafa, Haitian Hu, Cheng Wan and Rupesh Shelar.

I thank National Science Foundation, Semiconductor Research Cooperation for funding parts of my research, and IEEE, ACM and the University of Minnesota for providing financial support to attend conferences.

Of course, I am grateful to my parents for their encouragement and support throughout the years, ever since I was a child.

Finally, I would like to express my deep gratitude to my wife Min Zhao for her help and support, and most of all, for making my life more meaningful.

Contents

1 Introduction **1**

1.1 Background and Research Goal 1

1.2 Contributions 3

2 Preliminaries **6**

2.1 Basic Definitions 6

2.2 Delay Models 8

2.3 Soft Edges 9

2.4 Non-Hanan Optimization 12

3 Performance Driven Single Net Routing **20**

3.1 Introduction 20

3.2 Motivation for Using Fourth Order AWE 23

3.3 The Problem Environment and Problem Formulation for BINO 26

3.4 Problem Formulation and Properties for FAR-DS 30

3.4.1 Problem Formulation for FAR-DS 30

3.4.2 Properties of Solution Regions for FAR-DS 31

3.5 Algorithms . . . 34
3.5.1 Algorithm for BINO . . . 34
3.5.2 Algorithm for FAR-DS . . . 36
3.6 Complexity Analysis . . . 42
3.7 Experimental Results . . . 43
3.8 Conclusion . . . 47

4 Routing for Buffer Blockages and Bays 49

4.1 Introduction . . . 49
4.2 Problem Formulation . . . 54
4.3 The Grid Graph Construction . . . 56
4.4 Algorithm Description . . . 62
4.4.1 Overview . . . 62
4.4.2 Maze Routing . . . 63
4.4.3 Complexity Analysis . . . 65
4.5 Improving Efficiency . . . 66
4.5.1 Sparsified Grid Graph . . . 66
4.5.2 Branch and Bound Maze Routing . . . 67
4.6 Experiments . . . 69
4.6.1 Additional Routing Cost . . . 69
4.6.2 Delay Comparisons with Buffer Insertion . . . 71
4.6.3 Fixing Slew Problems . . . 73
4.7 Conclusion . . . 75

5 **Performance Driven Multi-net Global Routing** **77**

5.1 Introduction 77

5.2 Congestion Metric and Problem Formulation 80

5.3 Routing Flexibilities under Timing Constraints 81

5.3.1 Z-edges 82

5.3.2 Slideable Steiner Node (SSN) 82

5.3.3 Edge Elongation 83

5.4 Hierarchical Algorithm 84

5.4.1 Algorithm Overview 84

5.4.2 Basic Network Formulation 87

5.4.3 Construction of Arcs for Multi-crossing Trees 90

5.4.4 Utilization of Slideable Steiner Nodes (SSN) 92

5.4.5 Network Pruning 94

5.4.6 Experimental Results 95

5.5 Gradual Refinement Algorithm 97

5.5.1 Approximated Congestion Estimation 97

5.5.2 Algorithm Motivation 100

5.5.3 Algorithm Detail 102

5.5.4 Experimental Results 107

5.6 Conclusion 110

6 Integrated Buffer and Wire Planning 111

6.1 Introduction . . . 111

6.1.1 Buffer Block Planning Methodology . . . 112

6.1.2 Buffer Site Methodology . . . 114

6.1.3 Technical Contribution . . . 115

6.2 Problem Formulation . . . 116

6.3 Buffer and Wire Planning Heuristic . . . 120

6.3.1 Stage 1: Initial Steiner Tree Construction . . . 121

6.3.2 Stage 2: Wire Congestion Reduction . . . 121

6.3.3 Stage 3: Buffer Allocation . . . 122

6.3.4 Stage 4: Final Post Processing . . . 127

6.4 Experimental Results . . . 128

6.4.1 General Performance . . . 129

6.4.2 Variations . . . 131

6.4.3 Comparisons with Buffer Block Planning . . . 134

6.5 Conclusion . . . 137

7 Conclusions 138

List of Figures

2.1 An example of rectilinear Steiner tree for a two sink net. . . . 7

2.2 Model of a wire segment. 8

2.3 Cascaded drivers and driver model. 9

2.4 Routing with soft edges. 10

2.5 An example of Hanan grid and a minimum RST over the Hanan grid. 12

2.6 A general situation where node v_k is to be connected to an edge (v_i, v_j). 13

2.7 Delay violation function vs. Manhattan distance z of connection point. 15

2.8 The non-Hanan optimization algorithm. 16

3.1 A routing tree on which Elmore delay gives large errors. 23

3.2 An example where using the Elmore delay and a higher order AWE delay may result in a different connection choice. 25

3.3 Buffer spaces, the territory box and their applications in buffer insertion. 27

3.4 An example that buffer insertion can reduce wire cost further in non-Hanan optimization. 29

3.5 BINO, iterative buffer insertion algorithm. 35

3.6 Iterative buffer insertion vs. iterative buffer deletion. 36

3.7 BINO, iterative buffer deletion algorithm. 37

3.8 Solution search scheme for FAR-DS. 38

3.9 FAR-DS, reconnection and driver sizing in valley-guided search. 40

3.10 FAR-DS, reconnection and driver sizing in iterative search. . . 42

4.1 Example of how an alternative Steiner tree can enable buffer insertion. 51

4.2 An example showing the benefit of not avoiding all blockage. . 52

4.3 Buffer bay example. 52

4.4 An example Steiner tree showing the different node types. . . 55

4.5 An example of the grid graph induced from five x and four y coordinates. 57

4.6 Examples of using to compute usable tracks. 58

4.7 The Grid_graph procedure. 59

4.8 The grid graph for an example 3-pin net and a single rectangular blockage. 60

4.9 Configurations for edges in P in the proof of Theorem 4.1. . . 61

4.10 The Steiner tree construction algorithm. 63

4.11 Algorithm for maze routing connecting two subtrees. 64

4.12 An example of grid graph sparsification. 67

4.13 An example of branch-and-bound in maze routing. 69

5.1 An example of a slideable Steiner node (SSN). 82

5.2 An example of bisection. 85

5.3 An assignment result from network flow solution. 86

5.4 Network formulation of the example in Figure 5.2 without considering SSN. The number on each arc is its capacity. 88

5.5 Relative positions of a boundary and a soft edge. 88

5.6 Network formulation considering SSN. 93

5.7 Examples of primitive demand. 97

5.8 Enumerate routes with number of bends less than 3 to obtain probabilistic demand. 99

5.9 When an SSN slides along its locus, the bounding boxes of its incident edges change as well as the primitive demands. 102

5.10 Examples for setting post node for a backbone wire. 103

5.11 The gradual refinement global routing algorithm. 105

6.1 (a) A set of 68 buffer site locations can be tiled and (b) abstracted to a total number of buffer sites lying within each tile. 117

6.2 Driver with seven sinks, whereby the maximum distance allowed between gates is three. With this interpretation of the distance rule, the driving gate must drive 11 units of wirelength. 118

6.3 Example of spanning tree edge overlap removal. 120

6.4 Example of how buffer costs are computed. For a value of $L^i = 3$, the optimal solution is shown, having total cost 1.5. . . 123

6.5 Single-sink buffer sites allocation algorithm 124

6.6 Execution of the single source algorithm on the example in Figure 5. The optimal solution has cost 1.5 and the arrows show how this cost is obtained. 125

6.7 Multi-sink buffer sites allocation algorithm 126

6.8 For a node with two children, a buffer may be used to either (a) drive both branches, (b) decouple the left branch, or (c) decouple the right branch. 127

List of Tables

1.1 Overall technology roadmap from NTRS'97. 2

2.1 Comparisons between using and without using Hanan nodes under fourth order AWE model on .18μm IC technology. . . . 17

2.2 Comparisons between using and without using Hanan nodes under fourth order AWE model on MCM technology. 18

3.1 A comparison of the Elmore and the 4th order AWE delays with SPICE. 24

3.2 Experimental results on .18μm IC, $h = 1, \rho = 2.5$ for SERT, MVERT and BINO. 45

3.3 Experimental results on MCM, $h = 1, \rho = 2.5$ for SERT, MVERT and BINO. 46

3.4 Number of sinks and CPU times. 47

3.5 Comparison of the number of overlaps between buffer spaces and routing edges with and without using soft edges 48

4.1 Summary of additional routing costs of SMT versus BBB for the hand crafted test case. 70

4.2 Summary of additional routing costs of SMT versus BBB for the macro block test case. 71

4.3 Experimental results on average slack improvements for the hand crafted test case. 73

4.4 Experimental results on slack improvement for the macro block. 74

4.5 Slew results for SMT and BBB on the microprocessor test case. 75

5.1 Benchmark circuits. 95

5.2 Experimental results on timing-constrained global routing. . . 96

5.3 Description of Test Circuits. 107

5.4 Grid size and the number of backbone wires for each circuit. . 108

5.5 Experimental results, *vio* is the number of nets with timing violations and *ben* is the maximum number of bends on a backbone wire. 108

6.1 Test circuit statistics and parameters for the first set of experiments. 129

6.2 Stage by stage experimental results for the 6 CBL circuits. The final results are shown for the last four random circuits. . 132

6.3 Summary of results when the number of available buffer sites varies. 133

6.4 Experimental results with varying grid sizes for three CBL benchmarks. 134

6.5 Comparisons of our algorithm to BBP/FR. 136

Chapter 1

Introduction

1.1 Background and Research Goal

In recent years, Very Large Scale Integrated (VLSI) circuit technology has undergone dramatic progress as characterized by the exponential scaling of feature sizes, i.e., the minimum dimension of a transistor. Such scaling follows Moore's Law [1] at the rate of 0.7× reduction every three years and is predicted to lead to over half billion transistors integrated on a single chip with clock frequency of 2-3 GHz in 2009, according to 1997 National Technology Roadmap for Semiconductors (NTRS'97) [2], summarized in Table 1.1.

As feature sizes keep shrinking, the transistor switching speeds become proportionally faster. On the other hand, interconnect wires become thinner and longer, and consequently interconnect delay grows greater with the increasing wire resistance. Both trends lead interconnect delay to dominate logic delay and become a significant bottleneck in system performance [3], so that the majority of the clock period may be spent on the interconnect

Table 1.1: Overall technology roadmap from NTRS'97.

Technology (nm)	250	180	150	130	100	70
Year	1997	1999	2001	2003	2006	2009
#transistors (million)	11	21	40	76	200	520
Across chip clock (MHz)	750	1200	1400	1600	2000	2500
Area (mm^2)	300	340	385	430	520	620
Wiring levels	6	6-7	7	7	7-8	8-9

wires through which the signals between various parts of the chip are communicated. This fact makes interconnect performance optimization to be a crucial task for delivering desired system performance. This thesis addresses several issues on interconnect performance optimization including single net routing, buffer insertion, multi-net global routing and interconnect planning.

A routing net is composed of a source pin from which the signal starts, and a set of sink pins where the signal is to be delivered. The single net routing problem is to construct a rectilinear tree representing the wires to span all the pins for a given net. The objective in the tree construction is usually to minimize the total wirelength and to ensure that the signal propagation delay from source to each sink satisfies the required constraint. The quality of a routing tree according to this objective can be improved by exploiting a better routing tree topology or properly sized drivers. As another interconnect optimization technique, buffer insertion is able to reduce the signal delay along long wires and shield out non-critical load to obtain smaller delays for critical sinks.

Most traditional works restrict the topology space to only Hanan grid [4], which is composed by the rectilinear tracks that intersect with at least one of the given pins. The routing topology space off the Hanan grid is explored in

this thesis, in conjunction with driver sizing and buffer insertion for improving the routing quality for timing critical nets. However, buffer insertion is not always feasible at a desired location due to the occupation by other cells, and its effectiveness is consequently hindered. Another part of this work focuses on routing to avoid buffer blockages in order to enable better buffer insertion solution.

When multiple nets are considered in routing, the problem is more complicated, as all nets compete for limited routing resources, and wiring congestion needs to be minimized. Minimizing wiring congestion is well known to be a difficult problem even without considering timing issues. Optimizing both congestion and timing simultaneously is a more complex issue and constitutes another effort in this thesis work.

The growing importance of interconnect performance not only affects the VLSI system performance but also challenges traditional VLSI design methodology. Traditionally, interconnect is considered only in the late stages of the entire design flow when routing information is available. Consequently, the design in the early stages that does not consider interconnect effects operates in a blindfolded manner. To overcome this, it is important that interconnect should be planned in the early stages of a design to provide credibility to early performance estimates. Based on this requirement, one part of this work is targeted to planning both buffer and wire resource allocations.

1.2 Contributions

The major contributions of this thesis are:

- To obtain greater performance improvement for interconnect, available routing flexibilities shoule be exploited and premature commitment of routing paths should be avoided. According to this requirement, we propose the concept of a *soft edge* that captures the routing flexibilities under timing constraints at the global routing stage. It can help to improve the quality of single net routing and increase the utilization of limited buffer spaces. In addition, it plays an important role in timing-constrained simultaneous global routing of multiple nets.

- For single net routing where both delay constraint and wire resources are stringent, the utilization of nodes off Hanan grid [4] in the topology space is integrated with driver sizing and buffer insertion separately to obtain significant wirelength reduction subject to timing constraints. The curvature properties for the objective function are found and exploited to obtain an efficient solution search scheme. This part of work is introduced in Chapter 3.

- In Chapter 4, a fast heuristic is proposed to make wires to avoid buffer blockages with small detour through a maze routing on a customized grid graph in combination with the branch-and-bound technique. This heuristic can handle the situation with pre-allocated buffer bays as well. Experiments on industrial designs show that it remarkably improves the quality of buffer insertion in presence of many buffer blockages.

- In order to optimize both congestion and timing which are often competing objectives in global routing for multiple nets, two novel algorithms are developed and described in Chapter 5. In both algorithms, routing flexibilities under timing constraints are obtained through deferred decision making and exploited to reduce congestion subject to

timing constraints. In the first algorithm, an elaborated network flow model is constructed so that the delay slack consumption is adaptive to congestion distribution. The second algorithm is able to optimize the number of bends on wires in addition.

- To address the importance of considering interconnect effect in early stages of design, we develop an integrated buffer and wire planning method in which a four-stage heuristic is designed to minimize both wire and buffer congestion simultaneously for multi-pin nets. A wirelength-based dynamic programming algorithm is obtained to allocate buffer sites optimally at a fast speed. This interconnect planning work is presented in Chapter 6.

Chapter 2

Preliminaries

2.1 Basic Definitions

A given set of nets is represented as $\mathcal{N} = \{N^1, N^2, ...\}$, with each net N^i being defined by a source node v_0^i and a set of sink nodes $V_{sink}^i = \{v_1^i, v_2^i, ...v_p^i\}$. A routing problem for a net N (we omit the net index for simplification without loss of generality) is to find a set of Steiner nodes $V_{Steiner} = \{v_{p+1}, v_{p+2}, ...v_{p+q}\}$ and a set of edges $E = \{e_1, e_2, ...e_{p+q}\}$ to construct a rectilinear Steiner tree (RST) $T(V, E)$, where $V = v_0 \cup V_{sink} \cup V_{Steiner}$, such that E spans all of the nodes in V. The traditional definition of $V_{Steiner}$ includes two types of nodes: (i) *internal Steiner nodes* of degree three or four, denoted by the set $V_{internal}$, and (ii) *bend nodes* of degree two that denote a path switch between a horizontal and a vertical direction, denoted by the set V_{bend}. For example, a net with two sinks is given in Figure 2.1(a). An internal Steiner node and a bend node are introduced together with four edges in Figure 2.1(b) to form a rectilinear Steiner tree as a routing solution. A bend node is introduced to make the tree conform to the requirement of

rectilinear space, and an internal Steiner node is usually employed to reduce wirelength. The location for a node v_j is specified by its coordinates (x_j, y_j), and an edge in E is uniquely identified by the node pair (v_j, v_k), and is denoted as e_{jk} or e_k interchangeably. The edge length l_{jk} is given by the Manhattan distance between the two nodes, which is $|x_j - x_k| + |y_j - y_k|$. In order to make our presentation clearer, we define the following terms:

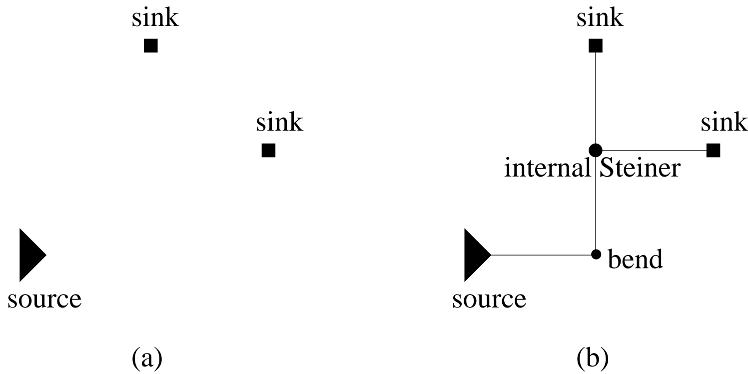


Figure 2.1: An example of rectilinear Steiner tree for a two sink net.

Definition 2.1 (backbone node) *In a routing tree, a backbone node is the source node, or a sink node, or an internal Steiner node.*

Definition 2.2 (backbone wire) *In a routing tree* $T(V, E)$*, a backbone wire is a set of consecutively adjoined edges* $\{(v, u_1), (u_1, u_2), (u_2, u_3), ...(u_m, w)\}$*, where* $v, w \in V$ *are backbone nodes and* $\{u_1, u_2, ...u_m\} \in V$ *are not backbone nodes.*

If the delay at an arbitrary sink v_a is $t(v_a)$ and its required arrival time is $RAT(v_a)$, then the delay slack $s(v_a) = RAT(v_a) - t(v_a)$. We also use the term *delay violation* $u(v_a) = -s(v_a)$ in this thesis. The timing slack $\mathcal{S}(T^i)$ for a routing tree T^i on the net N^i is the minimum delay slack among all the sinks in this net.

2.2 Delay Models

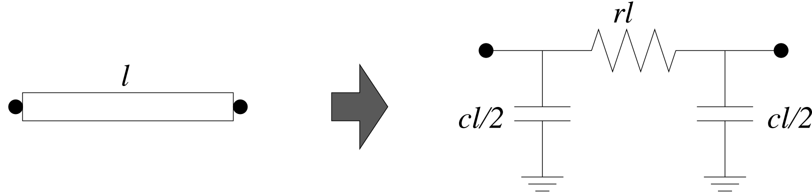


Figure 2.2: Model of a wire segment.

We use the π RC model for a wire segment of length l, as shown in Figure 2.2, where r and c are resistance and capacitance per unit length, respectively. One popular delay model for interconnect is the Elmore model [5]. We express driver resistance as R_d and let C_i denote the total downstream capacitance seen from node v_i. The Elmore delay from driver to a sink v_k is given as:

$$t_k = R_d C_0 + \sum_{e_{ij} \in path(v_0, v_k)} r l_{ij} \left(\frac{c l_{ij}}{2} + C_j \right) \tag{2.1}$$

Note that we assume that v_j is the downstream end of e_{ij}. The Elmore delay model has been widely used in many research works due to its simplicity and high fidelity [6]. Its simplicity not only removes the need for large amount of computation, but also provides a platform on which many theoretical properties can be derived and exploited.

As interconnect wires become increasingly thinner and longer, the interconnect resistance may overshadow the driver resistance. Consequently, the downstream capacitance is shielded to the driver resistance by the interconnect resistance. This effect is called resistive shielding [7]. The Elmore delay does not correctly take the resistive shielding effect into account and tends to overestimate the delay. This error can be remarkably large, especially for the stub situation (i.e., when a sink that is close to the source co-exists with a much longer wire), where the Elmore delay can be several times larger

than the actual delay. Hence, we use Elmore model for only the derivation of qualitative properties and global routing where the number of nets to be routed could be very large.

For the routing of critical nets whose timing constraint is stringent, we employ a fourth order AWE model [8]. The fourth order AWE model takes higher order moment information into consideration and can provide a much better accuracy, though the computation time becomes longer. The reason for choosing the fourth order will be explained in more details in Section 3.2.

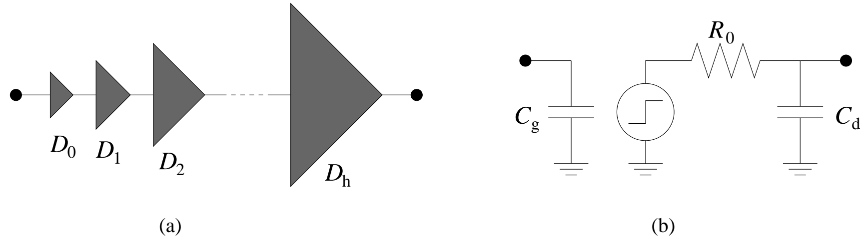


Figure 2.3: Cascaded drivers and driver model.

For driver sizing problem, we consider the situation where the signal net is driven by a series of cascaded drivers $D_0, D_1, D_2....D_h$ as in Figure 2.3(a). The driver D_0 is minimum sized and will not be changed in driver sizing. The driver and buffer model that we will use is shown in Figure 2.3(b). We denote the gate and drain capacitance of D_0 as C_g and C_d, respectively. The interconnect delay among these drivers is typically small and is neglected. The driver resistance and capacitance are assumed to change linearly with respect to the size of driver.

2.3 Soft Edges

In the process of routing for one net, multiple options are sometimes available and it is not obvious which of these is the best. Consider the example in

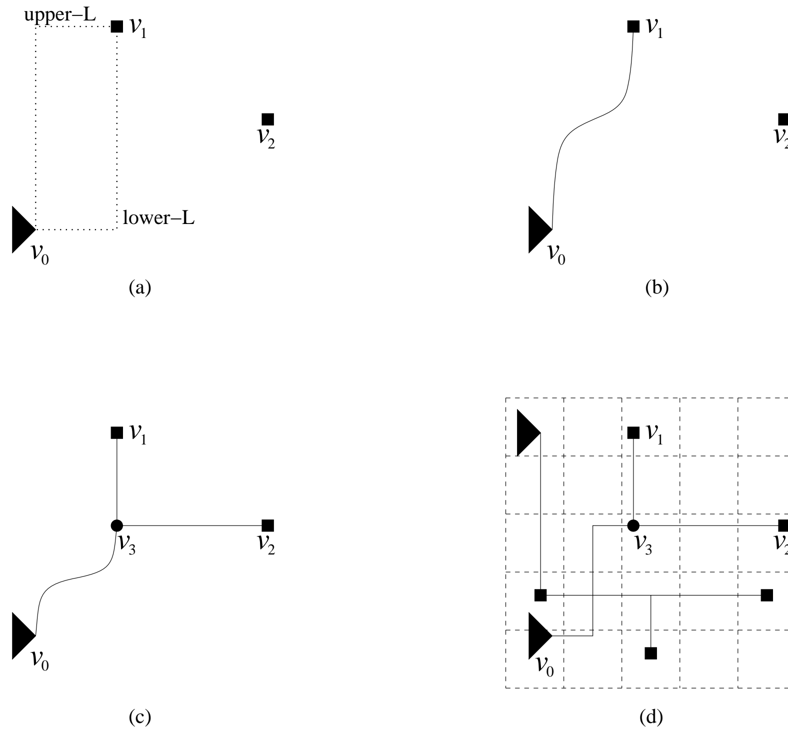


Figure 2.4: Routing with soft edges.

Figure 2.4, where a source v_0 and two sinks v_1 and v_2 are given, and a minimum Steiner tree is to be constructed on this node set by adding one node to the tree at a time. Since a routing tree is built in rectilinear space, each edge must be either horizontal or vertical. If we begin by connecting v_1 to v_0, there are two L-shaped connection options, shown by the dotted lines in Figure 2.4(a); one bend is required for each connection. The delay and wirelength from v_0 to v_1 are the same in these two options and it is hard to see which is better at this stage. Instead of fixing the edge orientation immediately as in usual approaches, we defer this decision-making to a stage when the effects of these options can be discriminated. Here, we formalize this by introducing another type of edge, a *soft edge*, whose route is not specified until there is an obvious better choice.

Definition 2.3 (soft edge) *A soft edge is an edge connecting two nodes* $v_i, v_j \in V$*, such that:*

1. $x_i \neq x_j$ *and* $y_i \neq y_j$*,*
2. *its edge length* $l_{ij} = |x_i - x_j| + |y_i - y_j|$*,*
3. *the precise edge route between* v_i *and* v_j *is not determined.*

We will refer to the traditional edges in a rectilinear tree with fixed orientations as *solid edges*. The soft edge connection between v_0 and v_1 is shown in Figure 2.4(b). In order to minimize wirelength, the sink v_2 is connected to the routing tree at the closest connection (CC) point, defined below, between v_2 and edge e_{01}.

Definition 2.4 (CC point) *The closest connection* (CC) *point between a node* v_k *and an edge* e_{ij} *is defined by its coordinates* x_{CC} *and* y_{CC} *such that:*

$$x_{CC} = median(x_i, x_j, x_k) \text{ and } y_{CC} = median(y_i, y_j, y_k).$$

Note that in Definition 2.4, the edge e_{ij} can be either a soft edge or a solid edge.

If the CC point does not coincide with either of v_i, v_j and v_k, a Steiner node is introduced at the CC point. In the example of Figure 2.4, Steiner node v_3 is introduced. After this connection has been made, edge e_{31} and e_{32} are solid edges.

At this stage, it can be seen that lower-L is a better choice for connecting v_0 and v_1 than upper-L, since it provides a shorter wirelength which is a result reached through deferred decision making. The advantage of using soft edges is that they provide a set of flexible connection choices for subsequent routing

steps and avoid premature suboptimal decisions. In fact, we do not need to choose the lower-L when connecting v_0 and v_1, and we may keep the edge e_{03} soft when v_2 is joined as indicated in Figure 2.4(c). If the extra delay on vias can be neglected, a soft edge can take many multi-bend monotone routes besides L-shaped routes. For a single net, it is again hard to see which of these routes is better for connecting v_0 and v_3. By keeping edge e_{03} soft, we can maintain these flexibilities until we consider congestion in global routing with other nets. In Figure 2.4(d), in the presence of another net, a Z-shaped route for e_{03} is chosen to reduce congestion. Using deferred decision making, routing topology flexibility can be therefore traded into congestion avoidance without hurting the delays. We will show later that the use of soft edges also has advantages that aid utilizing buffer spaces.

2.4 Non-Hanan Optimization

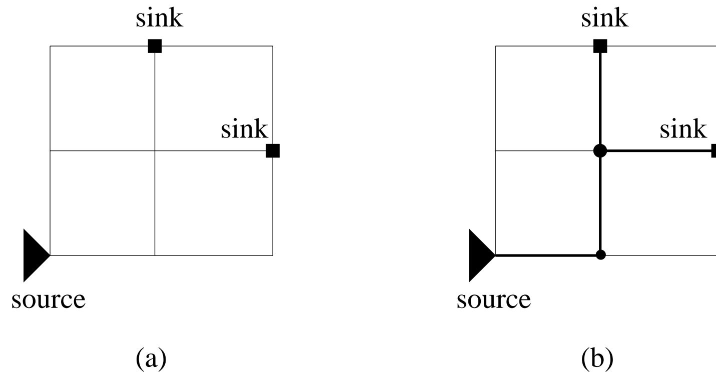


Figure 2.5: An example of Hanan grid and a minimum RST over the Hanan grid.

Drawing horizontal and vertical lines through all the pins in a given net results in the *Hanan grid* [4], illustrated in Figure 2.5(a). The researches on RST has long been restricted to the Hanan grid, because it is proved that

there is always an RST with minimum wirelength embedded in the Hanan grid as shown in 2.5(b) [4]. If the objective is to minimize a weighted sum of sink delays, there is always an optimal solution over the Hanan grid too [9].

Recently, the work of [10] showed that using Steiner nodes off the Hanan grid can yield greater reductions in the wirelength for the objective of minimizing wirelength subject to timing constraints for each sink. Based on this observation, this work proposed a non-Hanan optimization method to improve the performance of a routing tree.

As defined in [6], a maximal segment is a set of contiguous edges either all vertical or all horizontal. The work of [6] shows that the Elmore delay at each sink is a concave function with respect to the location of a Steiner node when the Steiner is moved along a maximal segment. The concavity property is exploited in non-Hanan optimization [10]. Although by definition, the orientation for a soft edge is not fixed, the concavity property continues to hold for a soft edge, and we can extend the philosophy of non-Hanan optimization to general edges including both solid and soft edges.

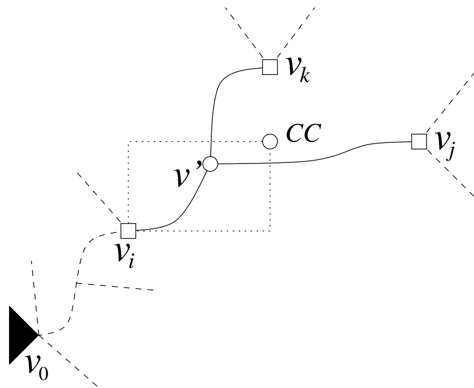


Figure 2.6: A general situation where node v_k is to be connected to an edge (v_i, v_j).

For a general form of a routing tree, shown in Figure 2.6, let us consider

the process of obtaining an optimal connection between node v_k and edge (v_i, v_j) such that wirelength is minimized subject to timing constraints. Note that v_i, v_j and v_k can either be a sink, or an internal Steiner node. The dashed lines are other nodes and edges of this routing tree, and CC represents the closest connection point between v_k and (v_i, v_j). It can be easily seen that any connection that is downstream of CC cannot give an optimal solution [6]. More specifically, we wish to search for an optimal connection point within the bounding box defined by v_i and CC. Suppose we connect v_k to (v_i, v_j) at point $v'(x', y')$. Let z be the Manhattan distance from v' to v_i, i.e., $z = |x' - x_i| + |y' - y_i|$. For convenience, we overload CC as its Manhattan distance to v_i.

Similar to the work of [6], a delay violation function model with respect to connection location for soft edges under the Elmore delay is derived as follows.

If a node is not downstream of node v_i, its Elmore delay from source v_0 is as follows:

$$f_1 = R_d(C_t - cz) + \lambda_0 + \lambda_1(l_{ik} - z), \tag{2.2}$$

where λ_0 and λ_1 are constants. Recall that C_t is the total load capacitance seen from the last stage of driver if v_k is connected to v_i.

The Elmore delay from v_i to v' is given by:

$$f' = rcz(\frac{z}{2} + l_{ij} - z + l_{ik} - z) + rz(C_j + C_k). \tag{2.3}$$

From v' to any node in T_j, which is the subtree rooted at v_j, the delay can be obtained as:

$$f_2 = r(l_{ij} - z)(\frac{c(l_{ij} - z)}{2} + C_j) + \lambda_2. \tag{2.4}$$

Similarly, the delay from v' to any node in T_k is:

$$f_3 = r(l_{ik} - z)(\frac{c(l_{ik} - z)}{2} + C_k) + \lambda_3. \tag{2.5}$$

Both λ_2 and λ_3 are constants. If a sink is in T_j, its Elmore delay is formed by the sum of f_1, f' and f_2. When a sink is in T_k, its Elmore delay is the sum of f_1, f' and f_3. If a sink is not downstream of n_i, its Elmore delay is simply f_1. In all these cases, the delay is either a linear or a quadratic function of z with non-positive coefficient for the second order term. Therefore, we can obtain the conclusion that delay or delay violation function for any sink is a concave function with respect to z, which is concluded as follows.

Theorem 2.1 *Under the Elmore delay model, the delay violation at any sink in the routing tree $T(V, E)$ is a concave function with respect to z, which is the Manhattan distance from the upstream end v_i of edge $(v_i, v_j) \in E$ to the connection point between a node $v_k \in V$ and (v_i, v_j).*

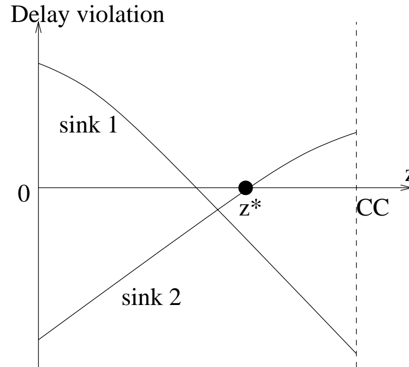


Figure 2.7: Delay violation function vs. Manhattan distance z of connection point.

Corollary 2.1 *Under the Elmore delay model, for any interval $[z_l, z_r] \subseteq [0, CC]$, if the delay violation at a sink is positive when the connection point*

is either at z_l or z_r, then the delay violation at this sink is positive when the connection point is at any location in this interval.

In Figure 2.7, the delay violation functions of a two-sink net are depicted. If the objective is to minimize wire cost subject to timing constraints, the optimal connection (Steiner) point here is a point with a non-positive delay violation, lying as close to CC as possible; for this particular example, this corresponds to z^*. As in this example, the optimal connection point is, in general, likely to be a non-Hanan point.

Algorithm: Non-Hanan_Optimization(T)
Input: Routing tree $T(V, E)$
Output: Optimized routing tree T'
1. $T' = T$
2. Sort all the nodes in descending order of distance to source
3. For each $v_k \in V, k \neq 0$
4.   Disjoin v_k and its subtree T_k from T
5.   For each edge $(v_i, v_j) \in T \backslash T_k$
6.   Reconnect v_k to (v_i, v_j) at optimal location
7.   If $\exists$ improvement compared to T'
8. $T' = T$

Figure 2.8: The non-Hanan optimization algorithm.

The work of [10] showed this advantage of using non-Hanan points and proposed the MVERT (Maximum delay Violation Elmore Routing Tree) algorithm to perform the non-Hanan optimization globally for an interconnect routing tree. Based on properties similar to Corollary 2.1, MVERT finds the

optimal connection point through a quasi-binary-search and obtains significant wire cost reductions. In fact, non-Hanan optimization can also help the net to meet timing constraints besides affecting wire cost reductions. In the example of Figure 2.7, it is probable that only non-Hanan points can satisfy the timing constraints for both sinks.

Table 2.1: Comparisons between using and without using Hanan nodes under fourth order AWE model on .18μm IC technology.

	SART		Hanan		NonHanan	
n	u_{max}	W	u_{max}	W	u_{max}	W
5	38.4	226	17.4	258	-9.4	226
5	10.1	204	-1.0	209	-24.9	188
5	42.9	288	-29.2	290	-6.2	279
10	17.3	388	7.3	381	-3.5	354
10	36.9	417	12.4	383	-3.1	396
10	52.4	502	7.5	526	-7.7	491
15	27.1	570	-5.2	539	-4.7	488
15	81.5	583	-3.1	508	-7.1	456
20	97.7	604	-5.0	612	-25.4	555
20	7.1	671	-2.8	666	-1.4	616
Ave	41.1	445	-0.2	437	-9.3	405

The algorithm of MVERT starts with SERT [6], then, all of the sinks are sorted in the descending order of Manhattan distance from source. Then each node v_k and its downstream subtree T_k are disconnected and reconnected back to the routing tree. The routing tree without T_k is represented by $T\backslash T_k$. In the search for the best reconnection point, v_k and T_k are connected to each edge in $T\backslash T_k$ tentatively at the local optimal point. The connection point is selected to be the choice that gives the largest improvement according to the objective. For two routing trees T_1 and T_2 on the same signal net, if T_1 cannot meet timing constraints and T_2 can provide a smaller maximum delay

violation among all sinks, this implies an improvement from T_1 to T_2 in spite of any cost increase. If T_1 can satisfy all the timing constraints, T_2 provides improvement only when it reduces cost and satisfies the timing constraints. For reference, the outline of the non-Hanan optimization algorithm is shown in Figure 2.8 [10].

Table 2.2: Comparisons between using and without using Hanan nodes under fourth order AWE model on MCM technology.

	SART		Hanan		NonHanan	
n	u_{max}	W	u_{max}	W	u_{max}	W
5	20.0	585	-17.9	543	-9.8	502
5	68.9	428	-2.3	532	-1.6	490
5	13.7	499	6.7	480	-19.5	472
10	93.0	803	14.3	760	-8.3	784
10	25.1	819	21.2	782	-1.6	662
10	18.2	845	-2.1	924	-2.3	806
15	42.4	1258	-4.0	1221	-14.2	1192
15	48.3	1110	-54.5	1119	-35.8	1004
20	64.8	1518	11.2	1469	-11.2	1410
20	268.1	1473	-0.5	1445	-12.0	1286
Ave	66.2	934	-2.8	927	-11.6	861

In fact, the conclusion from [11] on non-Hanan optimization is also valid under a higher order AWE model according to the experimental results shown in Table 2.1 and 2.2. The leftmost column in each table lists the number of sinks in each net. The technology parameters are same as those in [12]. The experiment starts by constructing routing trees for each net through SART, which is same as SERT [6] except that the Elmore delay model is replaced by a fourth order AWE model. The experimental results from this step are provided in column 2 and 3 in Table 2.1 and 2.2. The total wirelength is denoted as W in unit of $100\mu m$ and the maximum delay violation for each

net is represented as u_{max} in ps. Then, the optimization scheme in Figure 2.8 are performed on the SART trees also under a fourth order AWE model. We restrict the connection point in line 6 of Figure 2.8 to be only Hanan point in one variant of this optimization whose results are shown in column 4 and 5 in both tables. The results from original non-Hanan optimization are in the rightmost two columns. We can see that routing solution using only Hanan points sometimes results in positive delay violations, and these delay violations may be eliminated through using non-Hanan points. Moreover, using non-Hanan points can yield more wirelength reductions.

Chapter 3

Performance Driven Single Net Routing

3.1 Introduction

As the VLSI technology develops into the deep sub-micron era, the interconnect resistance is no longer negligible and its performance plays a critical role to the whole circuit. As the result, many efforts [6, 10, 11, 13–27] have been carried out in recent years to improve the interconnect performance. According to the classification in [13], these works have evolved along three major aspects: the *delay model*, the *objective formulation* and the *solution space*. The progress on each of these aspects will be briefly reviewed as follows.

When the interconnect resistance was not significant, it could be simply modeled as a lumped capacitance that is proportional to the wire length. Therefore, in early research, the interconnect performance criterion was purely geometric and focused on wire length based objectives such as reducing the routing radius and the total wire length. As wires have become longer and

thinner, this geometric evaluation no longer suffices to reduce interconnect delay as resistive effects become significant. A more elaborate *delay model* is necessary to augment wire length considerations in performance evaluation. The Elmore delay [5] model has been widely used due to its simplicity and high fidelity [6]. Its simplicity not only removes the need for large amount of computation, but also provides a platform on which many theoretical properties can be derived and exploited. One major Elmore delay based routing method, SERT [6], grows the routing tree in a greedy fashion to minimize the source-sink delay. Another Elmore delay application, the P-Tree algorithm [14], first searches for a good permutation of the sinks and then limits the solution space to the topologies induced by this permutation. In later work, the drawbacks of Elmore model have been addressed and second [16] and third order [17] models have been applied. Most recently, the work in [13] suggested a table lookup method to remedy the deficiencies of the Elmore delay model.

With regard to the *objective formulation*, the total wirelength (area) and delay are usually the major targets. Minimizing total wirelength can reduce fabrication cost, power consumption and improve the routability. All of these advantages lead to the use of wirelength minimization as a common baseline for objective formulations. For delay reduction, there are many forms in which the objective may be stated, including minimizing either a weighted sum of sink delays, or the maximum delay, or the critical sink delay. As a more appropriate formulation, the research in [10, 16, 20] focuses on satisfying the timing specification in an effort to trade off the unnecessary delay reduction into area minimization.

The *solution space* of nodes in the routing tree has long been restricted to the Hanan grid since it simplifies the problem, and it can be proven that

optimal solutions lie only on Hanan grid points if the unconstrained objective is to minimize the wirelength or a weighted sum of sink delays [9]. However, if we formulate the objective so as to satisfy the timing constraints, the optimal Steiner points are very likely to lie at non-Hanan grid points, as indicated in [10, 11]. The work of [10, 11] developed the MVERT algorithm, which exploits the piecewise concavity of delay violation functions to search for the optimal Steiner points. Its experimental results showed that expanding the solution space to non-Hanan points can significantly reduce the wire cost.

In this work, we continue the effort of non-Hanan optimization to deal with the condition where both timing and wire resources are stringent. We integrate buffer insertion and driver sizing with non-Hanan optimization in separate formulations to further improve the interconnect performance. These two approaches resemble each other in term of their algorithmic skeleton, although the nature of the problems is different.

Buffer insertion is a promising technique [21–26] that is essential for large nets. Most of the methods in [21–26] are implemented through dynamic programming in a bottom-up fashion. However, all of these methods have been restricted to only Hanan grid routing. Moreover, each of these approaches neglects the effects of restrictions to the buffer locations, i.e., it is assumed that buffers can be inserted in any arbitrary position as long as they can improve the interconnect performance. In real situations, this is not always permissible because the optimal buffer location may already have been occupied by other cells and it is undesirable to disturb the placement. Most recently, the work of [27] takes the restrictions to buffer locations into consideration and suggests an exact algorithm for two-pin nets. The problem environment we consider here is a limited set of buffer spaces where buffers are to be inserted into the interconnect after the placement stage. The con-

cept of soft edge is employed to increase the possibility that a buffer space is exploited. We guide each move in the optimization in a greedy fashion and conduct buffer insertion and non-Hanan optimization (BINO) simultaneously and iteratively until no further improvements are possible.

Another effort in our work is simultaneous driver sizing and non-Hanan optimization (FAR-DS). We have investigated the curvature properties of the delay as a function of the connection location and driver stage ratio in a two-dimensional space under the Elmore delay model. Though the Elmore model may be poor for specific points, it still provides a valid prediction of qualitative properties [6]. According to the solution region properties, we suggest two search schemes to find the optimal solution in the objective that can minimize a weighted sum of the wire cost and the driver cost, while satisfying the timing constraints. In both FAR-DS and BINO, we use a fourth order AWE delay model [8] to assure the integrity of the optimization.

3.2 Motivation for Using Fourth Order AWE

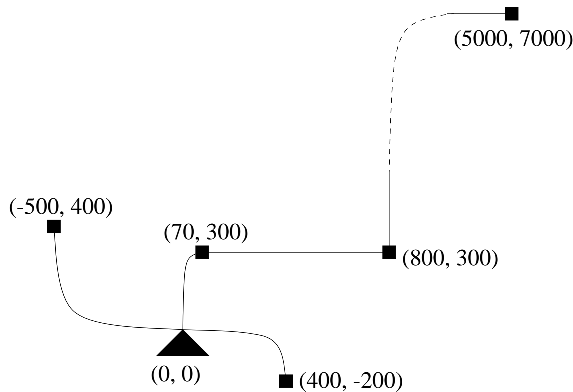


Figure 3.1: A routing tree on which Elmore delay gives large errors.

As interconnect wires become increasingly thinner and longer, the interconnect resistance may overshadow the driver resistance. Consequently, the

downstream capacitance is shielded to the driver resistance by the interconnect resistance. This effect is called resistive shielding [7]. The Elmore delay does not correctly take the resistive shielding effect into account and tends to overestimate the delay. This error can be remarkably large, especially for the stub situation (i.e., when a sink that is close to the source co-exists with a much longer wire), where the Elmore delay can be several times larger than the actual delay.

Table 3.1: A comparison of the Elmore and the 4th order AWE delays with SPICE.

Dist.	SPICE	Elmore	Error	4th AWE	Error
370	13.6	52.5	286%	12.8	-6%
600	9.5	39.8	319%	8.9	-6%
900	10.7	40.5	279%	10.5	-2%
1100	26.2	77.4	195%	25.5	-3%
12000	283.2	257.5	-9%	282.4	-0.3%

Table 3.1 shows an example of a net with five sinks to illustrate the inaccuracy of the Elmore delay. The routing topology of this net is illustrated in Figure 3.1. The load capacitance is the same for each sink. The delays at all sinks are computed using the Elmore formula, fourth order AWE and a SPICE transmission line model, and the percentage errors relative to SPICE are calculated. The Manhattan distance from each sink to the source are also listed for reference. We can see that the error of Elmore delay can be over 300% and the delay from fourth AWE is clearly superior. In fact, as the minimum feature size shrinks, this trend will become more and more severe.

To see how this will affect non-Hanan routing, consider the graph in Figure 3.2. The graph plots the delay violation function against the location of the connection point, z, as pictured in Figure 2.7. The dotted curve

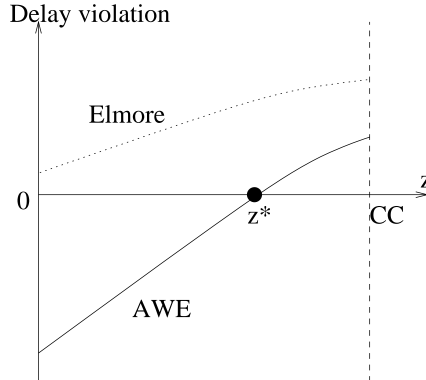


Figure 3.2: An example where using the Elmore delay and a higher order AWE delay may result in a different connection choice.

indicates the Elmore delay while the solid curve represents the fourth order AWE result. The solution corresponds to the point closest to CC where the delay violation function is negative or zero. For the Elmore delay, which overestimates the delay near the source, no solution is found, whereas an actual solution exists and corresponds to z^*.

On the other hand, we have observed that the Elmore model tends to under-estimate delay at sinks far from the source[1]. This may lead to the opposite error, as can be seen in the last row of Table 3.1. This under-estimation may result in over-reduction of cost while the timing constraints have not been satisfied yet. On the whole, a higher order model is greatly superior to the Elmore model in handling non-Hanan points.

In the computation of fourth order AWE delay, we first use the RICE algorithm [30] to obtain the moments. We solve the denominator of Padé approximation result, which is a fourth order polynomial, using a closed-

[1] The Elmore delay is theoretically proven to be an upper bound on the delay of an RC network in [28]. However, in practice, greater accuracies are obtainable by multiplying the Elmore delay formula of [29] by a factor of $\ln 2$, and we refer this quantity as the *Elmore delay* in our discussion, and this may be either optimistic or pessimistic.

form formula to obtain the poles. After an inverse Laplace transformation, the time-domain exponential functions are expanded about the Elmore delay to fourth order Taylor series polynomials. A closed-form solution to a fourth order polynomial exists and may be used to calculate the delay value. Since the Elmore delay may be far off from the correct value, sometimes the expansion about Elmore delay may still cause significant errors, though it is much smaller than the error from the Elmore delay. We restrict such errors by another iteration with expansion about the result from the first iteration. This process is iterated until convergence, and we found that we always converged within three iterations. This method is related to the Newton-Raphson root-finding method: the Newton-Raphson method uses a first order Taylor series in each iteration, and our method uses a fourth order expansion instead.

The reason that we choose fourth order instead of a second or third order model is that a second order yields less accuracy and for many examples that we tried, and we found that the third order model induces positive poles more often. The computation overhead for models with orders greater than four is large, since there is no closed form solution for equations with order beyond four. The additional computation cost of fourth order AWE as compared to a second order model is minor.

3.3 The Problem Environment and Problem Formulation for BINO

The BINO algorithm is applied in a post-placement scenario where buffer insertion is possible, but it is preferable to do so in regions that are left unoccupied by any cells, so as not to disturb the placement. This method is

also applicable to MCM technology, where a buffer location is desired to be within a chip and close to its chip bond pads, because it is not cost-effective to insert a buffer either on the substrate between chips or within a chip but far from any bond pad. The input to BINO then includes a set of pre-defined available buffer spaces scattered in the routing region. These buffer spaces are represented by small squares, as demonstrated by the dark grey areas b_1 and b_2 in Figure 3.3 (a). It is assumed that only one buffer can be inserted in each space and the center of the buffer must lie within the square. Larger buffer spaces can easily be expressed as a union of small spaces.

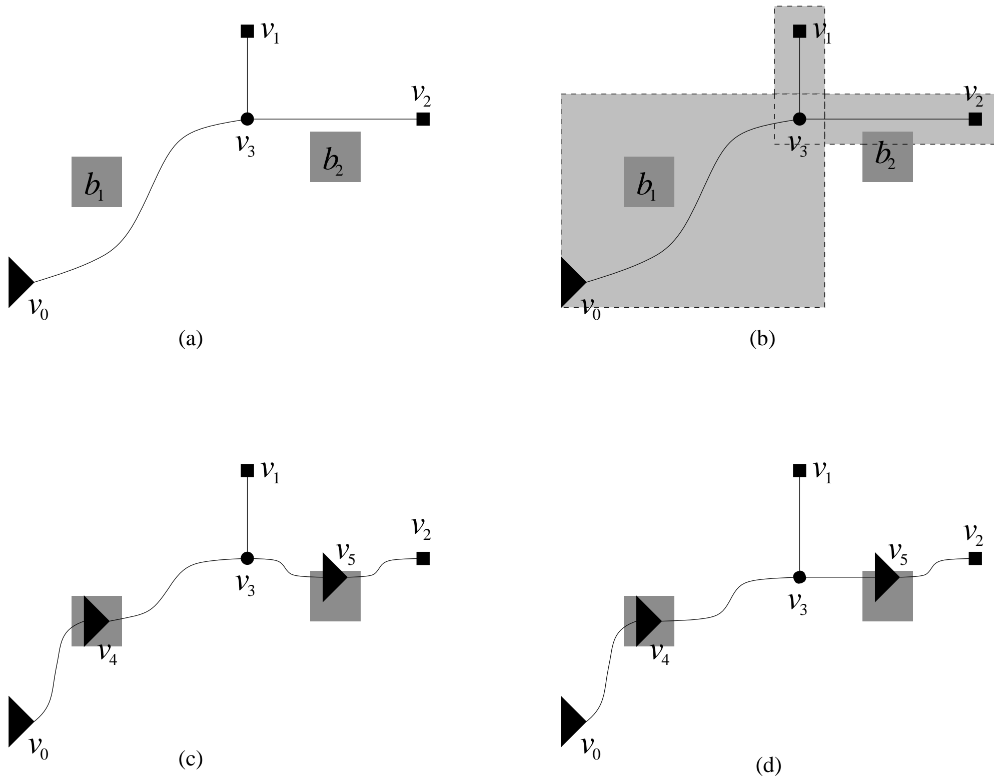


Figure 3.3: Buffer spaces, the territory box and their applications in buffer insertion.

Intuitively, a buffer space is considered for buffer insertion only when a routing path passes through it, since no extra wire cost is incurred under this

condition. However, even if no path passes through a buffer space, it may be worthwhile for the wire to make small detour to increase the possibility of exploiting a buffer space. Based on this idea, we define a territory box for an edge as follows:

Definition 3.1 (territory box) *For an edge* (v_i, v_j)*, its territory box is a rectangle specified by lower-left corner point* (x_{min}, y_{min}) *and upper-right corner point* (x_{max}, y_{max})*, such that:*

$$x_{min} = \min(x_i, x_j) - \phi,$$

$$y_{min} = \min(y_i, y_j) - \phi,$$

$$x_{max} = \max(x_i, x_j) + \phi,$$

$$y_{max} = \max(y_i, y_j) + \phi,$$

where ϕ *is a small amount of offset.*

The idea of a territory box is demonstrated by the light grey regions in Figure 3.3(b). Note that the territory box for the soft edge (v_0, v_3) is larger than for any solid edges between v_0 and v_3. The rule that we will follow is as follows: *a buffer space is considered for buffer insertion in an edge only when there is an overlap between this buffer space and the territory box of this edge.* In the example of Figure 3.3, buffer space b_1 overlaps with the territory box of edge (v_0, v_3) and b_2 overlaps with the territory box of (v_3, v_2); therefore, we can insert buffers v_4 and v_5 as in Figure 3.3(c). After the non-Hanan optimization following the buffer insertion, the wire slack in Figure 3.3(c) may be removed and the tree shown in Figure 3.3(d) may be obtained.

This example shows that the use of soft edges can greatly increase the possibility of overlaps as compared to using predetermined L-shaped connection composed of two solid edges.

We consider both inverting and non-inverting type buffers in our work. The inverting type buffer is simply an inverter and the non-inverting type buffer is composed of a pair of cascaded inverters. The inverter model is the same as the driver model in Chapter 2 and has a medium driver size.

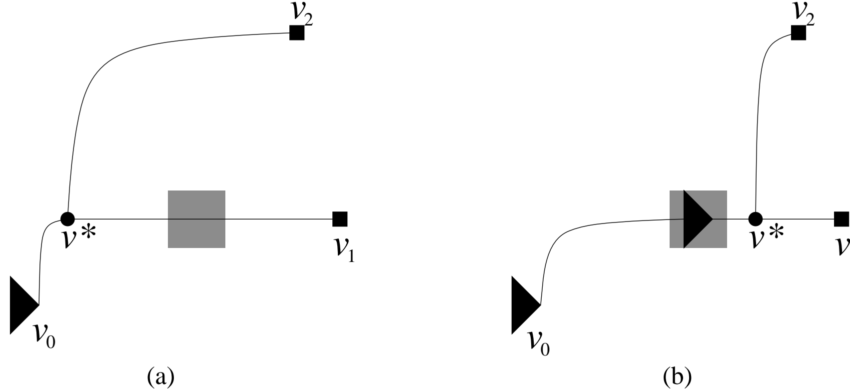


Figure 3.4: An example that buffer insertion can reduce wire cost further in non-Hanan optimization.

The motivation for combining buffer insertion with non-Hanan optimization can be illustrated by the example in Figure 3.4. In order to reduce wire cost, it is desired to move the connection point as close to CC as possible, i.e., to maximize z. However, the value of z may be capped by the constraint of non-positive delay violation as illustrated in Figure 3.4(a). The utility of buffer insertion is to relax this timing constraint, if possible, so as to achieve further wire cost reduction as in Figure 3.4(b).

We use u_i to represent the delay violation at sink v_i. The gate and drain capacitance of an inverting buffer are denoted as C_{gb} and C_{db}. The total wirelength is represented as W and γ is the weighting factor for the wire cost. The simultaneous buffer insertion and non-Hanan optimization problem is to minimize a weighted sum of buffer and wire cost subject to timing constraints. This is formulated as follows.

Problem 3.1 *Given a source* v_0*, a set of sinks* $V_{sink} = \{v_1, v_2 ... v_n\}$*, timing specifications* $Q = \{q_1, q_2, ..., q_n\}$ *for all sinks and a set of available buffer spaces* $P = \{p_1, p_2, ..., p_m\}$*, construct a Steiner routing tree and choose a subset* $B_{iv} \subseteq P$ *and* $B_{ni} \subseteq P$ *on which inverting and non-inverting buffers are inserted, respectively, such that the following is solved:*

$$\begin{array}{lc} \textit{minimize} & \gamma c W + (1-\gamma)(C_{gb} + C_{db})(|B_{iv}| + 2|B_{ni}|) \\ \textit{subject to:} & \max_{v_i \in V_{sink}}(u_i) \leq 0 \\ \textit{for a specific } \gamma & 0 \leq \gamma \leq 1 \end{array} \quad (3.1)$$

The purpose of including c (wire capacitance per unit length), C_{gb} and C_{db} in the objective function is to normalize the wire and the buffer cost into comparable quantities.

3.4 Problem Formulation and Properties for FAR-DS

3.4.1 Problem Formulation for FAR-DS

The driver sizing problem is to choose optimal number of driver stages h and the proper size for each driver. We choose the ratio of driver size at one stage to its previous stage to be uniform and refer to it as the stage ratio ρ.

The objective of FAR-DS is to minimize the cost of the routing tree, subject to a timing constraint at each sink. In contrast to [10], we extend the cost here to include both wire cost and driver cost, i.e., we perform topology optimization and driver sizing simultaneously. The rationale behind this is to permit the driver to share the task of delay optimization with the

interconnect by sizing it, thereby obtaining a better result than optimizing the driver size and interconnect topology separately.

We formally state the problem formulation as follows:

Problem 3.2 *Given a source v_0, a set of sinks $V_{sink} = \{v_1, v_2...v_n\}$, timing specifications $Q = \{q_1, q_2, ...q_n\}$ for all sinks, and stage ratio bound ρ_{max}, construct a Steiner routing tree and find ρ, h such that:*

$$\begin{array}{ll} \textit{minimize} & \gamma c W + (1-\gamma)(C_g + C_d)\sum_{j=1}^{h} \rho^j \\ \textit{subject to:} & \max_{v_i \in V_{sink}}(u_i) \leq 0 \\ \textit{and} & 1 \leq \rho \leq \rho_{max}. \end{array} \tag{3.2}$$

The second term in the objective function represents the total driver capacitance. The objective function can be interpreted as a minimization of the total wirelength and total driver capacitance. The parameter γ is a user-specified weighting factor.

3.4.2 Properties of Solution Regions for FAR-DS

For a general connection of a node and its downstream subtree to a partial tree, as illustrated in Figure 2.6, where a node v_k is to be connected to an edge (v_i, v_j), we investigate the properties of the delay violation function with respect to z and ρ in a two dimensional space. If the number of stages is h, the delay from the first stage to the last stage of the cascaded drivers is given by:

$$T_D = hR_0(C_d + \rho C_g) \tag{3.3}$$

Through a few algebra steps, we can combine the interconnect delay discussed in Section 2.4 with T_D to obtain a general form of the delay violation

of any sink u_i as a function of the connection position z and ρ, under the Elmore model as:

$$u_i = f(z, \rho) = -a_2 r c z^2 + \frac{R_0(C_t - cz)}{\rho^h} + a_1 z + R_0 C_g h \rho + a_0 \tag{3.4}$$

where

$$a_2 = 0 \text{ or } 1, \quad 0 \leq z \leq CC < \tfrac{C_t}{c}, \quad 1 \leq \rho \leq \rho_{max}, \tag{3.5}$$

with a_0 and a_1 being constants. The parameter C_t is the total load capacitance seen by the driver in the last stage when v_k is connected to v_i directly.

When ρ is fixed, $u_i = f(z)$ is a quadratic function of z and the coefficient of the second order term is non-positive. Therefore we can obtain the following result:

Property 3.1 *$u_i = f(z, \rho)$ is a concave function for a constant value of ρ.*

If we keep z constant, there are also properties that will help the search for the optimal solution. These properties can be found by investigating the partial derivatives of u_i with respect to ρ as follows:

$$\frac{\partial u_i}{\partial \rho} = -R_0(C_t - cz) h \rho^{-h-1} + R_0 C_g h \tag{3.6}$$

$$\frac{\partial^2 u_i}{\partial^2 \rho} = R_0(C_t - cz) h(h+1) \rho^{-h-2} \tag{3.7}$$

Since $C_t > cz$, $\frac{\partial^2 u_i}{\partial^2 \rho} > 0$ is always true, thus we have the following property:

Property 3.2 *$u_i = f(z, \rho)$ is convex function for a constant value of z.*

If we let $\frac{\partial u_i}{\partial \rho} = 0$, we can obtain a curve defined as follows:

$$\rho = \sqrt[h+1]{\frac{C_t - cz}{C_g}} \tag{3.8}$$

Property 3.3 *$f(z, \rho)$ has minimum value along the curve defined by equation (3.8).*

This property is especially useful in solution search, since it predicts the bottom of the valley shaped delay violation function surface in the two-dimensional space of z and ρ. One observation is that the curve in equation (3.8) is independent of which sink is considered, i.e., equation (3.8) defines the bottom of valley for the delay violation functions of *all* the sinks. We call the curve defined by equation (3.8) the *valley curve* for delay violations.

In equation (3.8), when z is at CC, the numerator reaches the minimum and becomes the total load capacitance seen by the driver in the last stage when v_k is connected to CC. Obviously, this total load capacitance is always greater than the minimum gate capacitance, C_g, of a driver. This fact provides the following property:

Property 3.4 *If $0 \leq z \leq CC$, then $\rho = \sqrt[h+1]{\frac{C_t - cz}{C_g}} > 1$.*

If we substitute equation (3.8) into equation (3.4), we can obtain another important conclusion:

Property 3.5 *$u_i = f(z, \rho)$ is a concave function of z along the curve defined by equation (3.8).*

This valley curve also sets a border for different monotone properties with respect to ρ as follows:

Property 3.6 *For a specific z, $f(z, \rho)$ is a monotone decreasing function of ρ when $\rho \leq \sqrt[h+1]{\frac{C_t - cz}{C_g}}$.*

These properties are derived from Elmore delays. Though the Elmore delay may have large errors for specific points, its qualitative fidelity is still true [6] and can serve as a good strategic guide. Our experimental results also support this assertion.

3.5 Algorithms

Both BINO and FAR-DS consist of two phases. Phase I is the routing tree construction process, which is the same for BINO and FAR-DS. This procedure is called SART (Steiner AWE Routing Tree), and is similar to SERT [6] except that the Elmore model is replaced by a fourth order AWE model and soft edges are employed.

In SART, starting with a single source, a partial routing tree is grown in a greedy fashion. In each growing step, a previously unconnected sink is selected and connected to an edge in the partial tree such that the maximum delay is minimized.

Phase II of BINO and FAR-DS are different, but share a similarity in the non-Hanan optimization framework described in Section 2.4.

3.5.1 Algorithm for BINO

Algorithm: BINO_IterativeBufferInsertion
Input: SART $T(V, E)$, a set of buffer spaces P
Output: Buffered and non-Hanan optimized routing tree T
1. While $P \neq \emptyset$ and $\exists$ improvement
2. For each $p \in P$
3. For each edge $(v_i, v_j) \in E$
4. If p overlaps with the territory box of (v_i, v_j)
5. Insert a buffer into (v_i, v_j) at p tentatively
6. Assign inverting/non-inverting type $\forall$ buffers $\in T$
7. Perform non-Hanan optimization for T (Figure 2.8)
8. Insert buffer at p_{best}, which gives the largest improvement
9. $P \leftarrow P - p_{best}$

Figure 3.5: BINO, iterative buffer insertion algorithm.

In BINO, the non-Hanan optimization framework is embedded in a greedy buffer insertion scheme illustrated by Figure 3.5. On each buffer space, we insert a buffer tentatively and conduct non-Hanan optimization. After all of the buffer spaces have been tested, the solution that can provide the largest improvement is chosen as the final decision. This process is repeated iteratively until there is no improvement or no buffer space is left. The optimal solution of assigning inverting or non-inverting type to each buffer (line 6 in Figure 3.5) can be achieved through dynamic programming.

Since we only insert one buffer in each iteration, the ability to obtain an optimal buffer insertion solution is hindered, as shown by the single-sink example in Figure 3.6. It is well known that optimal buffer locations often distribute evenly along an interconnect path [24]. Therefore, for the net in Figure 3.6, the optimal solution may be as shown in Figure 3.6 (d). If we insert only one buffer in an iteration, the first iteration is likely to result in the scenario shown in Figure 3.6 (b) and the optimal solution cannot be

reached.

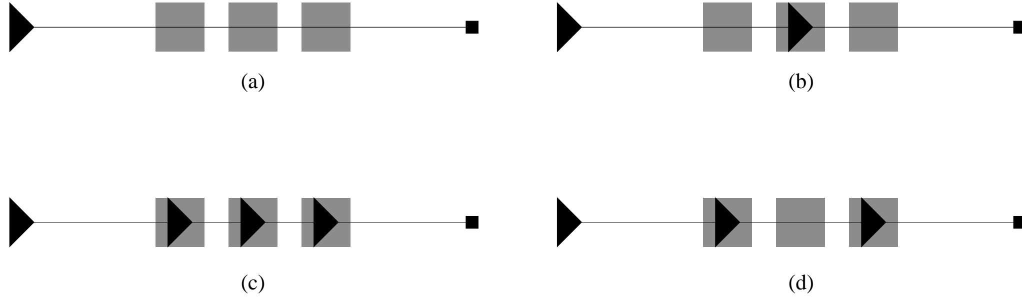


Figure 3.6: Iterative buffer insertion vs. iterative buffer deletion.

In order to alleviate the above difficulty, we supplement the method with an iterative buffer deletion procedure using a method similar to [18], that is described in Figure 3.7. In this scheme, we first insert buffers at all spaces that overlap with any edges. Then we delete one buffer in each iteration in a greedy fashion similar to iterative buffer insertion. Since this proceeds in the opposite direction as compared to the iterative buffer insertion, it plays a complementary role. For the example in Figure 3.6, the iterative buffer deletion starts with (c) and can naturally result in the optimal solution in (d). On the other hand, if the optimal solution is (b), iterative buffer deletion is worse than iterative buffer insertion.

In our work, we perform both iterative buffer insertion and iterative buffer deletion independently for a net and choose the better of the two results.

3.5.2 Algorithm for FAR-DS

In Phase II of FAR-DS, a two-dimensional search replaces the role of the quasi-binary-search in MVERT, which is line 6 in Figure 2.8, to find an optimal connection point and driver size simultaneously.

When we reconnect a node v_k to an edge (v_i, v_j), we look for a 3-tuple

Algorithm: BINO_IterativeBufferDeletion
Input: SART $T(V, E)$, a set of buffer spaces P
Output: Buffered and non-Hanan optimized routing tree T
1. $B \leftarrow \emptyset$
2. For each $p \in P$
3. For each edge $(v_i, v_j) \in E$
4. If p overlaps with the territory box of (v_i, v_j)
5. Insert buffer b into (v_i, v_j) at p
6. $B \leftarrow B \cup b$
7. Assign inverting/non-inverting type $\forall b \in B$
8. While $B \neq \emptyset$ and $\exists$ improvement.
9. For each buffer $b \in B$
10. Remove b from T tentatively
11. Assign inverting/non-inverting type $\forall b \in B$
12. Perform non-Hanan optimization for T (Figure 2.8)
13. Remove buffer b_{best}, which gives the largest improvement
14. $B \leftarrow B - b_{best}$

Figure 3.7: BINO, iterative buffer deletion algorithm.

(z, ρ, h) such that the objective function of Problem 3.2 is minimized while the delay violations for all sinks are non-positive. We vary h between 1 and h_{max} and search an optimal (z, ρ) pair in a two dimensional plane for a fixed h value.

For this case, $cW = C_t - cz$ and the objective in Problem 3.2 can be translated to:

$$\begin{array}{ll} \text{minimize} & g = -\gamma cz + (1-\gamma)(C_g + C_d)\sum_{j=1}^{h} \rho^j \\ \text{subject to:} & \max_{v_i \in V}(u_i) \leq 0 \\ \text{and} & 0 \leq z \leq CC,\ 1 \leq \rho \leq \rho_{max}. \end{array} \quad (3.9)$$

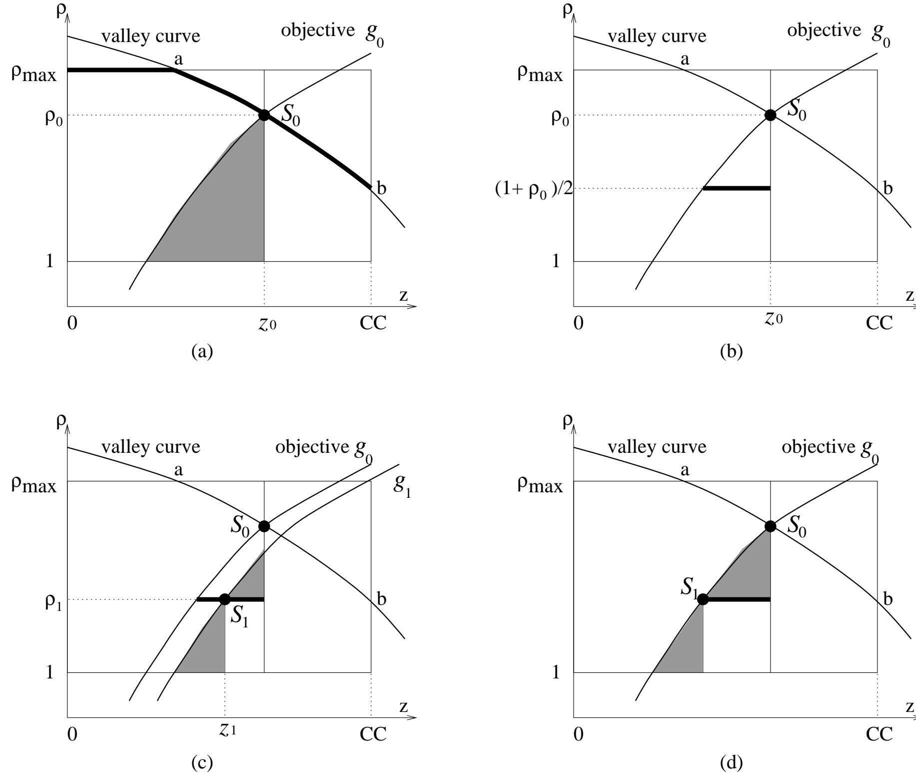


Figure 3.8: Solution search scheme for FAR-DS.

For a specific value of g, the objective function above corresponds to a curve in the (z, ρ) plane, as the objective curves shown in Figure 3.8. The objective (3.9) can be interpreted as to find a point in (z, ρ) plane such that the constraints in (3.9) are satisfied at this point and the point is on a objective curve as low as possible.

We will illustrate the optimal solution search scheme through Figure 3.8. The solution search can be restricted within the rectangle bounded by $0 \leq z \leq CC$ and $1 \leq \rho \leq \rho_{max}$. Consider the valley curve defined by equation (3.8). This curve is always above $\rho = 1$ in the interval $0 \leq z \leq CC$, according to Property 3.4 . One common scenario is that this valley curve intersects with upper border of the rectangle at a point a and with the right border at b, as in Figure 3.8(a). From Property 3.3 and Property 3.6, we can say

that in the rectangle defined above, u_i reaches its minimum on the segment $\rho = \rho_{max}$ to the left of a, and on valley curve specified by equation (3.8) to the right of a. These two segments can be integrated into a single function:

$$\rho = \min(\rho_{max}, \sqrt[h+1]{\frac{C_t - cz}{C_g}}), 0 \leq z \leq CC \tag{3.10}$$

which is the thickened line in Figure 3.8 (a). Note that equation (3.10) is valid even when the valley curve does not intersect the rectangle, or if the set of points on the segment to the left of point a is empty. This function provides us with a convenient way to check for the existence of a solution within the rectangle. From Property 3.3 and Property 3.6, if no solution that satisfies all constraints exists on the curve defined by equation (3.10), then we can say that no solution exists within the rectangle. According to Property 3.1 and Property 3.5, u_i is a concave function on the curve (3.10), both to the left and to the right of point a. Thus, we can apply the quasi-binary-search technique in [10,11] to search for the rightmost solution on this curve that satisfies all constraints. If such a solution exists, we call it the zero order solution, designated as $S_0(z_0, \rho_0)$ in Figure 3.8(a).

After the zero order solution has been found, the region can be further refined to search for the optimal solution. This is demonstrated in the shaded region in Figure 3.8 (a). The region $z > z_0$ can be excluded, since no feasible point exists on the valley curve in this region. An objective function curve is drawn through S_0, which satisfies:

$$g_0 = -\gamma c z_0 + (1 - \gamma)(C_g + C_d) \sum_{j=1}^{h} \rho_0^j \tag{3.11}$$

We can eliminate the region that lies above this curve, because the value of g at all points above this line exceeds g_0, and hence inferior to the currently

minimum value g_0. The remainder of the search space is the sector confined by the objective function curve defined by (3.11), by $z = z_0$ and by $\rho = 1$, which is indicated by the shaded region in Figure 3.8(a).

Algorithm: FAR-DS_ReconnectVSearch
Input: Routing tree $T \backslash T_k$, subtree T_k, node v_k, edge e_{ij}
Output: Optimal connection between v_k and (v_i, v_j), ρ and h
1. For $h = 1$; $h \leq h_{max}$; h++;
2. Search solution along valley curve defined by equation(3.10)
3. If no solution found, return
4. $S_0(z_0, \rho_0) \leftarrow$ rightmost solution
5. SearchSector$(S_0, 1)$
Function: SearchSector(S_{top}, ρ_{base})
F1. Obtain curve $g_{top} = -\gamma c z_{top} + (1-\gamma)(C_g + C_d) \sum_{j=1}^{h} \rho_{top}^{j}$
F2. $\rho_{mid} = (\rho_{top} + \rho_{base})/2$
F3. $S_{mid}(z_{mid}, \rho_{mid}) \leftarrow$ intersection between curve g_{top} and $\rho = \rho_{mid}$
F4. Search solution along $\rho = \rho_{mid}$ between z_{mid} and z_{top} by quasi-binary-search
F5. If no solution found
F6. If $\rho_{mid} - \rho_{base} < resolution$, return no solution
F7. Return $SearchSector(S_{top}, \rho_{mid})$ and $SearchSector(S_{mid}, \rho_{base})$
F8. Else
F9. If $\rho_{mid} - \rho_{base} < resolution$, return rightmost solution
F10. $S_{mid}(z_{mid}, \rho_{mid}) \leftarrow$ rightmost solution
F11. Obtain curve $g_{mid} = -\gamma c z_{mid} + (1-\gamma)(C_g + C_d) \sum_{j=1}^{h} \rho_{mid}^{j}$
F12. $S_{top}(z_{top}, \rho_{top}) \leftarrow$ intersection between curve g_{mid} and $z = z_{top}$
F13. Return $SearchSector(S_{top}, \rho_{mid})$ and $SearchSector(S_{mid}, \rho_{base})$

Figure 3.9: FAR-DS, reconnection and driver sizing in valley-guided search.

The search within this sector also proceeds in a binary search fashion, by starting from the middle segment defined by $\rho_1 = (1 + \rho_0)/2$, which is the thickened segment in Figure 3.8 (b). On this segment, Property 3.1 holds and

a quasi-binary-search can again be applied to obtain the rightmost solution on it, namely, $S_1(z_1, \rho_1)$; we refer to this as the first order solution. After the first order solution has been found, the previously described solution refinement technique can be used to obtain two new smaller sectors shown by the shaded regions in Figure 3.8 (c) where the optimal solution will be searched. Even if there is no solution on this segment, the search region can be refined to the two sectors like in (d). We call this solution search scheme as valley-guided search (V-search), and describe it in Figure 3.9.

The above is the method to search optimal (z, ρ) for a specific h. The optimal h is found by a sweep from $h = 1$ to $h = h_{max}$ and the above search is carried out for each h value. The value of h_{max} is given by [31]:

$$h_{max} = \left\lceil \frac{\ln(C_t/C_g)}{\ln \rho^*} \right\rceil, \tag{3.12}$$

$$\ln \rho^* = 1 + \frac{C_d}{C_g \rho^*}. \tag{3.13}$$

Since the use of valley curve increases the dependency of the solution on the Elmore delay model, and we use a higher order AWE model to evaluate the delays for every sink in our algorithm, it is possible that the discrepancy between Elmore model prediction and the actual AWE evaluation may give rise to a suboptimal solution.

We suggest an alternative search method called the iterative search (I-search) scheme that does not depend on Elmore model quantitatively and illustrate it in Figure 3.10. In this method, we begin with an initial ρ and perform non-Hanan optimization to obtain an optimal z for this value of ρ. Next, this z is fixed and an optimal ρ is searched and so on. This process is repeated until there is no further improvement. From Property 3.2, we know

that the delay violation function u_i is a convex function along ρ direction, thus, we cannot apply the quasi-binary-search suggested by [10] along ρ direction. We perform the search in a manner between binary search and linear search. If the maximum delay violation is non-positive for a specific value of ρ, we continue to search a better solution at a smaller ρ value, otherwise, we must search at both larger and smaller values.

Algorithm: FAR-DS_ReconnectISearch
Input: Routing tree $T \backslash T_k$, subtree T_k, node v_k, edge (v_i, v_j)
Output: Optimal connection between v_k and (v_i, v_j), ρ and h
1. For $h = 1$; $h \leq h_{max}$; h++
2. $\rho \leftarrow$ initial guess
3. While $\exists$ improvement
4. Search z_{best} which gives best improvement while ρ is fixed
5. $z \leftarrow z_{best}$
6. Search ρ_{best} which gives best improvement while z is fixed
7. $\rho \leftarrow \rho_{best}$

Figure 3.10: FAR-DS, reconnection and driver sizing in iterative search.

3.6 Complexity Analysis

From [11], the computation cost for MVERT is $O(n^4 + n^4 \cdot \frac{L}{\epsilon})$. The first term corresponds to the Phase I in MVERT, which is a variation of SERT. The parameter L is the maximum length over all edges and ϵ represents the resolution for the quasi-binary-search in the Phase II of MVERT. Since the quasi-binary-search may fall into a linear search in the worst case, there is

no logarithmic term here.

Although we use the fourth order AWE instead of Elmore in BINO, as the number of iterations is fixed, the complexity for each delay calculation remains $O(n)$. Thus the cost for Phase I (SART) in BINO is $O(n^4)$. In Phase II of BINO, there are two layers of iterations outside of each non-Hanan optimization, each of which is upper-bounded by the number of buffer spaces. The combination of the total cost is $O(m^2 \cdot n^4 \cdot \frac{L}{\epsilon})$. This conclusion is true for both iterative buffer insertion and iterative deletion.

The complexity of FAR-DS is same as MVERT in the outer loops. The difference is in the computation cost of reconnection part (line 6 of Fig. 9), where FAR-DS performs a search in the entire (z, ρ) space. The computation factor from searching along the ρ direction is bounded by $(\rho_{max} - 1)/\tau$, where τ is the resolution on ρ. Since the h value is swept from 1 to h_{max}, the complexity of FAR-DS is $O(h_{max} \cdot n^4 \cdot \frac{L}{\epsilon} \cdot \frac{\rho_{max}}{\tau})$.

The above results only provide a loose bound, because the worst case for the quasi-binary-search along the z direction is almost impossible in practice. Therefore, the computation cost in average case is one order lower than the above theoretic results.

3.7 Experimental Results

The experiments are emphasized to test the improvement from our algorithms in terms of both timing and cost objectives defined in the problem formulation. Each signal net is randomly generated and tested for BINO and two FAR-DS algorithms, as well as SERT and MEVRT for comparison. In order to obtain a more general conclusion, we include both IC and MCM

technology in the experiments and the number of sinks ranges from 5 to 20. To form a common base for comparisons with FAR-DS, we use cascaded drivers also in SERT, MVERT and BINO, and choose $h = 1$ and $\rho = 2.5$, which can provide a middle level of driving ability. For all of the timing results, driver and wire delays are calculated from RC and a fourth order AWE model, respectively.

The experimental results are shown in Table 3.2 and Table 3.3 for IC and MCM technology, respectively. The parameters for MCM are from [6]. The IC parameters correspond to 0.18 μm technology and are scaled from the data in [6]. The buffer space locations for BINO are generated randomly. The area of each buffer space is chosen to be $100\mu m \times 100\mu m$ for IC and $200\mu m \times 200\mu m$ for MCM. According to our experiments, the variations of delay from the change of a buffer position within a buffer space is small and can be neglected. The offset ϕ for the territory box of an edge is set to be half of the buffer space size. In the experiment for FAR-DS, the value of ρ_{max} was chosen as 4 for both IC and MCM technology. Since we consider the situation where the interconnect resources are more stringent, the weighting factor for wire cost is chosen to be 0.7 for both BINO and FAR-DS.

The parameter W is the total wirelength and the u_{max} is the maximum delay violation according to the fourth order AWE model results. The column labeled m corresponds to the number of input buffer spaces, and to its left is the number of buffers, $|B|$, finally inserted. The last row provides the percentage change of total wirelength compared to the result of SERT. The number of sinks for each test is given in column 3 of Table 3.4. The CPU time in seconds for BINO and FAR-DS are also listed in Table 3.4.

Since the timing constraints are quite stringent, most of the maximum delay violations, u_{max}, from the results of SERT are positive. Sometimes

Table 3.2: Experimental results on .18μm IC, $h = 1, \rho = 2.5$ for SERT, MVERT and BINO.

	SERT		MVERT		BINO			FAR-DS: I-search/V-search			
net	W	u_{max}	W	u_{max}	W	u_{max}	$\|B\|/m$	W	u_{max}	ρ	h
I1	226	4.51	243	-1.19	189	-9.48	3/6	187/187	0/0	2.9/2.9	3/3
I2	219	-1.26	166	1.31	153	-2.78	1/7	155/155	-1.70/-0.02	4.0/2.5	1/1
I3	292	-2.83	280	-1.03	231	-5.31	1/8	235/235	-0.01/-0.07	2.6/2.7	3/3
I4	374	-0.82	341	-0.54	280	-1.69	2/8	295/282	-0.07/-0.41	1.8/1.9	3/3
I5	310	-0.45	257	1.48	249	-2.87	1/7	252/252	-0.02/-0.33	2.9/3.1	1/1
I6	401	1.92	381	0.91	348	-4.32	1/8	349/349	-1.85/-0.02	4.0/3.1	1/1
I7	462	7.57	383	5.94	367	-2.28	2/9	370/370	-0.21/-0.72	1.7/1.8	3/3
I8	626	4.37	555	-0.25	470	-0.27	2/10	494/494	0/-0.09	2.1/3.6	5/3
I9	595	10.84	537	8.26	469	-0.02	3/10	488/495	-0.01/-0.43	3.4/1.9	3/3
I10	564	9.05	481	4.88	433	-1.60	2/11	452/452	-0.06/-0.47	1.9/1.9	3/3
Ave $\Delta W(\%)$			-10		-22			$-20/-20$			

MVERT even results in a worse delay violation than SERT, due to errors from the Elmore delay model. In other cases, the improvements from MVERT on both delay and wire cost are limited and the timing constraints are often unsatisfied, because the specification is unachievable without driver sizing or buffer insertion. This hinders the ability of pure non-Hanan optimization to reduce the cost further and BINO or FAR-DS becomes a necessary step. Both BINO and FAR-DS can also satisfy the timing constraints that are impossible for SERT and MVERT.

Besides timing improvement, we can see that BINO and FAR-DS can reduce significantly more cost than MVERT under these somewhat harsh conditions. Sometimes MVERT may even increase the wirelength to meet the timing constraint which can be seen from the result of the first net in IC technology. The results from the two different variants of FAR-DS have no significant differences.

Comparing the experimental results from BINO and FAR-DS, we can see that BINO can provide more wire cost reduction than FAR-DS in most cases,

Table 3.3: Experimental results on MCM, $h = 1, \rho = 2.5$ for SERT, MVERT and BINO.

	SERT		MVERT		BINO			FAR-DS: I-search/V-search			
net	W	u_{max}	W	u_{max}	W	u_{max}	$\|B\|/m$	W	u_{max}	ρ	h
M1	444	1.82	427	0.37	332	0	1/6	340/347	0/-0.14	1.7/1.8	3/3
M2	429	1.21	410	1.17	332	-0.71	1/7	376/357	-0.02/-0.07	1.8/4.0	3/1
M3	478	-0.57	454	0.17	381	-4.91	1/6	412/405	0/0	2.2/2.5	3/3
M4	617	4.14	539	3.59	480	-2.10	1/10	506/506	-0.04/-0.11	1.9/3.3	3/1
M5	624	-0.55	505	1.08	465	-0.96	2/7	482/482	-0.14/-0.01	4.0/3.8	1/1
M6	618	0.99	519	-0.07	486	-2.84	1/8	480/480	-2.01/-0.61	4.0/2.7	1/1
M7	810	3.80	755	2.43	652	-0.64	2/10	706/698	-0.04/0	4.0/3.4	1/1
M8	797	-0.99	695	0.37	656	-0.45	1/9	660/660	-0.01/-0.01	3.7/3.9	1/1
M9	1253	4.69	1149	4.06	900	-0.30	2/12	962/979	-0.12/-0.36	1.9/2.1	3/3
M10	1025	3.65	883	3.13	791	-1.91	2/8	826/862	-0.03/-0.10	3.7/2.7	1/3
Ave $\Delta W(\%)$			-10		-23			$-19/-19$			

and the larger timing slacks from BINO also indicate its potential on dealing with even more stringent timing constraints. Although FAR-DS is not so powerful as BINO, it shows an adaptive nature that can often trade off the timing slack into less driver cost. This is especially true for the *V-search* scheme of FAR-DS, whose most timing slacks are close to zero.

These experiments were carried out on a SUN Ultra-10 station. The computation time of FAR-DS mostly depends on the size of signal nets while the CPU time of BINO is more irregular because it also depends on the number of buffer spaces overlapping with the routing tree. In most cases, the CPU time is within one minute. In the worst case for a net of 20 sinks, the run time is less than four minutes for both FAR-DS and BINO. On the whole, the computational cost of our algorithm is reasonable, since these optimizations are carried out only for global timing-critical nets.

We also perform experiments to check the effect from using soft edges on the same set of nets and the result is shown in Table 3.5. This result confirms the conclusion that using soft edges can greatly increase the possibility that

Table 3.4: Number of sinks and CPU times.

tech	net	n	CPU BINO	CPU FAR-DS:I	CPU FAR-DS:V
IC	I1	5	9	2	2
	I2	5	6	2	3
	I3	5	3	3	3
	I4	10	24	13	19
	I5	10	14	19	25
	I6	10	33	14	19
	I7	15	61	35	49
	I8	15	86	64	86
	I9	20	204	119	166
	I10	20	88	79	104
MCM	M1	5	12	3	4
	M2	5	5	3	5
	M3	5	3	3	5
	M4	10	38	16	19
	M5	10	9	17	24
	M6	10	34	14	18
	M7	15	81	43	63
	M8	15	78	42	53
	M9	20	204	121	153
	M10	20	105	84	124

buffer spaces overlaps with routing edges.

3.8 Conclusion

When we extend the non-Hanan optimization to improve the performance of critical nets where both timing and wire resources are stringent, buffer insertion is shown to be a strong augmentation to the timing optimization toolkit, even with location restrictions. A combination of driver sizing and non-Hanan optimization can provide a continuous two-dimensional space. A search for the optimum in this space may be guided by properties derived

Table 3.5: Comparison of the number of overlaps between buffer spaces and routing edges with and without using soft edges

# nets	# overlaps w/o soft edge	# overlaps w soft edge
20	34	71

from the Elmore delay model, which may have large quantitative errors but good qualitative fidelity. These properties are used to direct heuristics that use a fourth order AWE model for wire delay calculation. For drivers, a more accurate model can be applied in place of an RC switch model in a similar fashion. Experimental results show that both BINO and FAR-DS can bring both timing and wire cost improvements significantly.

Chapter 4

Routing for Buffer Blockages and Bays

4.1 Introduction

Buffer insertion has become a necessary step in modern VLSI design (see Cong *et al.* [32] for a survey), especially for interconnect performance optimization. To the first order, interconnect delay is proportional to the square of the length of the wire. Buffer insertion effectively divides the wire into smaller segments, which makes the delay almost linear in terms of wire length. Additional advantages of buffer insertion, such as noise avoidance [33], will make this optimization even more pervasive as the ratio of device to interconnect delay continues to decrease.

Several works have studied the delay-driven buffer insertion problem. Closed form solutions have been proposed in [24, 34–36]. Van Ginneken's algorithm [21] is perhaps the best known work on buffer insertion. His dynamic programming algorithm finds the optimal buffer placement under the

Elmore delay model. Several extensions to this work have been proposed (e.g., [22,25,33,37–39]). All of these works (except for [22,25]) assume that a Steiner tree is given and that buffers must be placed along the Steiner wires. The works of [22, 25] also perform routing of the tree during buffer insertion but do not consider blockages.

When attempting to insert buffers into a floorplanned design (especially a hierarchical one), buffers may not be placed on top of pre-existing macros or blocks; we refer to these regions as *blockages*. If the existing Steiner tree has been routed almost entirely over blockages, then any buffer insertion algorithm that uses the routing topology will fail to find a solution. Figure 4.1(a) shows an example 2-pin net whose route runs over a large blockage, thereby making buffer insertion infeasible. However, buffers may be necessary not only to improve timing, but also to meet target slew and capacitance constraints. If one re-routes the tree as in Figure 4.1(b), then buffers can be judiciously inserted, albeit for an additional wire length cost.

In this methodology, the Steiner tree serves as a guide for buffer insertion, but does not represent the final route. The actual routing from the original pins to the buffers is performed after buffer insertion by the global router. Figure 4.1(c) shows how the global router may re-route the newly created nets while considering delay, noise, congestion, etc. Without this final step, the regions of the chip without blockages would become unnecessarily congested with interconnect.

Sometimes, the best Steiner tree construction will avoid some, but not all, of the the blockages. Figure 4.2 illustrates such an example. In (a), the existing route is completely blocked for buffering, while in (b), the re-routed tree avoids all blockage, allowing buffers to be inserted. However, the most efficient solution is shown in (c) which avoids only some of the blockage.

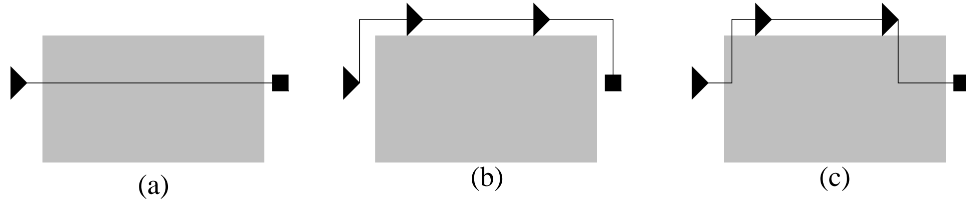


Figure 4.1: Example of how an alternative Steiner tree can enable buffer insertion. A tree routed exclusively over blockage (a) can be re-routed around the blockage (b), which enables buffer insertion. The newly created nets can then be globally routed over blockage (c).

The problem of buffer insertion in the presence of blockage constraints has been recently addressed in [27, 40]. The method in [40] optimizes the routing tree topology and inserts buffers simultaneously. While it obeys blockage constraints, the method makes no effort to construct routes that avoid blockage. Zhou *et al.* [27] present a clever approach which allows routing over some blockages while avoiding others. Their algorithm uses maze routing and dynamic programming techniques to find the buffered path with minimum delay (while obeying blockage constraints). However, the algorithm is limited to 2-pin nets.

In some design methodologies, it may be suitable to pre-allocate space for buffers during floorplanning, rather than trying to squeeze buffers between large blocks during physical design, which can cause both logical and wiring congestion. We call these pre-allocated regions *buffer bays*. For this methodology, the buffer insertion tool can view the entire layout area as blockage except for the buffer bays. Figure 4.3(a) shows an example of a two-pin net that does not cross any buffer bays and is thus totally blocked from buffer insertion. By re-routing the tree through a buffer bay (b), buffers can be suitably inserted (c).

This work makes the following contributions:

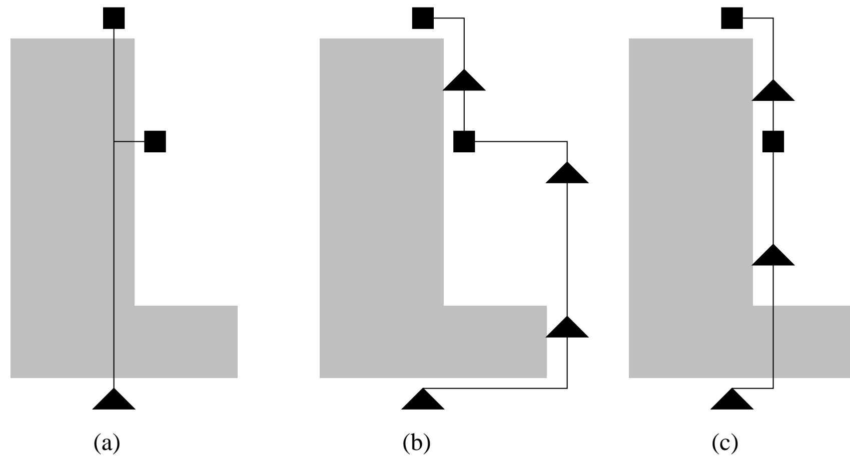


Figure 4.2: An example showing the benefit of not avoiding all blockage. A net routed totally over blockage (a) prevents any buffer insertion, while totally avoiding blockage (b) requires three buffers to be inserted. However, the best solution (c) avoids only some blockage which still permits two buffers to be inserted.

- We propose a general Steiner tree problem formulation for the application of buffer insertion with either blockage or bay constraints.

- We present a new Steiner tree construction that derives a heuristic solution to this problem. The algorithm iteratively rips up a sub-path of an existing Steiner tree and re-connects the two remaining sub-trees. Maze routing is used to achieve the lowest possible cost for this re-

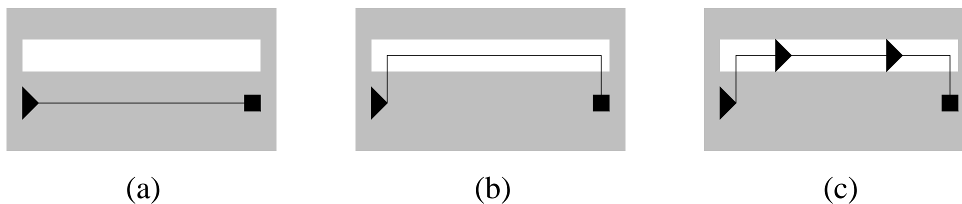


Figure 4.3: Buffer bay example. A tree which does not cross any buffer bays (a) can be re-routed through a buffer bay (b) which enables buffer insertion (c).

connection.

- Several speed-up techniques have been developed. These include the use of a customized grid graph that is appropriately sparsified, and the application of branch-and-bound methods that greatly improve the computational efficiency, without significantly altering the quality of the results.

- We show that for industry designs, our Steiner tree heuristic, when used with an effective buffer insertion algorithm, results in more useful solutions than a Steiner tree heuristic that does not account for blockages.

In contrast to the works of [25,27,40], which simultaneously insert buffers during routing, our methodology first constructs the Steiner tree, and then inserts buffers. The simultaneous approach is arguably superior considering that one cannot design the best tree until buffer locations are known. However, the simultaneous operations of tree construction and buffer insertion necessitate that the buffering component to be somewhat simplistic. The buffer insertion tool that we adopt has a wide user base and has several sophisticated features. It can

- handle a library of inverting and non-inverting buffers [39],

- simultaneously fix noise, slew and capacitance violations [33],

- run with higher-order gate and interconnect delay computations [37],

- trade off the number of buffers inserted with solution quality [39],

- simultaneously perform wire sizing,

- and insert buffers to conform with the net's hierarchical structure.

It is neither prudent nor necessarily feasible to integrate a simultaneous Steiner tree construction while maintaining both the features and performance of the tool as it currently exists.

4.2 Problem Formulation

Given a unique source v_0 and a set of sinks $V_{sink} = \{v_1, v_2, ..., v_p\}$, a *rectilinear Steiner tree* (RST) $T(V, E)$ is a spanning tree in the rectilinear plane that connects every node in $V = \{v_0\} \cup V_{sink} \cup V_{Steiner}$, where $V_{Steiner}$ is a set of additional Steiner nodes. The traditional definition of $V_{Steiner}$ includes two types of nodes: (i) *internal Steiner nodes* of degree three or four, denoted by the set $V_{internal}$, and (ii) *bend Steiner nodes* of degree two that denote a path switch between a horizontal and a vertical direction, denoted by the set V_{bend}. For the purposes of our discussion, we add a third type of node to $V_{Steiner}$: a node is a *boundary node*, belonging to the set V_{by}, if it has degree two, one incident edge lies over blockage, and the other incident lies in a blockage-free region. For example, the RST in Figure 4.4 shows a Steiner tree with source $V_0 = s$ and sinks $V_{sink} = \{d, i, k\}$. All the other nodes are in $V_{Steiner}$ with $b \in V_{internal}$, $g, j \in V_{bend}$, and $a, c, e, f, h \in V_{by}$.

Definition 4.1 (2-path) *A* 2-path *of a tree* $T(V, E)$ *is a path* $p(u, v) \in T$, $\{(u, v_1), (v_1, v_2), \ldots, (v_m, v)\}$ *such that* $\{v_1, \ldots, v_m\} \subseteq V_{by} \cup V_{bend}$ *and* $u, v \in \{v_0\} \cup V_{sink} \cup V_{internal}$.

Every tree T can be uniquely decomposed into a set of 2-paths. For example, the tree in Figure 4.4 can be decomposed into four 2-paths: $p(s, b), p(b, d), p(b, i)$ and $p(i, k)$.

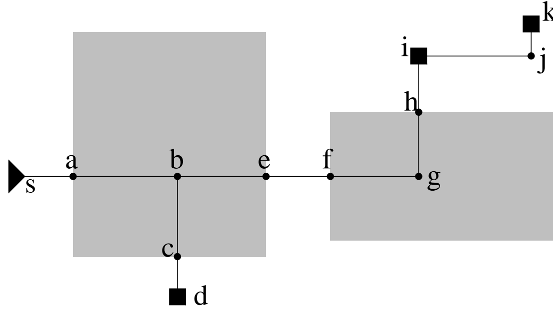


Figure 4.4: An example Steiner tree showing the different node types.

Unlike traditional maze routing [41,42], we permit the routes to intersect blockages, although it is preferable to avoid them. Hence, our cost function imposes a higher penalty on tree edges that intersect blockages.

A *rectangle* r has a unique *bounding box* $(x_1, y_1), (x_2, y_2)$, whereby $x_1 \leq x_2$ and $y_1 \leq y_2$. Given a set of rectangles B (i.e., the blockage map) , we define an edge $e \in E$ to be *inside* B (denoted by $e \in B$) if there exists a rectangle $r \in B$ such that both endpoints of e lie inside the bounding box of r. Let l_e denote the length of edge e. Our problem formulation is as follows:

Problem 4.1 (dual region rectilinear Steiner tree problem) *Given a parameter α, a source v_0, a set of sinks V_{sink}, and a set of rectangular blockages B, construct a Steiner tree $T(V, E)$ such that $\{v_0\} \cup V_{sink} \subseteq V$ and*

$$cost(T(V,E)) = \sum_{e \in E} l_e + \alpha \sum_{e \in B} l_e \tag{4.1}$$

is minimized.

The parameter α represents the degree of the penalty for routing over blockage. This problem is NP-Complete by the obvious reduction to the Rectilinear Minimal Steiner tree problem from setting $\alpha = 0$. Observe that we can also write $cost(T(V, E))$ as the sum of the costs of all 2-paths in T,

where the cost of a 2-path is given by:

$$cost(p(u,v)) = l_{p(u,v)} + \alpha \sum_{e \in B \cap p(u,v)} l_e \quad \text{where} \quad l_{p(u,v)} = \sum_{e \in p(u,v)} l_e \tag{4.2}$$

For example, if $\alpha = 1$, then edges that intersect blockage have twice the cost of the other edges. Recall the re-route in Figure 4.1. If the wire length more than doubles when changing the route from (a) to (b), then (a) is the lower cost solution. The appropriate value for α depends on the technology and the user requirements. Doubling the wire length might have allowed buffering, but might not have reduced delay due to the additional interconnect resistance and capacitance introduced by the longer route.

An advantage of this cost function is that it can be used to handle buffer bays just as easily as blockages. If B is instead a set of buffer bays, then routing over rectangles in B should actually reduce the cost function. If one chooses α to lie between -1 and 0, this effect is achieved. For example, if $\alpha = -\frac{1}{2}$, then the cost of routing over blockage is twice that of routing inside a buffer bay.

Of course, Equation (4.1) is only one possible objective function. One could also incorporate, e.g., the maximum length over all sub-paths that intersect blockage, the sum of the squared lengths of these sub-paths (scaled), or actual path delays into the objective. More sophisticated objectives may be better suited for buffer insertion, but more difficult to incorporate into an optimization.

4.3 The Grid Graph Construction

Our Steiner tree heuristic is based on maze routing, which has an appealing flexibility for handling multiple cost functions. Maze routing approaches have

been used elsewhere in recent research, e.g., [27, 43, 44]. Maze routing can be inefficient as it is liable to perform its search over numerous locations, a large number of which do not lead to worthwhile solutions.

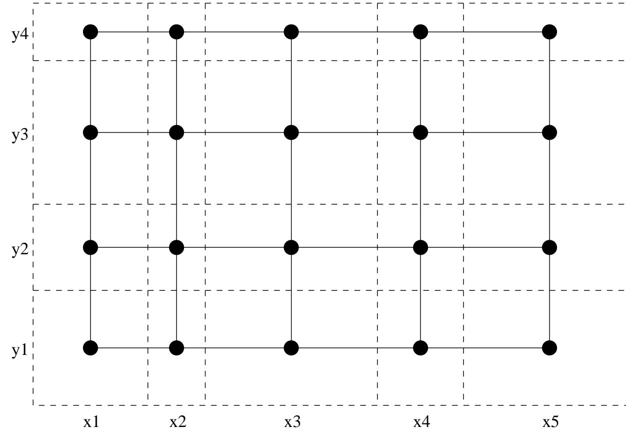


Figure 4.5: An example of the grid graph induced from five x and four y coordinates. Edges of the grid graph are solid lines while the dashed lines show the boundaries for the corresponding tiles.

A fundamental notion in maze routing is the concept of a *grid graph*, $G(V_G, E_G)$. A grid graph can be viewed as a tessellation of rectangular tiles with V_G being the set of tile centers and E_G being edges that connect tile centers, as illustrated in Figure 4.5. A grid graph can be uniquely induced by the sets $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$ of sorted non-duplicate coordinates. The *induced grid graph* $G(V_G, E_G)$ from X and Y has vertices $V_G = \{(x, y) \mid x \in X, y \in Y\}$, and edges $E_G = \{((x_i, y), (x_{i+1}, y)) \mid 1 \leq i < |X|, y \in Y\} \cup \{((x, y_i), (x, y_{i+1})) \mid 1 \leq i < |Y|, x \in X\}$.

Most maze routing algorithms utilize a uniform grid graph, which forces a routing algorithm to spend an equal amount of time searching each part of the routing area. This can be wasteful due to the non-uniform distributions of sinks and blockages. Uniformity also adds the dilemma of choosing the appropriate tile size: too refined a tiling causes excessive computation time,

while too coarse a tiling can overlook good solutions. Our proposed grid graph is non-uniform, allowing high density channels in difficult routing areas and low density channels elsewhere.

We assume that some low-cost RC tree T has already been computed over $\{v_0\} \cup V_{sink}$. In particular, our methodology begins with such a tree constructed by a wire length-based heuristic and modifies it to become suitable for buffer insertion.

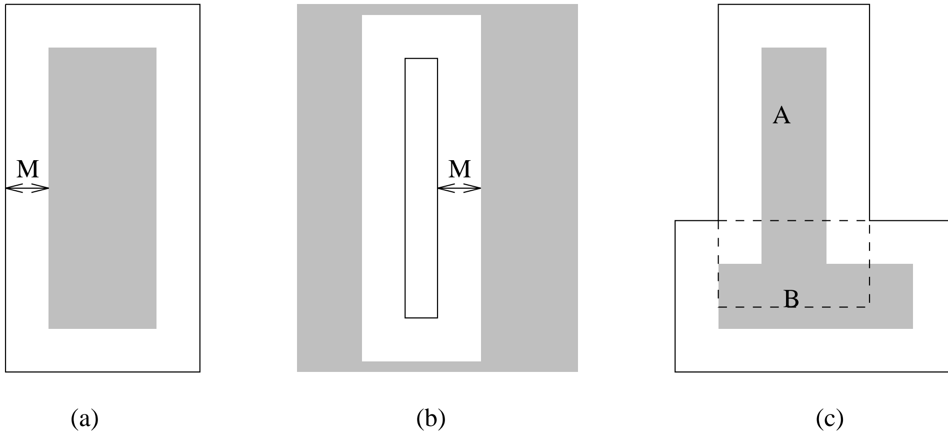


Figure 4.6: Examples of using M to compute usable tracks (a) around blockages, (b) inside buffer bays and (c) with overlapping blockages. Dashed lines indicate tracks that are infeasible for buffer insertion.

Our grid graph is a superset of the *Hanan grid* [4] for T. We require that no buffer can be placed within a distance of less than M units from a blockage, where the value of M is half the width (or height if greater than width) of the largest buffer. Therefore, we add routes that surround each blockage by this prescribed offset parameter M. Similarly for buffer bays, the offsets are added internally to each region, allowing sufficient room for buffers. See Figure 4.6 for examples.

We construct a grid graph according to the procedure shown in Figure 4.7. Step 1 initializes sets X and Y to be empty, and Step 2 adds the coordinates

Procedure Grid_graph(T, B)
Input: Steiner tree $T(V, E)$, a set of rectangles B
Output: Grid graph $G(V_G, E_G)$
1. Set $X = \emptyset, Y = \emptyset$.
2. For each $v \in V$ with coordinates (x, y),
$X \leftarrow x \cup X$, $Y \leftarrow y \cup Y$.
3. For each rectangle $r \in B$ with bounding box $(x_1, y_1), (x_2, y_2)$
If r is a blockage
$X \leftarrow (x_1 - M) \cup (x_2 + M) \cup X$,
$Y \leftarrow (y_1 - M) \cup (y_2 + M) \cup Y$.
If r is a buffer bay
$X \leftarrow (x_1 + M) \cup (x_2 - M) \cup X$,
$Y \leftarrow (y_1 + M) \cup (y_2 - M) \cup Y$.
4. Sort the coordinates in X and Y
5. Generate the induced grid graph $G(V_G, E_G)$ from X and Y.
6. For each edge $e \in E_G$
Compute the value for the the *blocked*(e) property.

Figure 4.7: The Grid_graph procedure.

of each tree node into X and Y. Step 3 adds the coordinates of the blockages, and Steps 4-5 construct the grid graph induced by X and Y. Finally, Step 6 sets the attribute *blocked*(e) for each edge e in G. If e overlaps with a blockage in B or does not overlap with a buffer bay in B, then the attribute is set to $true$; otherwise, it is set to $false$. We refer to this grid graph as the Extended Hanan Grid (EHG). An example grid graph constructed from a 3-pin net and a single blockage is shown in Figure 4.8.

Since the EHG may be very sparse in some regions, a natural question to ask is whether any loss in optimality is incurred by considering only tracks on the EHG and neglecting the large spaces off the EHG. This question can

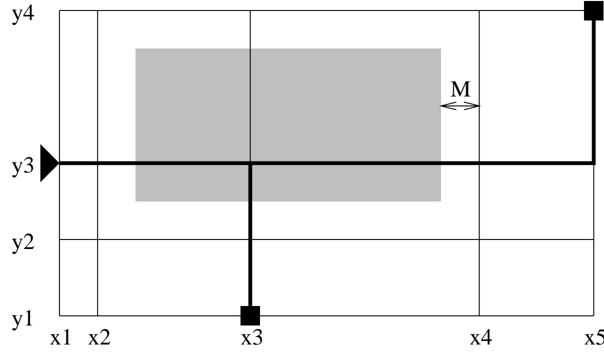


Figure 4.8: The grid graph for an example 3-pin net and a single rectangular blockage.

be answered by the following theorem.

Theorem 4.1 *Given two disjoint subtrees T_s and T_t embedded on the EHG, there exists a minimum cost 2-path $p(u_0, u_m)$ such that $u_0 \in T_s$, $u_m \in T_t$ and each edge in $p(u_0, u_m)$ lies on the EHG.*

Proof: We begin by emphasizing that the cost function here is not the wire length, but the function in Equation (4.2).

Suppose $P = p_{min}(u_0, u_m) = \{(u_0, u_1), (u_1, u_2), ..., (u_{m-1}, u_m)\}$ is a minimum cost 2-path with $u_0 \in T_s$ and $u_m \in T_t$. For $0 < j \leq m$, let e_j represent edge $(u_{j-1}, u_j) \in P$. For any edge e_j not on EHG, we will show that e_j can be moved onto EHG without affecting $cost(P)$.

For $1 < j < m$, the configuration of the three contiguous edges e_{j-1}, e_j and e_{j+1} may either be U-shaped or Z-shaped as shown by the solid lines in Figure 4.9 (a)-(d). The dashed lines in Figure 4.9 correspond to the EHG, and g_1 and g_2 are two closest edges to e_j on the EHG, that are parallel to e_j.

For any U-shaped sub-path, there always exists an EHG edge such that if e_j is moved toward it, the 2-path cost will be reduced; in Figure 4.9(a), this

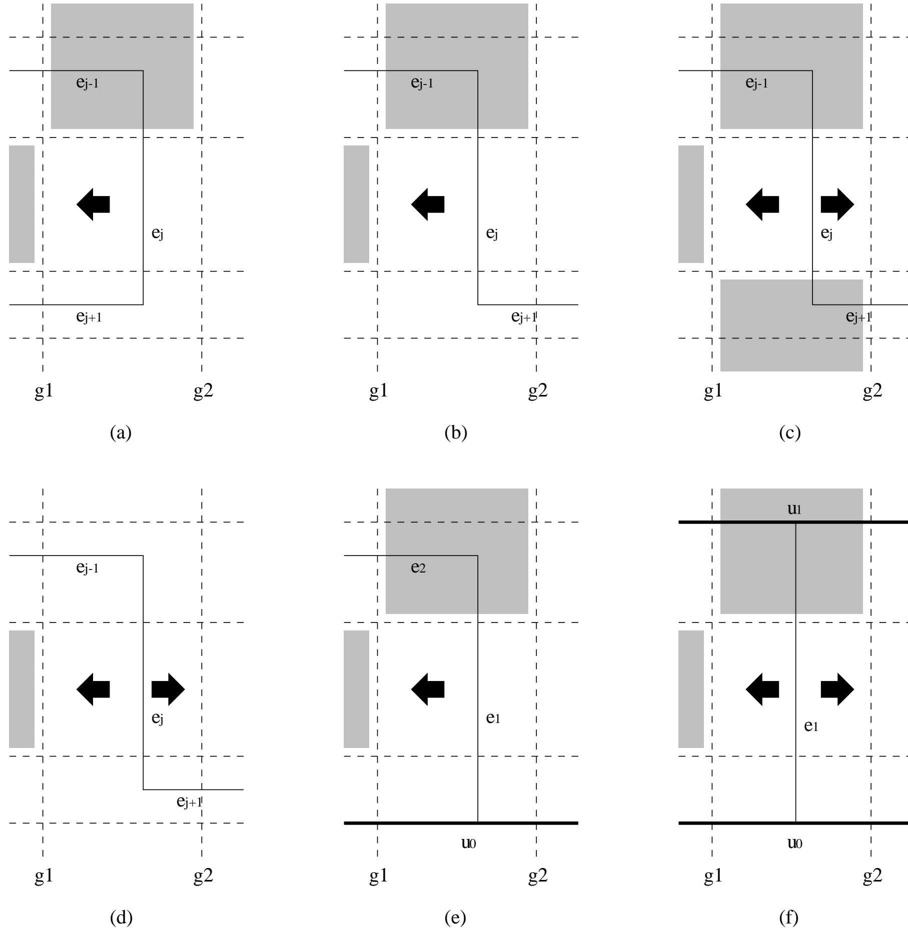


Figure 4.9: Configurations for edges in P in the proof of Theorem 4.1.

edge overlaps with g_1. A reduction in path cost contradicts the assumption that P is a minimum cost path. Hence, e_{j-1}, e_j and e_{j+1} cannot be in a U-shaped configuration unless e_j already lies on the EHG.

For a Z-shaped configuration, if one end point of e_j is within a buffer blockage and the other is not, as in Figure 4.9(b), the 2-path cost can also be reduced by moving e_j towards a neighboring EHG edge. Therefore, if e_j does not lie on the EHG, any corresponding Z-shape must have either both ends overlapping blockage or both ends outside blockage. These cases are shown in Figure 4.9(c-d). For these cases, e_j may be moved onto either g_1 or

g_2 without affecting the 2-path cost.

If e_1 does not lie on the EHG, the possible configurations for e_1 are shown in Figure 4.9(e-f), with T_s and T_t shown in bold. Similarly to the previous analysis, e_1 can be moved to an EHG edge in Figure 4.9(e) to reduce the cost, which makes this configuration impossible. In Figure 4.9(f), e_1 can be moved onto an EHG ege without increasing the cost. The exact analysis can also be applied to e_m.

In conclusion, edges in P can be moved so that each edge in P lies on the EHG and so that the cost of P does not increase. □

Since the initial routing tree is assumed to be on the EHG, according to the above observation, it is reasonable to restrict the solution search to the EHG.

4.4 Algorithm Description

4.4.1 Overview

Our algorithm first decomposes the existing Steiner tree into disjoint 2-paths, then computes the cost for each 2-path. The algorithm then iteratively chooses a poorly routed 2-path, removes it, and then re-routes it. The 2-path with highest cost is not necessarily the most poorly routed path, as the highest cost path may simply be a very long path completely out of any blockage. We choose the 2-path $p(u,v)$ with the highest value of $cost(p(u,v))/l_{p(u,v)}$ to re-route. This 2-path has the highest ratio of wire length routed over blockage to total wire length. When this ratio is high, it is likely that the route of the 2-path can be improved significantly. The algorithm proceeds iteratively by ripping up and re-routing 2-paths in this manner.

Steiner_Tree Algorithm (T, B)	
Input:	$T(V, E)$, a Steiner routing tree
	A set of rectangles B representing buffer blockages or bays
Output:	Re-routed Steiner tree T
1. $G(V_G, E_G) = \text{Grid_graph}(T, B)$ (see Figure 4.7).	
2. Compute the set P of disjoint 2-paths in T.	
Compute the cost of each 2-path in P from Equation (4.2).	
3. While $P \neq \emptyset$	
4. Choose $p(u, v) \in P$ such that $cost(p(u, v))/l_{p(u,v)}$ is maximized.	
5. Remove $p(u, v)$ from T and P, thereby creating two sub-trees.	
Label corresponding sub-tree embedded in E_G that contains	
v_0 as T_s and the other as T_t.	
6. Find 2-path $p(q, w) = \text{Maze_routing}(G, T_s, T_t)$.	
7. Add the edges in the 2-path $p(q, w)$ to T.	

Figure 4.10: The Steiner tree construction algorithm.

A complete description of the algorithm is given in Figure 4.10. Step 1 computes the underlying grid graph for T and B. Step 2 finds the set of all 2-paths, and Steps 3 and 4 iterate through these 2-paths, each time picking the one with the highest overlap cost. The 2-path with highest cost length ratio is removed in step 5, which induces two subtrees T_s and T_t. Step 6 performs the maze routing which returns a minimum cost 2-path between T_s and T_t, and Step 7 re-connects the tree using this 2-path. We now explain how the minimum cost 2-path between subtrees is computed in Step 6.

4.4.2 Maze Routing

The minimum cost path joining two subtrees is found by maze routing. The original maze routing algorithm [42] is designed for point-to-point con-

Maze_routing(G, T_s, T_t) **Algorithm**
Input: Underlying grid graph $G(V_G, E_G)$. Two disjoint Steiner trees T_s and T_t embedded in G. **Output:** 2-path $p(q, w)$ with $q \in T_s, w \in T_t$
1. For each grid node $v \in V_G$, set $label(v) = \infty$, $visited(v) = false$, and $parent(v) = \emptyset$. 2. For each node $v \in T_s$ Set $label(v) = 0$ and set $Q = Q \cup \{v\}$. 3. While $Q \neq \emptyset$ 4. Let $v \in Q$ be the grid node with minimum $label(v)$. Delete v from Q. Set $visited(v) = true$. 5. For each node u, such that $(u, v) \in E_G$ and $u \neq parent(v)$ $newLabel = label(v) + l_{(u,v)}$. If $blocked(u, v)$ then $newLabel = newLabel + \alpha l_{(u,v)}$ 6. If $newLabel < label(u)$ then $label(u) = newLabel$, Set $parent(u) = v$. 7. If $visited(u) = false$ and $u \notin T_t$, insert u into Q. 8. Find node $w \in T_t$ such that $label(w)$ is minimum. 9. Find the path $p(q, w)$ from w to a node $q \in T_s$ by tracing back $parent$. Return $p(q, w)$.

Figure 4.11: Algorithm for maze routing connecting two subtrees.

nections. It runs on a grid graph, and grid edges are the only legal candidates for any routing path. Each grid edge is assigned a cost, commonly the edge length (and blocked edges have infinite cost). Maze routing is equivalent to Dijkstra's shortest path algorithm [45] applied on the grid graph. The source node is initially assigned zero cost, and then wave expansion proceeds out from the source, labeling all intermediate nodes until the target node is reached. The grid node labels reflect the cost from the source. For a linear cost function, maze routing guarantees the least cost path for connecting two

points. One primary difference between our algorithm and traditional maze routing is that all nodes in the source tree are assigned zero cost and that the target node is any node in the target tree.

The complete procedure is shown in Figure 4.11. Step 1 initializes three arrays, *label*, *visited*, and *parent* for each node in the grid graph. The $label(v)$ value is the cost of the best path from a node in T_s to v, the $visited(v)$ value indicates whether v has been explored in the maze routing search, and $parent(v)$ is used to store the best path to v. Step 2 initializes the labels of all nodes in T_s to zero and puts them into a node set Q. Implementing Q as a priority queue gives the most efficient runtimes.

Steps 3-7 search the grid graph by iteratively deleting the node v with smallest label from Q and exploring that node. Each neighbor node u of v is explored in Steps 5-6, and the label for u is updated according to length of edge (u, v) and whether edge (u, v) is blocked. If the new label, corresponding to a path to u through v, is less than the previous label for u, the label is updated and v becomes the parent for u. Steps 8-9 find the node with the smallest label in the target tree, and uncover the path back to the source tree by following the parent array. This path is then returned to the calling procedure.

4.4.3 Complexity Analysis

Given a tree $T(V, E)$ and a set of blockages B, let $n = |V|$ and $k = |B|$. The number of nodes and edges in the grid graph constructed is $O((n + k)^2)$. If one implements Q as a priority queue in Figure 4.11, then procedure Maze_routing has complexity $O((n + k)^2 \log(n + k))$. The number of times this procedure is called by the Steiner_Tree procedure is the same as the

number of 2-paths in T, which is bounded by $O(n)$. The complexity for the entire algorithm is thus $O(n(n+k)^2 \log(n+k))$. We present speedups in Section 4.5 that are able to achieve fast CPU times despite the high worst case complexity.

4.5 Improving Efficiency

The high time complexity of the algorithm suggests that one can speed up the algorithm significantly without necessarily sacrificing solution quality. We have incorporated two techniques, a sparsified grid graph construction and branch-and-bound maze routing, that together improve runtimes by more than a factor of ten.

4.5.1 Sparsified Grid Graph

When $|B|$ is large, the induced grid graph can be very dense. Blockages that do not perfectly line up can cause several edges in the grid graph to be extremely close together. A routing tree construction could choose any of these edges and result in essentially the same tree. A *track* is a set of edges all with the same x or y coordinate. Given a step size, such as 0.1 mm, two parallel tracks are called *redundant* if they are closer than the step size and if at least one of them does not intersect a net pin (source or sink). Given two redundant tracks a and b, if track a intersects a net pin while b does not, then track b is removed. If neither a nor b intersects a net pin, then one track is arbitrarily chosen for removal.

Figure 4.12 shows an example of a grid graph (a) before and (b) after sparsification. The pairs of tracks given by coordinates x_1 and x_2 and by y_3

and y_4 are redundant. Since x_1 intersects the source, x_2 is removed. Neither y_3 or y_4 intersect a net pin, so y_4 is randomly removed.

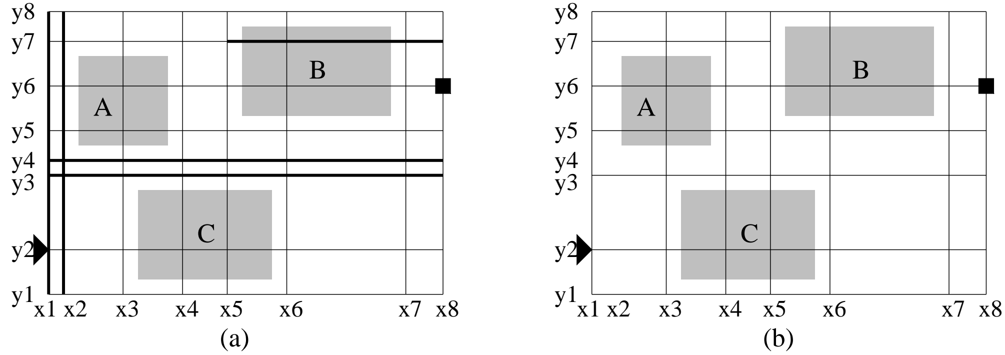


Figure 4.12: The original grid graph (a) has (shown in bold) two pairs of redundant tracks and a severable track. The sparsified grid graph (b) has no redundant nor severable tracks.

The second sparsification technique refrains from having some tracks span the entire grid graph. For example, in Figure 4.12(a), the track y_7 is induced by the upper border of the rectangle representing blockage A; thus, a routing path that uses track y_7 results from avoiding blockage A. When the path hits B it can either overlap or circumvent B. If the routing cost according to Equation (4.2) of circumventing B is less than the cost of overlapping B, then we say the corresponding track is *severable*. The bold part of track y_7 (a) that firsts hits the blockage B can be removed (b).

From Figure 4.12, the application of the above sparsification techniques reduces the number of grid nodes from (a) 64 to (b) 46.

4.5.2 Branch and Bound Maze Routing

When expanding the grid node with the smallest cost label in the wavefront, maze routing cannot distinguish between good and bad global directions.

The expansion may proceed in a direction completely opposite the target sub-tree. Much computation time can be wasted exploring regions of the grid graph that are nowhere near the target.

Branch and bound techniques can prevent some unnecessary expansions. Recall Steps 3-7 of Figure 4.10 which iteratively delete and then reconstruct 2-paths. The 2-path $p(u, v)$ removed in Step 4 has $cost(p(u, v))$ which is also an upper bound for the cost of the new 2-path (since the cost should never increase). Let $upCost$ denote this value.

After Step 4 of Figure 4.11, one can compare $label(v)$ to $upCost$ to determine if node v is worth expanding. If $label(v) > upCost$ then the cost of the path from T_s to v is already higher than the cost of the original 2-path, which makes it wasteful to expand v. Whenever a grid node v in the target subtree T_t is reached by the wave expansion, the value for $upCost$ can be replaced by $label(v)$ if this value is less than $upCost$.

The bound can be made even tighter by using a lower bound on the cost of the remaining routing to be done from v to T_t. Let $dist(v, T_t)$ be the Manhattan distance from v to the bounding box of T_t (which can be computed in constant time).[1] Now the test becomes whether $label(v) + dist(v, T_t) > upCost$ holds. If so, node v is not worth further exploration and and Step 7 of Figure 4.11 is skipped.

An example of this branch and bound scheme is illustrated in Figure 4.13, where α is set to 2. In (a), the original 2-path $p(u, w)$ is removed and $upCost$ is set to 18. For node v in Figure 4.13(b), the sum of its cost label (10) and the estimated lower bound on the cost of its connection to the target subtree (12) is 22, which is greater than $upCost$, which makes further exploration of v

[1] If $\alpha < 0$, then $(1 + \alpha)dist(v, T_t)$ is the lower bound.

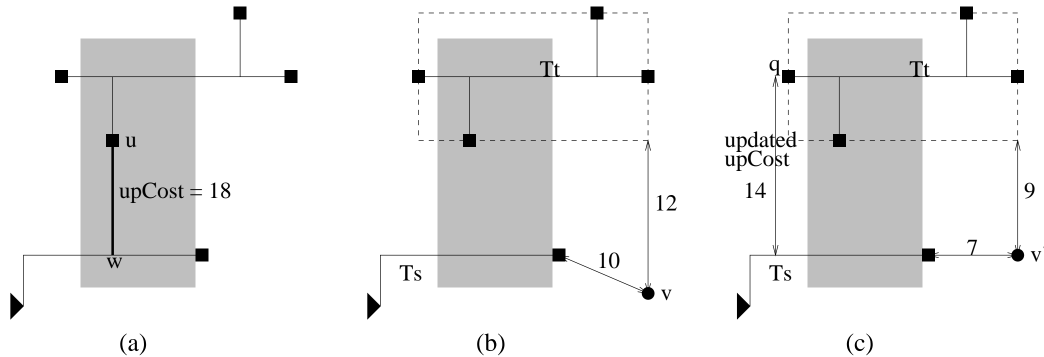


Figure 4.13: An example of branch-and-bound in maze routing.

unnecessary. A different scenario is illustrated in Figure 4.13(c). When node q in the target sub-tree is reached by the maze router, the value of *upCost* is updated to 14, the cost label of q. If node q is reached prior to node v', then v' will not be expanded, because the sum of its label (7) and estimated lower cost bound to target subtree (9) is greater than the new value for *upCost*.

4.6 Experiments

We performed experiments on the following three designs: (1) a small hand crafted test, (2) a large macro block, and (3) a hierarchical microprocessor design. The comparisons that follow are made between two algorithms, SMT, a Steiner minimal tree algorithm that is used for net analysis within an industrial physical design tool suite, and BBB, our proposed algorithm. All run times are reported in seconds for an IBM RS6000/595 processor with 512MB of RAM.

4.6.1 Additional Routing Cost

Our first set of experiments measures the additional wire length caused by BBB compared to SMT. Since BBB is aware of blockage constraints while

SMT is not, BBB should naturally increase the total wire length, but decrease the wire length running through blockages. Tables 4.1 and 4.2 present these results for the hand crafted and large macro block test cases respectively. All the CPU times given the experimental results are in seconds.

				Avg. wire length			Avg. blocked wire length			
mode	#net	#pins	#rects	SMT	BBB	%imprv.	SMT	BBB	%imprv.	Ave CPU
blckg	23	2-8	2-7	21.3	21.7	-1.8%	15.9	5.8	63.5%	0.2
bays	30	2-8	4-7	22.7	22.2	-2.2%	22.2	10.4	52.7%	0.1

Table 4.1: Summary of additional routing costs of SMT versus BBB for the hand crafted test case.

For the hand crafted test case, SMT and BBB were on 23 nets with 7 random blockages inserted. Both algorithms were also run on 30 different nets with 7 random buffer bays inserted. The #rects column indicates the number of blockages actually used by the BBB grid graph. The results were averaged over all nets and are summarized in Table 4.1. Observe that the average wire length increased by only 1.8% for blockages and by only 2.2% for buffer bays. This result indicates that BBB is almost as good as SMT for computing a low wire length Steiner tree. However, the total wire length in blocked regions was reduced by 63.5% for blockages and 52.7% for buffer bays by BBB.

For the macro block, we chose 16 high capacitance nets that had differentiating characteristics (number of pins, pin locations, wire length topology, etc.) and ran both SMT and BBB with the 54 blockages that were present in the design. The results for each net is presented in Table 4.2. Overall, we observe similar behavior to the hand crafted case, namely, a 2.5% increase in total wire length and a 33.3% reduction in blocked wire length. The results

net	#pins	#rects	Wire length			Blocked wire length			CPU
			SMT	BBB	%imprv.	SMT	BBB	%imprv.	
N1	2	26	10.7	12.2	-13.9%	9.3	2.0	78.6%	0.5
N2	2	36	9.0	9.0	0.0%	5.2	0.4	92.9%	0.8
N3	9	30	14.6	15.1	-3.8%	12.7	4.9	61.4%	1.3
N4	9	31	14.6	15.2	-4.6%	12.8	7.1	44.4%	1.3
N5	9	47	18.4	18.7	-1.7%	18.2	14.0	23.2%	2.2
N6	11	47	17.1	17.6	-2.8%	17.1	2.6	84.9%	2.7
N7	17	53	24.1	24.1	-0.1%	22.4	21.9	2.3%	5.8
N8	19	47	19.7	20.7	-5.0%	19.7	16.6	16.0%	5.2
N9	19	47	20.2	20.8	-3.2%	20.2	17.7	12.3%	5.6
N10	25	47	22.2	22.3	-0.3%	22.0	20.9	4.9%	4.7
N11	25	47	22.6	22.7	-0.4%	22.4	21.3	4.9%	4.8
N12	25	47	23.6	24.1	-2.1%	23.5	14.6	37.8%	5.9
N13	29	33	23.3	23.9	-2.8%	15.7	10.9	30.3%	5.4
N14	29	33	24.9	25.1	-0.6%	18.4	14.2	22.6%	4.9
N15	29	53	30.5	31.4	-3.0%	23.3	11.2	51.8%	9.8
N16	29	53	29.0	30.4	-5.0%	19.9	8.6	56.7%	14.0
Ave.	18	42	20.3	20.8	-2.5%	17.7	11.8	33.3%	4.7

Table 4.2: Summary of additional routing costs of SMT versus BBB for the macro block test case.

vary widely for different nets. For example, the reductions on in-blockage length for net7, net10 and net11 are very limited because the majority of the pins themselves actually lie within blockage. Despite the inclusion of these difficult cases, BBB achieves substantial reduction of wire length contained in blockages with a trivial additional wire length penalty.

4.6.2 Delay Comparisons with Buffer Insertion

To assess the utility of BBB versus SMT trees, buffer insertion must be performed after routing. The next set of experiments were performed on a net by net basis with the following methodology:

1. Compute the SMT tree for the net.
2. Compute the delays to each sink, then compute the slack to the most critical sink based on the required arrival times supplied by the static timing analyzer.
3. Run BBB re-routing.
4. Perform buffer insertion using the algorithm discussed in Section 1. This algorithm attempts to find the best possible solution for each possible number of buffers.
5. Re-compute the slack to the most critical sink. Let $\Delta slack$ denote the difference between this slack and the slack computed in Step 2.

Skipping Step 3 of this methodology yields buffer insertion with the SMT algorithm while including Step 3 yields results for the BBB algorithm.

Average results for the hand-crafted test case are presented in Table 4.3. The values for $\Delta slack$ are presented in picoseconds. The quoted $\Delta slack$ values are for the best buffer insertion solutions generated for both the SMT and BBB algorithms. Observe that BBB utilized more buffers than SMT (2.9 versus 2.2 for blockage and 2.3 versus 1.9 for bays) since the BBB routing offered more candidate buffer locations that did not overlap with blockage. BBB trees also resulted in an additional 337 (768) ps of slack increase versus SMT trees for blockage (bay) mode.

We also ran the same experiments for the 16 nets chosen from the macro block test case. The results for each net are presented in Table 4.4. For the net by net comparisons, we first took the SMT solution which yielded the best value for $\Delta slack$, then compared it against the BBB solution with the same number of buffers as the SMT solution. Thus, each row in Table

mode	#net	SMT + Buffering		BBB + Buffering		
		Ave. $\Delta slack$	Ave. #bufs	Ave. $\Delta slack$	Ave. #bufs	Ave. CPU
blockage	23	2064	2.2	2401	2.9	4.0
bays	30	2494	1.9	3262	2.3	4.3

Table 4.3: Experimental results on average slack improvements for the hand crafted test case.

4.4 uses the same number of inserted buffers. The runtimes reported are for the combination of BBB plus the buffer insertion step. By comparing these runtimes to those reported for BBB alone in Table 4.2, we see that the runtimes of BBB do not dominate the buffer insertion runtimes.

Observe from Table 4.4 that SMT trees resulted in an average slack improvement of 519.4 ps for the critical path as compared to 694.6 ps for BBB. This 175.2 ps difference would actually increase by an additional 16 ps if the $\Delta slack$ values were chosen by the best number of buffers inserted for BBB trees as opposed to SMT trees. However, the average number of buffers would also increase from 2.6 to 3.4.

4.6.3 Fixing Slew Problems

Finally, we considered the problem of using buffers not to necessarily reduce delay to the most critical sink, but to fix slew problems. In high performance design, it is common for each gate to have a requirement for the maximum permissible slew rate on the input signal to the gate. Too slow a slew rate will cause a long gate delay and poor circuit performance. A signal with a sharp slew rate at the driver will degrade significantly while traversing a long interconnect to cause a slew violation at the input to the sink. Buffers can be

net	#pins	SMT $\Delta slack$	BBB $\Delta slack$	#bufs	BBB CPU
N1	2	1032	1118	2	1.2
N2	2	1034	1036	1	1.2
N3	9	109	239	2	2.1
N4	9	109	236	2	2.2
N5	9	190	452	1	2.9
N6	11	7	71	1	4.0
N7	17	850	1181	2	7.8
N8	19	578	1089	2	7.3
N9	19	605	880	2	7.8
N10	25	277	299	2	7.6
N11	25	295	323	2	7.4
N12	25	205	228	2	9.0
N13	29	223	308	5	24.4
N14	29	371	375	4	25.2
N15	29	1049	1605	7	35.3
N16	29	1376	1674	5	36.6
Ave.	18	519.4	694.6	2.6	11.4

Table 4.4: Experimental results on slack improvement for the macro block.

used to fix such problems by repowering a degrading signal and sharpening the slew rate.

For the microprocessor test case, designers identified 29 non-critical nets that had slew violations. We attempted to fix these violations using the routes provided by both SMT and BBB in conjunction with buffer insertion. The designers also identified several buffer bays for placing buffers; the remaining regions were considered completely blocked.

Of the 29 nets, 5 of them had pins nowhere near the designated buffer bays

Algorithm	#nets	#nets fixed	#nets improved	#nets failed
SMT	24	7	6	11
BBB	24	17	4	3

Table 4.5: Slew results for SMT and BBB on the microprocessor test case.

and so neither the SMT nor the BBB approach could fix the slew violations. The results for the remaining 24 nets are shown in Table 5. Of the remaining 24 nets, BBB was able to successfully re-route and fix 17 of the nets while SMT was only able to fix 7 nets. Of the 7 nets for which BBB failed, BBB was able to improve the slew (but not quite fix it) for 4 nets, while it did not insert any buffers for 3 of the nets. SMT could not insert buffers on 11 nets since they did not intersect buffer bays, but it was able to improve, but not fix, the slew on 6 of the nets.

Overall, BBB showed that it is better suited for fixing slew violations than a routing algorithm that ignores blockage. A less stringent sprinkling of buffer bays in the design would no doubt have helped BBB succeed in fixing more of the slew violations.

4.7 Conclusion

We have proposed a new Steiner tree routing heuristic for making nets more amenable to buffer insertion in the presence of blockage constraints. The problem is formulated to handle either buffer blockages or buffer bay floor-planning methodology. Our heuristic iteratively deletes and re-routes sub-paths of an existing Steiner tree and can handle complex buffer blockage and bay distributions. Several speedup techniques have been incorporated so that

the empirical run times are practical, even though the theoretical time complexity of the algorithm is high. Experimental results show that our method achieves the objective of avoiding buffer blockages (seeking buffer bays) and can provide significant improvements in terms of delay and slew when used in conjunction with an industrial buffer insertion tool.

Chapter 5

Performance Driven Multi-net Global Routing

5.1 Introduction

Global routing is an important stage in VLSI physical design, in which a given set of global nets is routed coarsely, in an area that is conceptually divided into small regions called routing cells. For each net, a routing tree is specified only in terms of the cells through which it passes. In this chapter, we propose new approaches to enhance the quality of global routing.

For a boundary between two neighboring cells, the number of available routing tracks across it, called supply, is limited. One fundamental goal of global routing is to minimize the congestion so that the number of nets across each boundary does not exceed its supply, i.e., no overflow occurs. Since minimizing congestion is very hard to achieve and is essential for global routing, it has long been a focus of research [18,46–56] in global routing. Most of these works belong to one or a combination of the following genres: the sequential

approach, hierarchical methods, linear programming or multicommodity flow based algorithms, and rip-up-and-reroute techniques.

In the sequential approach, the nets are routed one after another. In [46], for each net, a minimum weighted Steiner tree spanning the grid graph is sought to minimize the congestion, with the weights being proportional to the density of wires in each routing cell. The sequential approach requires the nets to be routed in some order, on which the quality of the solution depends. As a solution to avoid this ordering problem, the hierarchical method [47–49] recursively splits the routing region into successively smaller parts. At each hierarchical level, all of the nets are routed simultaneously (often through linear programming) and refined in the next hierarchical level until the lowest level of the hierarchy is reached. Sometimes the whole global routing is formulated and solved through linear programming followed by a randomized rounding [50]. Another method is the application of multicommodity flow model [51–53], in which the fractional solutions are rounded to obtain the routing solutions. For global routing on standard cell designs, the work of [18] proposed an iterative deletion technique to avoid the net ordering problem. The works of [54–56] first route each net independently, then rip up the wires in congested areas and reroute them to spread out the routing density. The rip-up-and-reroute technique is very practical and popular in industrial applications.

When interconnect becomes a performance bottleneck in deep submicron technology, merely minimizing congestion is not enough. In later works [57–59], interconnect delays are explicitly considered during global routing. In [57], each net is initially routed in SERT-C [6], after which the congested area is ripped up and rerouted by applying a multicommodity flow algorithm locally. In [58], the delay issues are considered more strictly. Beginning with

a set of routing trees satisfying timing constraints for each net, a multicommodity flow method is applied to choose a single routing tree for each net, such that the congestion is minimized. At places where overflow occurs, the wires are ripped up and rerouted through a maze routing procedure in which the timing objective is combined with wirelength and congestion. Both works combine the application of rip-up-and-reroute and the multicommodity flow.

In global routing, congestion and delay are often competing objectives. In order to avoid congestion, some wires must make detours, and the signal delay may consequently suffer. We propose a new approach to global routing such that both congestion and timing objectives can be optimized at the same time. One key observation is that there are several routing topology flexibilities that can be traded into congestion reduction while ensuring that timing constraints are satisfied. These flexibilities are expressed through the concepts of a soft edge and a slideable Steiner node. Our strategy is to obtain desired timing performance for each net independently and then trade the flexibilities into congestion reduction without hurting delay.

Based on this strategy, we developed two global routing algorithms. In the first algorithm, the crucial part is routing through hierarchical bisection and assignment as in [60, 61]. However, due to interdependence on timing slack consumption and the presence of slideable Steiner nodes, the assignment is not straightforward as in [60, 61]. We construct a network flow model so that the timing slack consumptions are adaptive to the congestion distributions. Finally, a timing-constrained rip-up-and-reroute process is performed to overcome any inabilities of the hierarchical approach in satisfying congestion constraints. Since the timing performance of initial routing solution can be preserved, our methods provides a general framework that can accommodate any single-net routing scheme and can be applied on any delay

model.

Besides congestion and timing, the number of bends for each wire needs to be limited. A wire bend usually implies a switching of layers, which involves a via resistance that adds to the delay and reduce reliability. Moreover, vias will consume more wiring space because of their larger pitch requirement. Particularly for MCM and PCB designs, the via resource is stringent and the number of bends on wires need to be optimized. In work of [49], a hierarchical global routing algorithm is proposed to control the number of vias for each wire. There are many other works in routing to minimize the number of vias, such as V4R [62]. The second algorithm we propose handles this objective together with timing and congestion through a probability-based gradual refinement approach.

5.2 Congestion Metric and Problem Formulation

As in conventional global routing, we tessellate the entire routing region for a set of nets $\mathcal{N}$ into an array of uniform rectangular cells. We represent this tessellation as a graph called the grid graph $G(V_G, E_G)$, where $V_G = \{g_1, g_2, ...\}$ corresponds to the set of grid cells, and a grid edge $b_{ij} = (g_i, g_j) \in E_G$ corresponds to the boundary between two adjacent grid cells $g_i, g_j \in V_G$. There are a limited number of routing tracks across any grid edge, b, called the *supply* of the grid edge and expressed as $s(b)$. During the routing, the number of tracks occupied by wires across a grid edge b is designated as the

demand, $d(b)$. The *overflow* $f_{ov}(b)$ at grid edge b is defined by:

$$f_{ov}(b) = \begin{cases} d(b) - s(b), & d(b) > s(b) \\ 0, & \text{otherwise} \end{cases} \tag{5.1}$$

The *demand density* for a grid edge b is defined as $\mathcal{D}(b) = \frac{d(b)}{s(b)}$. We use the maximum demand density $\mathcal{D}_{max} = \max_{b \in E_G}\{\mathcal{D}(b)\}$ and total overflow $\mathcal{F}_{ov} = \sum_{\forall b \in E_G} f_{ov}(b)$ to evaluate the congestions in the final results.

For a given set of nets $\mathcal{N}$ and a grid graph G over the area of $\mathcal{N}$, our objective is to construct routing trees T^i for every $N^i \in \mathcal{N}$, such that timing slack (the minimum delay slack among all sinks) $\mathcal{S}(T^i) = \geq 0$ and the congestion is minimized in terms of $\mathcal{D}_{max}$ and $\mathcal{F}_{ov}$. In the second algorithm, an additional objective is to bound the number of bends on each backbone wire to be no greater than 5.

5.3 Routing Flexibilities under Timing Constraints

In this section, we will explore the routing flexibilities under timing constraints. Usually there are many routing tree topologies that can satisfy the required arrival time (RAT) for each sink in a net, if the RAT are in a reasonable range. For example, both P-Tree [14] and RATS-tree [16] can generate a set of routing topologies satisfying timing constraints. Such timing-constrained routing flexibilities are exploited in the global routing work in [58]. In this work, we will use some other types of routing flexibilities including: soft edges, Z-edges, slideable Steiner nodes and edge elongation. The concept of a soft edge is described in Section 2.3. Others will be summarized as follows.

5.3.1 Z-edges

When the number of bends along a route connecting two nodes is restricted to be no greater than two and its path length to be the Manhattan distance between the two nodes, this route can only be straight, L-shaped or Z-shaped. A *Z-edge* is an edge that can take only such a route. Even though the routing flexibility from a Z-edge is less than that of a soft edge, this flexibility can preserve timing performance with bounded number of bends.

5.3.2 Slideable Steiner Node (SSN)

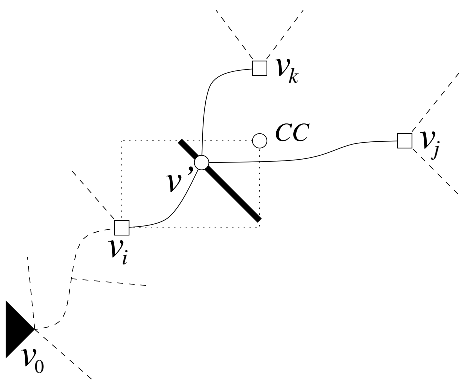


Figure 5.1: An example of a slideable Steiner node (SSN).

We will use the example in Figure 5.1 to describe another timing-constrained routing flexibility. In this example, we consider to join node v_k to edge e_{ij} such that the total wirelength is minimized while the timing constraint at each sink is satisfied. It has been shown in Section 2.4 that the optimal connection point can be off the Hanan grid, such as node v' in Figure 5.1. This point is specified by the Manhattan distance from the upstream end v_i of e_{ij} to v'. There are often many points with the same distance. The set of locations for a distance form a segment of locus as illustrated by the thickened segment in Figure 5.1. When we slide the Steiner node v' along

this locus, the length of its incident edges are preserved and so does the delay at each sink. Similar to the rationale for soft edges, we only specify this locus instead of a point for this Steiner node and call it as *slideable Steiner node* (SSN). The concept of a slideable Steiner node provides extra flexibility for the routes of its incident edges and can again be used to reduce the congestion in global routing without degrading timing performance or area.

5.3.3 Edge Elongation

After performance-driven routing for a net, it is possible that the timing slack of the routing tree is still positive. This positive slack can be consumed through edge elongation to provide more routing flexibilities under timing constraints. Note that either a soft edge or a solid edge may be elongated under timing constraint.

The maximum allowed elongation Δl_{ij} for routing edge $e_{ij} = (v_i, v_j)$ can be computed under the Elmore delay model. We assume v_i is the upstream end of this edge. The length of a routing path from source v_0 to a node $v_i \in V$ is denoted as p_i, and the shared path length for two nodes $v_i, v_j \in V$ from the source is expressed as p_{ij}. For any sink $v_k \in V$, we can compute the maximum Δl_{ij} such that the delay slack $s(v_k)$ is non-negative. If $v_k \notin T_i$, i.e., v_k is not in downstream of v_i,

$$\Delta l_{ij} = \frac{s(v_k)}{(R_d + rp_{ik})c} \tag{5.2}$$

where R_d is the driver resistance and r and c are the wire resistance and capacitance per unit length. If $v_k \in T_i$, Δl_{ij} satisfies the following equation:

$$s(v_k) = f(\Delta l_{ij}) = (R_d + rp_i)c(\Delta l_{ij}) + \frac{1}{2}rc(2l_{ij}(\Delta l_{ij}) + (\Delta l_{ij})^2) + r(\Delta l_{ij})C_j, \tag{5.3}$$

where C_j is the total downstream capacitance from v_j. This equation can be solved to obtain the Δl_{ij}. In the case of double roots for this equation, we choose the one where the slope of function $f(\Delta l_{ij})$ is positive, since the delay slack should be monotonically increasing with respect to the allowed elongation. We compute Δl_{ij} for all the sinks in the routing tree and choose the minimum value as a safe value. Note that different edges may have different values of maximum allowed elongations.

5.4 Hierarchical Algorithm

5.4.1 Algorithm Overview

This algorithm includes three phases: (1) performance driven routing for each net, (2) **HBA**: hierarchical bisecting of routing regions and assigning soft edges to boundaries along the bisector, and (3) **TRR**: timing-constrained rip-up-and-reroute.

In phase 1, each net is routed to meet its timing constraints without considering congestion. Any single-net performance driven routing method, e.g., P-Tree [14], RATS-tree [16] or MVERT [11], can be applied here. Besides satisfying timing constraints, each routing tree should be soft, i.e., should not contain any bend node. This can be achieved through utilizing soft edges during routing as in the example of Figure 2.4 or replacing L-shaped connections in the results with soft edges. Thus, at the end of phase 1, timing-constrained routing trees are generated along with topology flexibilities to be exploited in the subsequent phases.

In phase 2, a routing region is recursively bisected into subregions in a top-down manner. At the topmost level, the whole routing region is bisected into

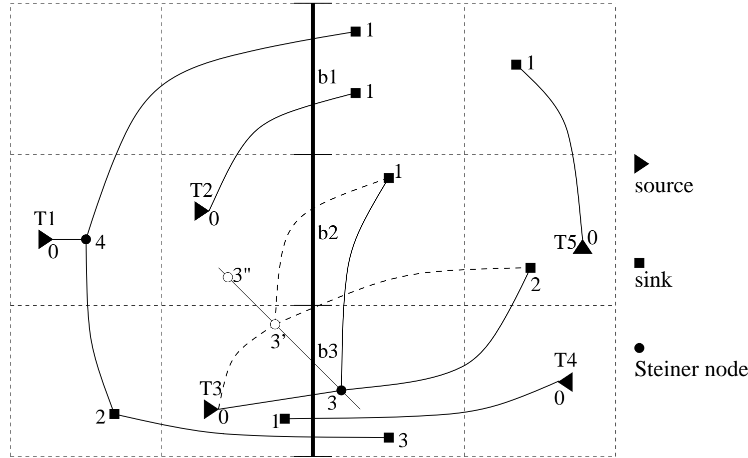


Figure 5.2: An example of bisection.

left(upper) and right(lower) halves with the same or similar size by a bisector line which is formed by a column(row) of consecutive vertical(horizontal) grid cell boundaries. For example, in Figure 5.2, the thickened bisector line is composed of three boundaries, b_1, b_2 and b_3. Each soft edge that intersects this bisector is assigned to a boundary. After the assignment, a pseudo-pin is inserted into the soft edge at the assigned boundary, and therefore this soft edge is split into two new soft edges that belong to two separate subregions. One assignment for the example in Figure 5.2 is shown in Figure 5.3. In the next hierarchical level, bisections and assignments are applied on the left(upper) and right(lower) half region along an orthogonal orientation. This process is repeated until the subregion is a single grid cell or a pair of neighboring grid cells. Thus, at the end of this process, the route for each soft edge is specified to the detailed level of grid cells it goes through.

The crucial part is to determine how to assign the soft edges to the boundaries on the bisector line. The basic goal is to assign *all* of the soft edges without exceeding any boundary supply and without causing any delay violations. The absence of delay violation implies that the delay slack for each net is non-negative. In order to make the assignment feasible, sometimes it is

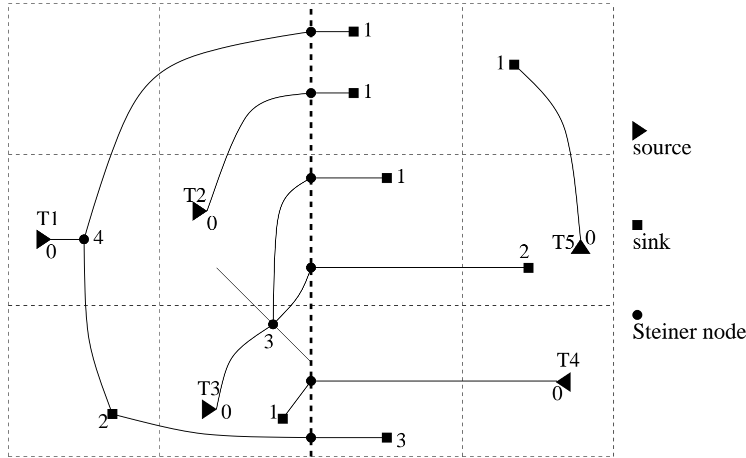


Figure 5.3: An assignment result from network flow solution.

necessary to allow some wires to detour, which inevitably increases delay, i.e., some timing slack is consumed to reduce congestion. In addition to ensuring absence of delay violations, it is naturally desirable that the consumption of the timing slack is minimized, since the timing slack may be needed in the subsequent levels of bisection and assignment. These objectives are achieved through a min-cost network flow formulation. Because of the involvement of timing issues, this formulation is not as straightforward as that in [60,61]. We run a min-cost max-flow algorithm [63] to solve this network flow problem. It is well known that there are polynomial time optimal algorithms for the min-cost max-flow problem.

The hierarchical bisection and assignment in phase 2 is a method of divide-and-conquer that has the advantage of simplifying the problem nature. In this global routing approach, it reduces a two-dimensional problem into one dimension. The price that this simplification inevitably pays is on congestion reduction, since a decision at a higher hierarchical level may overlook the needs at a lower level. In phase 2, any soft edge that could not be assigned in the network solution is temporarily assigned to a boundary such that the maximum demand density is minimized and no delay violation is

incurred. These residual overflows will be cleaned in phase 3.

The third phase is a timing-constrained rip-up-and-reroute process. It is similar to traditional rip-up-and-reroute except that a constraint on edge length is imposed to ensure no timing violation. It rips up the edges on a set of most congested boundaries and reroutes them through maze routing. The cost in maze routing is defined as the summation of demand densities over all boundaries that a path passes through, and these densities are dynamically updated. The edge length can be elongated to the extent that no delay violation is incurred.

5.4.2 Basic Network Formulation

After one bisection, the assignment problem is formulated as:

Problem 5.1 *Given a bisector line B composed of a set of consecutive boundaries $\{b_1, b_2, ...\}$, and a set of soft edges $E_X = \{e^i_{jl} | e^i_{jl}$ intersects $B\}$, assign each soft edge to a boundary $b_k \in B$ such that there is no overflow on any boundary $b_k \in B$ or no delay violation on any routing tree T^i which has at least one soft edge $e^i_{jl} \in E_X$, and the timing slack consumption is minimized.*

We solve this problem through a formulation of the network flow problem and applying a min-cost max-flow algorithm on it. The network $G_F(V_F, A_F)$ is a directed graph consisting of a set of vertices V_F and arcs A_F. The vertex set V_F includes all boundaries in B and soft edges in E_X, plus a source s and target t. For the bisection in Figure 5.2, its corresponding network is illustrated in Figure 5.4. We do not use SSN at this moment for simplicity and only e^3_{03} in T^3 is included in the network. The usage of SSN will be introduced in section 5.4.4. There are three types of arcs: (1) from source s

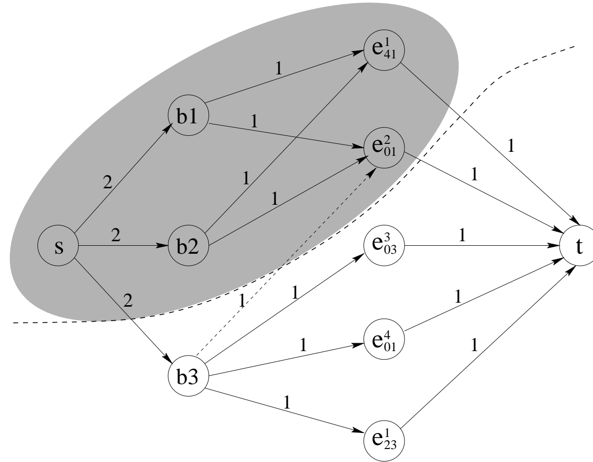


Figure 5.4: Network formulation of the example in Figure 5.2 without considering SSN. The number on each arc is its capacity.

to every boundary vertex, (2) from some boundary vertices to some soft edge vertices, (3) from every soft edge vertex to the target t. Each arc has a cost and a capacity associated with it. For each type 1 arc, its cost is 0 and its capacity is the corresponding boundary supply. In this example, we assume that each boundary has a supply of 2. For each type 2 arc, its capacity is 1 and its cost will be defined later. For each type 3 arc, its capacity is 1 and its cost is 0.

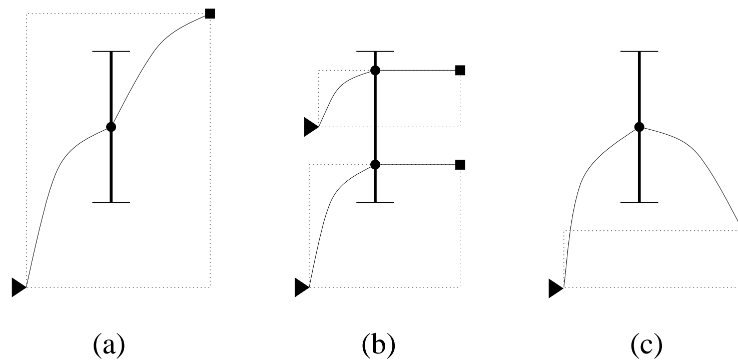


Figure 5.5: Relative positions of a boundary and a soft edge.

An arc from a boundary vertex to a soft edge vertex implies a candidate assignment between them. Not every pair of boundary and soft edge vertices

is automatically qualified for constructing a type 2 arc between them. For any boundary and any soft edge, there are three relative positions between them as shown in Figure 5.5. In Figure 5.5(a), the boundary lies entirely within (the bounding box of) the soft edge. If we choose an assignment of the soft edge to this boundary, there will be no change in the length of the soft edge, and two vias are induced. If a boundary lies partially within the bounding box of a soft edge, as in Figure 5.5(b), we have an *L-intersection* between the boundary and the soft edge, where no change in the soft edge length is required and one via is induced. In either of these two cases, i.e., if a boundary is within or has an L-intersection with a soft edge, we can always set up an arc between them without affecting the delay. These arcs are called *basic arcs*, and they are the solid type 2 arcs in Figure 5.4. The third situation is shown in Figure 5.5 (c), where the soft edge does not intersect with the boundary. In this case, an assignment on this pair will require a wire detour, and we need to check whether or not this may cause any delay violation. An arc can be constructed for such a pair only if the assignment on this pair will not cause any delay violation. For the example in Figure 5.2, if the timing slack of T^2 remains non-negative when the soft edge e^2_{01} goes through boundary b_3, then an arc (a dashed line) between them is constructed in Figure 5.4. We call such a construction as an *edge expansion* and each expansion implies a timing slack consumption.

We categorize the trees across the bisector line B into single-crossing trees and multi-crossing trees, which are the trees that cross B only once (such as T^2 in Figure 5.2) and more than once (such as T^1 in Figure 5.2), respectively. Initially, we construct all the basic arcs for all the soft edges in E_X and perform an expansion for all the soft edges that belong to single-crossing trees. The expansions of edges in multi-crossing trees will be discussed in

the next section.

The cost of a type 2 arc is defined according to the timing slack of its corresponding tree, since one major objective is to minimize timing slack consumption. If the timing slack of tree T^i is $S_{old}(T^i)$ before the assignment, and is $S_{new}(T^i)$ if its soft edge e^i_{jl} is assigned to boundary b_k, then we define the arc cost as:

$$cost(b_k, e^i_{jl}) = \frac{S_{old}(T^i)}{S_{new}(T^i)}. \tag{5.4}$$

It can be seen that if an edge intersects with a boundary entirely or partially, its corresponding type 2 arc has a cost of unity. As a secondary objective, we hope to reduce the number of vias in the wiring. Therefore, for the situation in Figure 5.5(b), we reduce its cost by a small user-specified offset $\epsilon, 0 < \epsilon < 1$.

5.4.3 Construction of Arcs for Multi-crossing Trees

Generally speaking, adding a type 2 arc between a boundary vertex and a soft edge vertex may increase the likelihood of obtaining a feasible network flow solution. Hence, an edge expansion is usually desired as long as no delay violation is incurred. One issue that was not discussed in the last section is the procedure for those soft edges that belong to multi-crossing trees, such as T^1 in Figure 5.2. The difficulty here is that the timing slack computations for the soft edges are correlated. For some specified timing constraints, whether an edge can be expanded, or how far it can be expanded, depends on whether other crossing edges in the same tree are expanded, and how far they have been expanded. For example, in Figure 5.2, the expansion of e^1_{41} depends on whether e^1_{23} has been expanded and how far, i.e., to b_2 or to b_1. In fact,

these edges compete with each other on a common timing slack resource, which must be allocated properly. A uniform allocation may overlook local congestion distribution, and result in some unnecessary expansions while some necessary expansion is not performed.

We solve this difficulty by identifying the necessary expansions through the min-cut method. It is well known that the max-flow equals the forward capacity of the $s-t$ min-cut in a network flow problem [63]. In the beginning, we run a max-flow algorithm on the partially constructed network to obtain an $s-t$ min-cut $(X, \bar{X}), s \in X,\ t \in \bar{X}$. The forward capacity of this cut is denoted by $U_{min}(X, \bar{X})$. If $U_{min}(X, \bar{X}) \geq |E_X|$, then it is guaranteed that every edge can be assigned to a boundary without any overflow, and thus, no more expansion is necessary. Otherwise, the maximum feasible flow is less than the number of edges to be assigned, thus we need to increase the capacity of the min-cut through additional edge expansions. In the example for Figure 5.2, before the expansion for multi-crossing trees, the min-cut is indicated in the dashed curve in Figure 5.4, where the vertices in X are in the shaded region and vertices in $\bar{X}$ are unshaded. We can see that the forward capacity $U_{min}(X, \bar{X}) = 4$ while there are 5 edges that need to be assigned, thus, we need to expand some edge(s) from the multi-crossing tree T^1 if possible.

The min-cut result shows us not only whether more expansions are necessary but also the congestion distribution information or where to make the expansion. Every forward arc in the min-cut must be saturated [63], e.g., $(s, b_3), (e^1_{41}, t)$ and (e^2_{01}, t) are saturated. If a soft edge vertex e^i_{jl} is in X, its downstream arc must be saturated and therefore, it can always be assigned to a boundary without inducing overflow, i.e., it is not in a congested area. On the other hand, if a boundary vertex b_k is in $\bar{X}$ (and not all of its down-

stream arcs are saturated), its upstream arc must be saturated and the soft edges corresponding to its downstream vertices are located in a congested area. Adding an arc from a boundary vertex $b_k \in X$ to a soft edge vertex $e^i_{jl} \in \bar{X}$ matches a soft edge in a congested area to an uncongested boundary.

Lemma 5.1 *The necessary and sufficient condition to increase the max-flow f_{max} of a network is to add a forward arc between X and $\bar{X}$ for every min-cut $(X, \bar{X})$ with $U_{min}(X, \bar{X}) = f_{max}$.*

We make a sweep among all the soft edges in multi-crossing trees and pick at most one soft edge from each tree to expand in order to increase the capacity of min-cut. More precisely speaking, for each multi-crossing tree T^i, from all the $b_k \in X$ and $e^i_{jl} \in \bar{X}$ pairs, we choose one with minimum cost to add an arc between them if no delay violation is induced. After one iteration of expansions, we run the max-flow min-cut algorithm again to repeat this process until $U_{min}(X, \bar{X}) \geq |E_X|$ or no more feasible arc can be found. Note that the timing slack computation in a later iteration of expansions should account for any wire detour in other soft edges of the same tree in previous expansions. In the example in Figure 5.4, We can make an expansion between $b_2 \in X$ and $e^1_{23} \in \bar{X}$ if no delay violation is induced, and then the network problem becomes feasible. The iterative min-cut and expansion technique makes the allocation of timing slack in multi-crossing trees adaptive to the congestion distribution, and expansions are made only when necessary, without waste.

5.4.4 Utilization of Slideable Steiner Nodes (SSN)

In phase 1, if we use the MVERT algorithm together with soft edges, we can have a slideable Steiner node that provides extra flexibility in routing.

The appealing feature of SSN is that when we slide it along its locus, the timing performance is preserved, i.e., no timing slack is consumed. Again, we integrate this flexibility into the formulation of the network flow problem so that it can be exploited in a unified network flow solution.

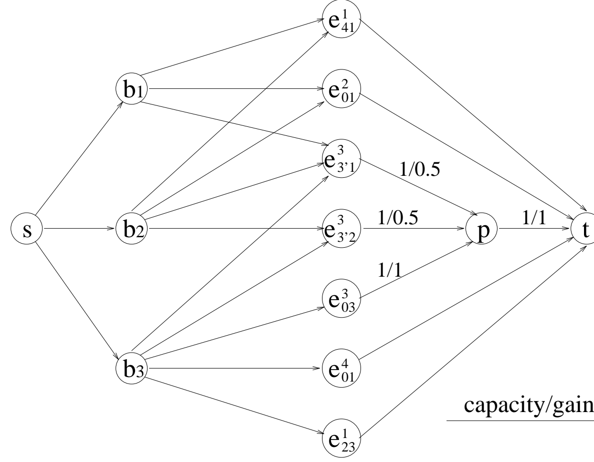


Figure 5.6: Network formulation considering SSN.

The positions of a SSN within a grid cell do not affect wire congestion distributions, hence we can consider one arbitrary position for a SSN within a grid cell. For each SSN whose locus intersects with B, we consider only two candidate positions, each on a different side of the bisector line B, such as v_3^3 and $v_{3'}^3$ in Figure 5.2. We need to consider candidate positions on both sides of B, since they result in remarkably different intersections between their incident soft edges and the bisector line B. On each side of B, we only consider the grid cell that has a boundary in B such that this boundary intersects the locus of the SSN, since the SSN position in this grid cell can provide the maximum overlap between its incident soft edge(s) and B. For example, in Figure 5.2, $e_{3'2}^3$ intersects with two boundaries b_2 and b_3, while $e_{3''2}^3$ would intersect only with b_2. It is evident that a larger overlap implies a larger number of basic arcs which are preferred as they will not consume timing slacks. For v_3^3 and $v_{3'}^3$, all three associated soft edges $e_{3'1}^3$, $e_{3'2}^3$ and e_{03}^3 are included in

the vertices in the network as shown in Figure 5.6. Obviously, e^3_{03} cannot be assigned simultaneously with $e^3_{3'1}$ or $e^3_{3'2}$. This exclusiveness constraint can be instantiated through adding a pseudo-vertex p and applying generalized network flow algorithm [63]. In a generalized network flow problem, each arc has a gain factor associated with it. For example, the amount of flow will reduce 50% after passing through an arc with gain factor of 0.5. We solve this generalized network flow problem through Wayne's algorithm [64].

After the assignment, only one of the candidate SSN positions is selected. The locus of the SSN is truncated at the intersection with B, and the part where the selected position located would be retained, as shown in Figure 5.3.

5.4.5 Network Pruning

Since the computational complexity of the min-cost max-flow algorithm depends on the number of arcs and vertices in the network, it is always desirable to reduce the number of arcs and vertices without affecting the quality of solution. One simple observation is that if a soft edge vertex has only one upstream incident arc and this arc is a basic arc, we can assign it to its upstream boundary vertex immediately and remove it and its incident arcs from the network. For a boundary vertex b_k, if the number of its downstream arcs is no larger than the capacity of its upstream arc, then all of its downstream soft edge vertices can be assigned to it without inducing overflow, i.e., this boundary is not congested. Then, for each of its downstream soft edge vertex e^i_{jl} we can remove its upstream arcs whose cost is larger than $cost(b_k, e^i_{jl})$, since we can always assign e^i_{jl} to b_k with less cost and without inducing overflow.

5.4.6 Experimental Results

The experiments aim to test the effect of the proposed algorithm on both timing and congestion. Traditional rip-up-and-reroute(RR) and timing-constrained rip-up-and-reroute(TRR) methods are tested together with our algorithm(HBA+TRR) on the same set of circuits. The circuits that we tested belong to the CBL/NCSU benchmark suite whose statistics are shown in Table 5.1. For some circuits, more than one set of netlists are obtained through different placements. We have implemented these methods in C++ and conducted experiments on a SUN Ultra-10 workstation. The initial routing trees are obtained through MVERT [11] algorithm so that they must satisfy timing constraints. The results are listed in Table 5.2.

Table 5.1: Benchmark circuits.

Circuit	# modules	# nets	# pins
apte	9	45	162
ami33	33	85	480
ami49	49	390	913
xerox	10	203	696

The congestion results are expressed in terms of total overflow $\mathcal{F}_{ov}$ and the maximum demand density $\mathcal{D}_{max}$(both are defined in section 5.2). The congestion results from rip-up-and-reroute(RR) are generally good. When we look at the congestion results from TRR, we can see that they are much worse than RR, because the algorithm may get stuck in a deadlock and fail to find a solution under timing constraints. A naive combination of timing constraints with rip-up-and-reroute does not work, and a crafted approach is necessary to optimize these two competing objectives simultaneously. Table 5.2 also shows the congestion results from HBA for reference. Even though

they are usually better than TRR, they are not ideal yet and not comparable with those of RR, and should be considered as intermediate results. When we combine TRR with HBA, the congestion results are found to be good and are mostly better than even RR, which does not satisfy the timing constraints. Since hierarchical approach is better at a global planning level while rip-up-and-reroute is specialized to find local and more detailed routes, it is natural that a combination of these two complementary approaches can yield a good result on congestion reductions. When we compare the timing results, it is not surprising that only RR causes delay violations while there is no delay violation in the results from TRR or our algorithm. The percentage of nets with delay violations from RR are listed in column 6, and ranges from 6−32%.

Table 5.2: Experimental results on timing-constrained global routing.

			Rip-up-and-reroute			TRR		HBA		HBA+TRR		
Circuit	Grid	$\|E\|$	$\mathcal{F}_{ov}$	$\mathcal{D}_{max}$	DV %	$\mathcal{F}_{ov}$	$\mathcal{D}_{max}$	$\mathcal{F}_{ov}$	$\mathcal{D}_{max}$	$\mathcal{F}_{ov}$	$\mathcal{D}_{max}$	CPU
apte	45 × 55	145	1	1.00	9	56	1.50	19	1.83	0	1.00	7.1
ami33.1	19 × 31	489	0	1.00	21	15	1.29	10	1.57	0	1.00	30.3
ami33.2	19 × 27	497	0	1.00	12	19	1.25	0	1.00	0	1.00	26.6
ami49.1	43 × 45	594	2	1.09	17	11	1.18	16	1.36	1	1.09	19.6
ami49.2	44 × 44	594	0	1.00	12	61	1.56	0	0.94	0	0.94	12.2
xerox.1	24 × 24	583	2	1.06	6	41	1.38	1	1.06	0	1.00	12.8
xerox.2	28 × 32	569	0	1.00	11	17	1.22	1	1.05	0	1.00	16.3
xerox.3	41 × 34	569	3	1.11	32	43	1.33	0	1.00	0	1.00	23.8

The total CPU time for three phases of our algorithm on each circuit are listed in the rightmost column in seconds. Since each circuit has different number of nets and the number of pins on one net may be between two and several dozens, it would be more interesting to evaluate the average CPU time on each 2-pin net as a normalized comparison. The third column, $|E|$ gives the total number of soft edges from the initial routing trees in each circuit. It is conceivable that the formulation of soft edges is equivalent to a decomposition to 2-pin nets. Based on this data, the average CPU time is

found to be 0.06 second/2-pin-net in the worst case.

5.5 Gradual Refinement Algorithm

5.5.1 Approximated Congestion Estimation

In addition to the traditional congestion metrics, we use a couple of other approximate estimation methods during different phases of the second global routing algorithm we proposed, all of which will be introduced in details as follows.

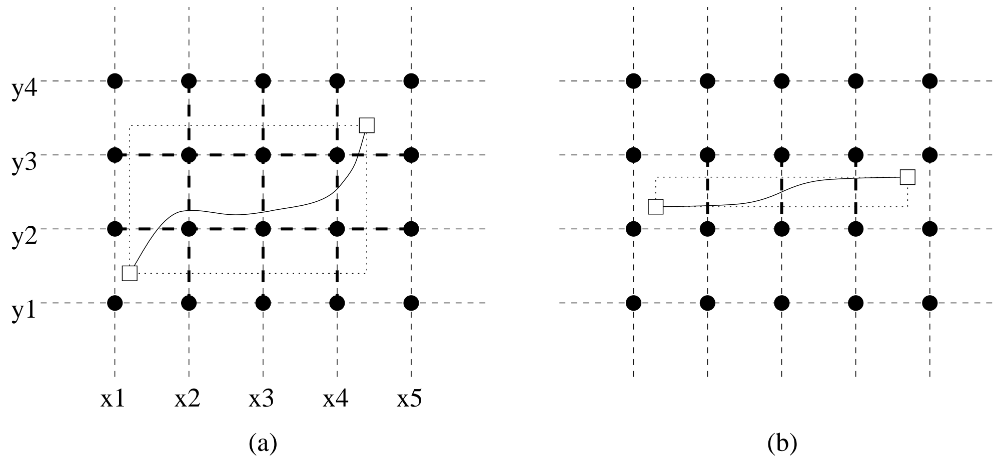


Figure 5.7: Examples of primitive demand. (a) each grid edge corresponds to a horizontal(vertical) thickened boundary segment has primitive demand of $\frac{1}{4}(\frac{1}{3})$. (b) each grid edge corresponds to a thickened boundary segment has primitive demand of 1.

The *demand* defined in section 5.2 is based on the the wire routes in solid edges. However, we arrive at a rough estimation of the congestion from soft edges. We use the concept of *primitive demand* to indicate the possibility of wires crossing a grid edge. This is demonstrated in the example in Figure 5.7,

in which the dashed segments represent the tessellation by the grid graph, G. The bounding box for a soft edge e_{ij} is obtained as the dotted rectangles and denoted as B_{ij}. In Figure 5.7(a), there are three vertical boundary segments at $x = x_2$ overlapping with B_{ij} and only one of them is crossed by e_{ij} in the final route. Therefore, we define the primitive demand incurred by e_{ij} over the grid edge corresponding to each of these three boundary segments as $\frac{1}{3}$. Similarly, each grid edge for a thickened horizontal (vertical) boundary segment in Figure 5.7(a) has a primitive demand of $\frac{1}{4}(\frac{1}{3})$. We refer to the demand, as defined in section 5.2, as the *determined demand*, which actually is a special case of the primitive demand. A primitive demand for a soft egde is equivalent to its determined demand when its value aggregates to 1, as in Figure 5.7(b). If we denote the primitive demand incurred by routing edge e_{ij} over grid edge b as $d_{prim}(b, e_{ij})$, then:

$$d_{prim}(b) = \sum_{\forall e_{ij} \text{intersecting } b} d_{prim}(b, e_{ij}) \tag{5.5}$$

Definition 5.1 (primitive demand) *If the bounding box B_{ij} of an edge e_{ij} passes through m rows and n columns in the grid graph G, the* primitive demand *$d_{prim}(b, e_{ij})$ from e_{ij} on each grid edge b corresponding to a vertical(horizontal) boundary overlap with B_{ij} is $\frac{1}{m}(\frac{1}{n})$.*

A Z-edge has less routing flexibility than a soft edge and its possible routes are easier to enumerate. In this scenario, we adopt a probabilistic estimation which is similar to [65, 66] and will be illustrated in the example in Figure 5.8. Without loss of generality, we can arbitrarily specify one end of the soft edge as source node v_s and the other end as target node v_t, and denote the grid cell in which they are located as (r_s, c_s) and (r_t, c_t), respectively. We use

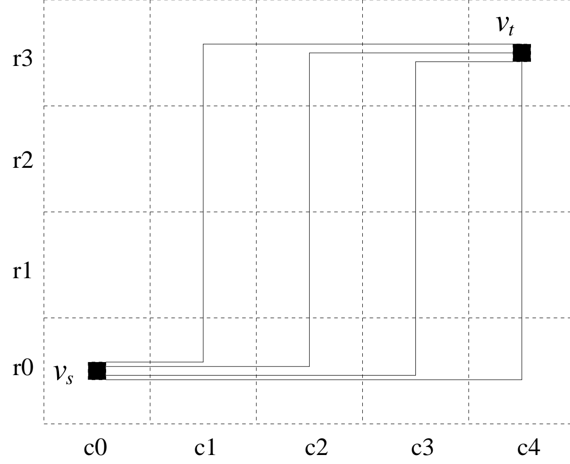


Figure 5.8: Enumerate routes with number of bends less than 3 to obtain probabilistic demand.

r and c to represent row and column indices. Through simple enumeration, we can conclude that:

Lemma 5.2 *There are* $Z = |r_t - r_s| + |c_t - c_s|$ *monotone routes with number of bends no greater than* 2 *between two grid cell* (r_s, c_s) *and* (r_t, c_t) *in a grid graph.*

For example, there are seven possible route for the Z-edge in Figure 5.8. In our congestion estimation, we assume a uniform probability distribution for these routes, i.e., every route has the same chance to be chosen in later stages. Then, we can obtain the probability that a grid edge is crossed by the soft edge, again through simple enumeration. To simplify the description, we initially consider only those routes that leave source node v_s horizontally toward the target node. Obviously, there are $|c_t - c_s|$ such routes which are depicted in Figure 5.8. For each horizontal grid edge above the grid cells $\{(r_i, c_i)|r_s \leq r_i < r_t, c_s < c_i \leq c_t\}$, there is one route across it. Thus, in the example of Figure 5.8, the probability that the soft edge runs across the

horizontal grid edge above cell (r_1, c_2) is 1/7. For each vertical grid edge to the right of grid cell $(r_s, c_j), c_s \leq c_j < c_t$, the probability is $|c_t - c_j|/Z$. For example, the probability that the wire run across the vertical grid edge to right of grid cell (r_0, c_1) is 3/7. The probability at other grid edge can be counted similarly. Based on these probabilities, we define the probabilistic demand as follows:

Definition 5.2 (probabilistic demand) *The probabilistic demand from a Z-edge e_{ij} to a grid edge b is the probability that Z-edge run across this grid edge, and is denoted as $d_{prob}(b, e_{ij})$.*

In this definition, we restrict the number of bends for each soft edge to be no greater than 2. We can also relax the number of bends to be a specific number greater than 2, and then the probability estimation can be obtained by using the recursive technique described in [67].

5.5.2 Algorithm Motivation

It is well-known that even minimizing only congestion for only 2-pin nets is an NP-complete problem, and considering timing constraints and the number of bends makes the problem even harder. The objectives of congestion, timing constraints and number of bends often compete with each other in global routing.

Our strategy is to obtain the required timing performance first and then concentrate on optimizing the congestion and the number of bends while preserving the timing performance obtained. Therefore, we initially route each net individually through timing driven algorithms without considering congestion or the number of bends. We use soft edges and slideable Steiner

nodes in this phase so that the routing result is composed of only backbone nodes and every backbone wire is a single edge.

In the second phase, we will try to specify the details for the slideable Steiner nodes and backbone wires in an effort to minimize congestion and control the number of bends on each backbone wire. The strategy here is to refine the route gradually according to available congestion information, even this information is not accurate.

We can compare the underlying mechanism with the sequential and rip-up-and-reroute approach. In the sequential approach, the earlier routing steps are performed without any knowledge of the locations of the subsequent nets, and are therefore somewhat blindfolded. The routing of nets that are considered later is based on the routes of previous nets, which may be suboptimally placed, due to this false feedback. Similarly, rip-up-and-reroute, which proceeds according to the locations of other nets, may also be suboptimal. Due to its iterative nature, rip-up-and-reroute method has the ability to correct the false feedback gradually and may become successful in reducing congestion after many iterations. However, the efficacy of rip-up-and-reroute may be hindered if constraints on timing and the number of bends are imposed.

Fortunately, our method can obtain some rough congestion estimation based on the locations of soft edges after phase one. This estimation is by no means completely accurate, but it is better than no feedback or incorrect feedback. Since we know that this estimation is not entirely accurate, we will not fix the route completely in one step. Instead, we will settle a part of the route to obtain a better congestion estimation and complete the routes gradually to avoid blindfolded or incorrect decision.

5.5.3 Algorithm Detail

Our algorithm includes two phases:

(I) Timing-driven routing for each net individually without considering congestion or number of bends.

(II) Specifying the route for each backbone wire obtained in phase I so that congestion is minimized subject to timing constraints and bends constraints.

In phase I, we route each net through the MVERT [11] algorithm using soft edges and slideable Steiner nodes so that the timing constraints can be satisfied and the resulting routing tree consists of only backbone nodes and each backbone wire is either a solid edge or a soft edge.

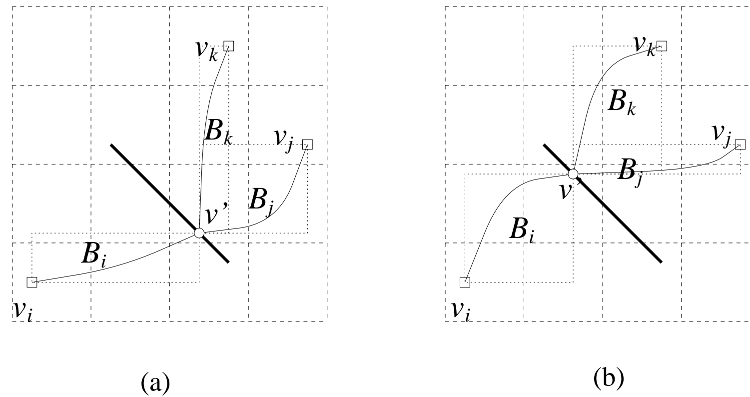


Figure 5.9: When an SSN slides along its locus, the bounding boxes of its incident edges change as well as the primitive demands.

After phase I, we can obtain a rough estimation of congestions through the concept of *primitive demand* defined in section 5.5.1. The first step in phase II is to fix the position of each SSN (Slideable Steiner Node) to minimize the peak primitive demand density. Recall that when an SSN slides along a locus of points, the lengths of its incident edges are not changed, and nor is the delay at any sink. An example of a sliding Steiner node is illustrated in Figure

5.9, and it can be seen that the bounding boxes of the incident edges, denoted by B_i, B_j and B_k, are changed after the sliding of the SSN. In Figure 5.9, when the Steiner node v' is slid toward northwest from (a) to (b), the bounding box B_i becomes thinner and taller and the vertical (horizontal) primitive demand incurred by (v_i, v') on each horizontal (vertical) cell boundary overlaps with B_i may be larger (smaller). The effect of this move on B_j and B_k is the opposite. Based on this observation, we can tune the position of the SSN on its locus in a way such that the maximum demand density among the grid edges involved is minimized. This objective is achieved through a linear search for all the grid cells that the SSN intersects, and fixing the location in a grid cell such that the maximum demand density is minimized.

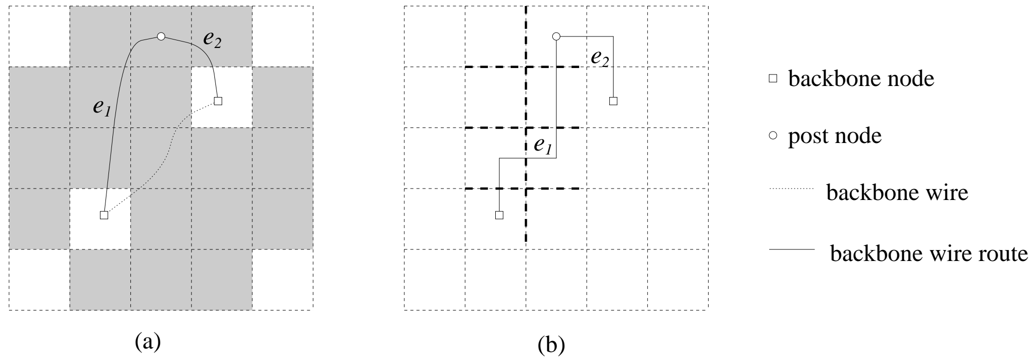


Figure 5.10: Examples for setting post node for a backbone wire.

After fixing the SSN, we will make two sweeps of all the backbone wires in a fixed order to specify their routes. Instead of specifying the complete route immediately in one step, we first only specify *one* grid cell that the backbone wire has to pass through. Note that neither of the end nodes of this backbone wire can be within this grid cell. We insert a pseudo node, which we call the *post node* in the backbone wire within this selected grid cell. For example, in Figure 5.10(a), a post node represented by a small circle is inserted into the backbone wire represented by a dotted curve. Before choosing the grid

cell for the post node, we need to choose the candidate grid cells that will be considered. The routing flexibility from edge elongation is utilized here. For a backbone wire e_{ij} with edge length l_{ij}, we calculate the maximum allowed elongation Δl_{ij} under timing constraints as in section 5.3.3. If we insert a post node v_k into e_{ij}, the location of v_k must satisfy $l_{ik} + l_{kj} \leq l_{ij} + \Delta l_{ij}$. In Figure 5.10(a), the candidate grid cells are shaded.

Similarly, a formerly solid edge can also be elongated to reduce congestion. Thus, we see that a "solid" edge is not exactly solid any more, and can have some routing flexibility as well. Note that the maximum allowed elongation for each backbone wire is calculated dynamically, since the allowed elongations for backbone wires in a same routing tree depend on each other.

After the post node is inserted, the former backbone wire is split into two subedges. We specify that each subedge can take only a Z-edge. By setting the post node and restricting to Z-edges, we can bound the number of bends for each backbone wire to be no greater than five. Moreover, we can obtain a better estimation of the congestion by using the probabilistic demand defined in section 5.5.1. We choose the post node so as to minimize the congestion cost for the two sub-edges. Before setting the post node for a backbone wire e_i, we need to remove the primitive demand generated from the soft edge of e_i. Then, the congestion cost of a subedge e_j is defined as:

$$cost(e_j) = \sum_{\forall b \text{intersecting } e_j} \mathcal{D}(b)^2 \times d_{prob}(b, e_j). \tag{5.6}$$

Algorithm: ZigPuzzle$(\mathcal{N}, G)$
Input: A set of nets $\mathcal{N}$, grid graph $G(V_G, E_G)$ **Output:** T^i for each N^i min congestion s.t. timing constraints, ≤ 5 bends per backbone wire
1. For each $N^i \in \mathcal{N}$ 2. $T^i \leftarrow$ MVERT(N^i) 3. Generate primitive demand for each backbone wire 4. Fix all of the SSNs, min peak demand density 5. For each tree T^i 6. For each backbone wire $e^i_{jk} \in T^i$ 7. Remove its primitive demand 8. Compute max elongation Δl_{jk} without timing violation 9. Find candidate grid cells for post node 10. Find a post node, min congestion cost 11. Generate probabilistic demand from two Z-edges 12. For each tree T^i 13. For each backbone wire $e^i_{jk} \in T^i$ 14. Recompute its post node 15. Remove its probabilistic demand 16. Fix routes for two Z-edges, min congestion cost 17. Generate determined demands from two Z-edges

Figure 5.11: The gradual refinement global routing algorithm.

Recall that $\mathcal{D}(b) = \frac{d(b)}{s(b)}$ is the demand density at boundary b and $d_{prob}(b, e_j)$ is the probability that the Z-edge e_j runs across boundary b. In the example in Figure 5.10(b), the cost of edge e_1 is the summation of the estimated costs from thickened grid edges.

After setting the post node, we generate the probabilistic demands from the two new Z-edges. The process of setting the post node is performed for each backbone wire in every routing tree, which is the first sweep. During

this sweep, the backbone wires that have been processed are Z-edges while those have not been processed are still soft edges. Thus, in the congestion cost computation, both primitive and probabilistic demand may co-exist at the same time. Therefore, we multiply the primitive demand by a penalty coefficient of less than one to make its contribution weaker than that of the probabilistic demand.

After post nodes have been selected for all of the backbone wires, all of the demands become probabilistic demands. Based on this improved congestion estimation, we start the second sweep for all backbone wires to specify their routes in the same order as in previous sweep. For each backbone wire, we recompute its post node before fixing the routes of its two Z-edges. A backbone wire appears early in the order list may have a poor post node location in the previous sweep, since this location is chosen according to mostly primitive demands. In this second sweep, this backbone wire has a chance to adjust its post node location from a more accurate congestion information. The procedure of replacing the post node is the same as in previous sweep. Then, we choose the route for the Z-edge e_j to minimize congestion in term of a cost defined as:

$$cost(e_j) = \sum_{\forall b \text{intersecting } e_j} \mathcal{D}(b)^2. \tag{5.7}$$

The minimum cost route can be found through simple enumeration in a manner similar to calculating the probabilistic demand in section 5.5.1. After fixing the routes for each backbone wire, its probabilistic demand is replaced by determined demand. The complete algorithm is summarized in Figure 5.11.

5.5.4 Experimental Results

We have implemented ZigPuzzle global routing in C++ and conducted experiments on a SUN Ultra-10 workstation. The circuits that we tested includes benchmark suite $ami33$, $ami49$ and $xerox$ and three sets of randomly generated nets, whose statistics are shown in Table 5.1 and 5.3. Similar to section 5.4.6, each result is measured in terms of two congestion metrics: total overflow $\mathcal{F}_{ov}$ and maximum demand density $\mathcal{D}_{max}$.

Table 5.3: Description of Test Circuits.

Circuit	# nets	# pins
$test1$	1109	2464
$test2$	2100	4576
$test3$	3107	6817

For comparisons, we also implemented three variations of rip-up-and-reroute (RR) global routing algorithm. In the base version of RR, we initially route each net separately in MVERT same as in phase I of ZigPuzzle but using solid edges. Then, we rip up every backbone wire in the region with wire overflow, and reroute them through maze routing to minimize congestion cost which is the same as Equation 5.7 except that the demand is determined. Three variations are: RR+B (RR with bends control), RR+T (timing-constrained RR) and RR+B+T (timing-constrained RR with bends control). In order to control the number of bends in RR, we replace the cost in maze routing with a weighted sum of congestion and number of bends. We run the RR+B with several different values of weight and choose the result that provides the best congestion, while assuring that the number of bends for each backbone wire is no greater than five, which is the same as in ZigPuzzle. In RR+T, the timing constraints are imposed on the wirelength

for each backbone wire. Both of these methods are combined in obtaining the results for the RR+B+T case. We did not compare with the hierarchical method in section 5.4, because of the difference on objective.

Table 5.4: Grid size and the number of backbone wires for each circuit.

Circuit	Grid size	$\lvert E \rvert$
ami33.1	22 × 36	489
xerox.1	54 × 55	587
xerox.2	54 × 61	571
ami49.1	40 × 41	594
ami49.2	52 × 53	593
test1	52 × 52	1468
test2	53 × 53	2648
test3	52 × 52	3992

Table 5.5: Experimental results, *vio* is the number of nets with timing violations and *ben* is the maximum number of bends on a backbone wire.

	RR+B			RR+T			RR+B+T		ZigPuzzle		
Circuit	$\mathcal{F}_{ov}$	$\mathcal{D}_{max}$	*vio*	$\mathcal{F}_{ov}$	$\mathcal{D}_{max}$	*ben*	$\mathcal{F}_{ov}$	$\mathcal{D}_{max}$	$\mathcal{F}_{ov}$	$\mathcal{D}_{max}$	CPU
ami33.1	0	1.00	5	1	1.20	12	1	1.20	0	1.00	15.8
xerox.1	0	1.00	6	5	1.14	14	5	1.14	0	0.86	16.1
xerox.2	0	1.00	6	2	1.06	12	2	1.06	2	1.06	23.3
ami49.1	1	1.10	34	0	1.00	18	5	1.10	0	1.00	29.8
ami49.2	0	1.00	38	0	1.00	14	51	1.80	2	1.10	25.9
test1	4	1.29	34	0	1.00	17	58	1.21	1	1.07	207
test2	5	1.13	28	0	1.00	17	107	1.12	0	1.00	562
test3	4	1.16	35	0	1.00	20	63	1.14	0	1.00	1013
Average	1.8	1.09	23	1	1.05	16	36.5	1.22	0.6	1.01	237
Comp	3×	1.05×		1.67×	1.03×	3.2×	61×	1.20×			

The experimental results are shown in Table 5.5 and 5.4. In Table 5.5, Column 2 to 4 are results from RR+B, whose number of bends is bounded to be no more than five for each backbone wire. Its congestion results are

generally good with limited wire overflows in large circuits. However, it does not have the capability to satisfy timing constraints and there are always a number of nets that have timing violations as indicated in column 4. The results from RR+T is given in column 5 to 7, where the congestion is good subject to timing constraints, but the maximal number of bends per backbone wire can be very large (in a range of $12 - 20$) as listed in column 7 of Table 5.5. When we impose both timing and bends constraints directly on RR, the results in column 8 and 9 show that the congestion can be very poor especially in large circuits. On the other hand, our ZigPuzzle can optimize congestion subject to timing and bend constraints, and works well in both small and large circuits. In the last row of Table 5.5, the results from RR are compared with that of ZigPuzzle.

The rightmost column lists the CPU time in seconds. Since a circuit may include many multi-pin nets, it would be more interesting to evaluate the CPU time for each 2-pin net as a normalized comparison. Column 3 in Table 5.4 lists the number of backbone wires in each circuits. It is easy to regard these backbone edges as a decomposition into 2-pin nets. Therefore, we can get the average CPU time for a 2-pin net is 0.2 seconds.

In our approach, we use primitive demand to estimate the congestions from soft edges. Instead of this, we can use a probabilistic estimation similar to that in section 5.5.1 for soft edges by using the technique described in [67], and it may initially seem that this would provide a more accurate estimate of congestion. We have also performed experiments using this approach. However, we find that this approach takes a longer CPU time and often produces worse results. This increased CPU is not unexpected, since the computation related to probabilistic estimation will be more computationally intensive than primitive demand calculations, especially when the allowed

number of bends is large. In addition, a backbone wire could be elongated at a later stage, and therefore, the increased accuracy obtained from probabilistic estimation at current stage may become meaningless later. The spirit of primitive demand is to catch a rough estimation in a quick operation and our experiments have demonstrated that it is better than a more sophisticated measure at the early stage of global routing.

5.6 Conclusion

We have proposed two novel algorithms to timing-constrained global routing. We have formalized the routing tree topology flexibilities under timing constraints through the concepts of a soft edge and a slideable Steiner node, and have traded these flexibilities into congestion reduction while the timing constraints are satisfied. Experimental results show that the traditional rip-up-and-reroute method may cause significant delay violations and is poor on congestion when timing constraints are imposed directly. Our algorithm can achieve good congestion results while satisfying timing constraints. In addition to exploiting timing constrained routing flexibilities, we have applied a simple gradual refinement method based on probabilistic congestion estimation, which leads to simultaneous optimization on congestion, timing and the number of bends.

Chapter 6

Integrated Buffer and Wire Planning

6.1 Introduction

Early planning of buffers and wires is vitally important to achieve ambitious performance goals. It is generally accepted that buffer insertion has become a critical step in deep submicron design as interconnect now plays a dominating role in determining system performance. Current designs easily require thousands of nets to be buffered, and Cong [3] speculates that close to 800,000 buffers will be required for designs in 50 nanometer technology. Achieving timing closure becomes more difficult when buffer insertion is deferred to the back end of the design process, and the buffers must be squeezed into whatever left over space remains. The problem is particularly acute for custom designs, where large IP core macros and custom data flow structures are present, blocking out significant areas from buffering possibilities. ASIC designs can also run into similar headaches if they are dense, or have locally

dense hot spots. To manage the large number of buffers and also achieve high performance on the critical global nets, buffers must be planned for early in the design, so that the rest of the design flow is aware of the required buffering resources. In addition, design routability has also become a critical problem; one must make sure that an achievable routing solution exists during the physical floorplanning stage. Thus, global wiring must be planned early to minimize routing congestion, hot spots, and crosstalk problems later on in the flow.

6.1.1 Buffer Block Planning Methodology

In response to the need for an interconnect-centric design methodology, a new body of research on buffer block planning has recently established itself in the literature [66,68–71]. These works focus on physical-level interconnect planning, as described in [72]. The works of [66,68,71] all propose the creation of additional buffer blocks to be inserted into an existing floorplan. These buffer blocks are essentially top-level macro blocks containing only buffers. Cong *et al.* [68] proposed to construct these blocks using feasible regions. A feasible region is the largest polygon in which a buffer can be inserted for a particular net such that the net's timing constraint is satisfied. Sarkar *et al.* [66] added a notion of independence to the feasible regions in [68] while also trying to relieve routing congestion during optimization. Tang and Wong [71] proposed an optimal buffer block planning algorithm in terms of maximizing the number of inserted buffers (assuming that one buffer is sufficient for each net). Finally, Dragan *et al.* [69] presented a multi-commodity flow-based approach to buffering 2-pin nets assuming that a buffer block plan had already been created. This approach was extended to multi-pin nets in [70]. In the buffer block planning methodology, buffers are essentially

packed between larger existing floorplanned blocks. We argue there are two fundamental problems with the buffer block planning approach:

1. Since buffers are used to connect global wires, there will be considerable contention for routing resources in the regions between macro blocks. The design may not be routable due to heavy congestion between blocks.

2. Buffers must be placed in poor locations since better locations are blocked. Some blocks may even be so large that routing over the block is infeasible, even if buffers are inserted immediately before and after the block. For example, signal integrity could degrade beyond the point of recovery or wire delay may simply be too high. One may be able to alleviate the problem by using wider wires on thick metal, powering up to very large buffers, etc., but these solutions exacerbate the congestion problem.

The flaws are not with buffer block planning *per se*; rather, it is certainly a reasonable method for pre-planning buffers within current design flows. However, buffer block planning is really an interconnect-centric idea being applied to a device/logic-centric flow. Ultimately this methodology will not be sustainable as design complexity continues to increase. A different methodology is required. Ideally, buffers should be dispersed with some regularity throughout the design. Clumping buffers together, e.g., in buffer blocks, or between abutting macros invites routing headaches. A more uniform distribution of buffers will also naturally spread out global wires. There must be a way to allow buffers to be inserted inside blocks.

6.1.2 Buffer Site Methodology

We propose an alternative methodology. Macro block designers must allow global buffer and wiring resources to be interspersed within their designs wherever possible. This resource allocation need not be uniform; a block with a lower performance requirement and complexity may be able to afford to allocate a higher percentage of its resources. A cache or blocks within a datapath may not be able to allocate any resources. Ideally, as this *hole in a macro* methodology becomes widespread, even future IP blocks will have some of their area devoted to buffering resources.

To set aside a buffer resource within a block, the designer can insert what we call a buffer site, i.e., physical area which can denote either a buffer, inverter (with a range of power levels), or even a decoupling capacitor. When a buffer site gets assigned to a net, a logical gate from the technology is actually specified. Until this assignment takes place, buffer sites are not connected to any nets. Allocating a percentage of a macro block for buffer sites may be viewed as wasteful; however, if the sites are not used for buffering there are other ways to utilize them. For example, they can be populated with spare circuits to facilitate metal-only engineering changes late in the design cycle. Or, the sites can be populated with decoupling capacitors to enhance local power supply and signal stability. Thus, one can actually afford to allocate many more buffer sites than will ever be used. Buffer sites can also be a powerful tool for semi-custom designs. For example, in a data flow there are typically regular signal buses routed across collections of data flow elements. These routes are generally expected to be implemented with straight wires if possible. If buffering for some or all of the strands of a data bus are required, it is important to have buffer locations available within the data path itself. If buffer sites are designed into the original data path

layout, it is possible to add buffers late in the design cycle while maintaining straight wiring of the buses. Buffer sites can also be used for a flat design style, e.g., a sea of buffer sites can be sprinkled throughout the placement. For hierarchical designs, one can view the buffer sites as flat to derive a similar sprinkling, but their distribution will likely be less uniform. Some regions could have, say, 5-10% of the area devoted to buffer sites, while a region containing the cache will have none. No matter which design style is used, a resource allocation algorithm can view buffer sites as flat, which enables it to make assignments to global routes based on buffer site distribution.

6.1.3 Technical Contribution

We propose a new buffer and wire resource allocation formulation. Assuming that locations for buffer sites have already been chosen, the problem is to assign buffers to global nets such that each buffer corresponds to an existing buffer site. We model the problem with a tile graph to manage the complexity of thousands of buffer sites and to integrate wire congestion into the problem statement. We propose the following four stage heuristic:

1. Construct low-cost, low-radius Steiner trees for each net.

2. Rip-up and re-route nets to reduce wire congestion.

3. Insert buffers on all nets which require them. This stage is based on a Van Ginneken [21] style dynamic programming algorithm, yet we can find the optimal solution for a given net more efficiently than [21].

4. Rip-up, re-route, and re-insert buffers on nets to reduce both wire and buffer congestion.

Unlike the approaches in [68, 69, 66, 71] , our algorithm is designed to handle nets with multiple sinks (as is [70]).

6.2 Problem Formulation

There are two fundamental characteristics of buffer and wire planning which drive our formulation.

1. Finding the absolute optimal locations for a buffer is not particularly important. Cong *et al.* [68] showed that one may be able to move a buffer a considerable distance from its ideal location while incurring a fairly small delay penalty. Their concept of feasible regions for buffer insertion is based on the principle that there is a wide range of reasonably good buffer locations.

2. At the interconnect-centric floorplanning stage, timing constraints are generally not available since macro block designs are incomplete and global routing and extraction have not been performed. Potentially crude timing analysis could be performed, but the results are often grossly pessimistic because interconnect synthesis has not taken place. At this stage, one needs to globally insert buffers while tracking wire congestion before the floorplan can even be evaluated. For example, in a design with a desired 5 ns clock period, say that one floorplan has a worst slack of -40 ns while a different floorplan has a worst slack of -43 ns. The designer cannot determine which floorplan is better because the slacks for both are so absurdly far from their targets. Buffer and wire planning must be efficiently performed first, then the design can be timed to provide a meaningful worst slack timing that the designer

can use for evaluation. We envision performing buffer and wire planning each time the designer wants to evaluate a floorplan. The first characteristic suggests that one does not need to worry about exactly where buffer sites are placed. The block designers should have the freedom to sprinkle buffer sites into their designs so that performance is not compromised; there just needs to be a sufficient number of buffer sites somewhere.

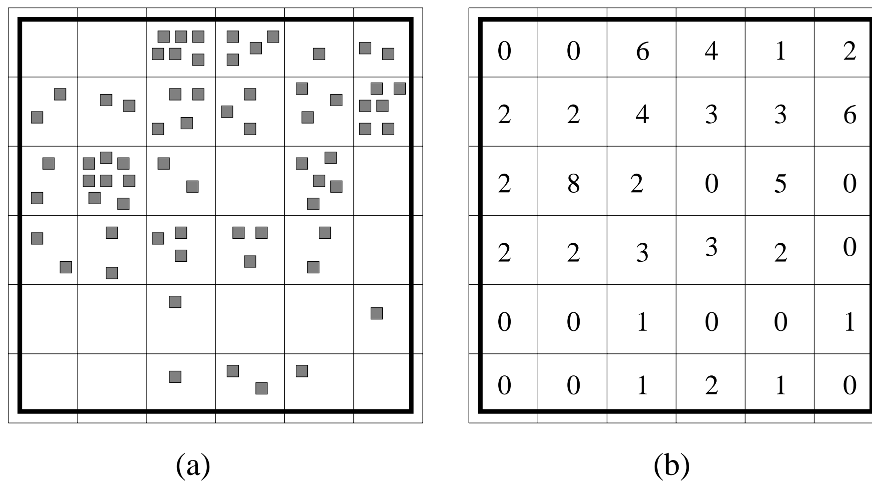


Figure 6.1: (a) A set of 68 buffer site locations can be tiled and (b) abstracted to a total number of buffer sites lying within each tile.

The optimization algorithm can view the thousands of buffer sites within a tile graph. Figure 6.1(a) shows 68 buffer sites lying within the region of the chip. A tiling over the chip's area can be used to abstract each individual buffer site to a set of buffer sites lying at the center of each tile (Figure 6.1(b)). The tile graph offers both a complexity reduction advantage (especially when there are thousands of buffer sites) and also the ability to manage routing congestion across tile boundaries. The granularity of the tiling depends on the desired accuracy/runtime trade-off and on the current stage in the design flow.

The second characteristic suggests that timing constraints are not reliable in the early floorplanning stage. Our formulation relies on a global rule of thumb for the maximum distance between consecutive buffers. This rule of thumb was also used for buffer planning by Dragan *et al.* [69]. They note that for a high-end microprocessor design in $0.25\mu m$ CMOS technology, repeaters are required at intervals of at most $4500\mu m$. Such a rule is necessary to ensure that the slew rate is sufficiently sharp at the input to all gates.

We represent a tiling by a graph $G(V_G, E_G)$ where $V_G = \{g_1, g_2, ...\}$ is the set of tiles and edge $\hat{e}_{ij}$ is in E_G if g_i and g_j are neighboring tiles. Given a tile g, let $s(g)$ be the number of buffer sites within the tile. Let $\mathcal{N} = \{N^1, N^2, ...\}$ be the set of global nets and let $s(\hat{e}_{ij})$ be the maximum permissible number of wires that can cross between g_i and g_j without causing overflow. If $d(g)$ denotes the number of buffers assigned in $g \in V_G$, the buffer congestion for g is given by $d(g)/s(g)$. Similarly, given a global routing of $\mathcal{N}$, if $d(\hat{e}_{ij})$ denotes the number of wires which cross between tiles g_i and g_j, the wire congestion for edge $\hat{e}_{ij}$ is given by $d(\hat{e}_{ij})/s(\hat{e}_{ij})$.

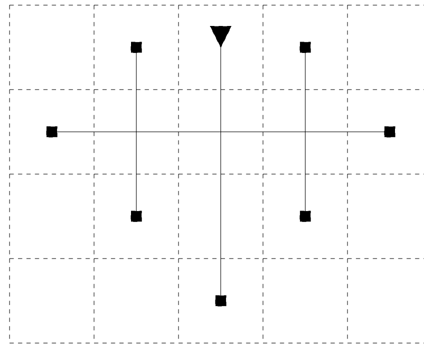


Figure 6.2: Driver with seven sinks, whereby the maximum distance allowed between gates is three. With this interpretation of the distance rule, the driving gate must drive 11 units of wirelength.

For net N^i, let L^i be the maximum wirelength, in units of tiles, that can be driven by either the driver of N^i or a buffer inserted on N^i. This interpretation of maximum distance avoids the scenario that could occur in Figure 6.2. The figure shows a driver with seven sinks whereby the distance between the driver and each sink is three tile units. Using this interpretation of the distance constraint results in a legal solution where the source gate drives 11 tile units of wirelength without requiring any buffers. For a slew-based distance rule, the extra interconnect (and sink load) will likely cause weak signals at the sinks. Thus, our distance rule requires that the total amount of interconnect that can be driven by any gate is no more than L^i.

Problem 6.1 *Given a tiling of the chip area $G(V_G, E_G)$, nets $\mathcal{N} = \{N^1, N^2, ...\}$, the number of buffer sites $s(g), g \in V_G$, and tile length constraints L^i, assign buffers to nets such that*

- *$d(g) \leq s(g) \forall g \in V_G$, where $d(g)$ is the number of buffers assigned to tile g.*
- *Each net $N^i \in \mathcal{N}$ satisfies its tile length constraint, L^i.*
- *There exists a routing after buffering such that for all $\hat{e} \in E_G$, the number of wires crossing from g_i to g_j is less than or equal to $s(\hat{e}_{ij})$.*

A solution to this problem means that constraints are satisfied, though secondary objectives can also be optimized, such as total wirelength, maximum and average wire congestion, maximum and average buffer congestion, and net delays. Our heuristic seeks to find a solution which satisfies the problem formulation while also trying to minimize these secondary objectives.

Note that the purpose of our formulation should not be used to find the final buffering and routing of the design. Rather, it can be used to estimate

needed buffering and routing resources or as a precursor to timing analysis for more accurate floorplan evaluation. Once deeper into the physical design flows, nets which generate suboptimal performance or are in timing-critical paths should be re-optimized using more accurate values of timing constraints and wiring capacitances.

6.3 Buffer and Wire Planning Heuristic

The purpose of our proposed heuristic is to show how buffer and wire planning can be integrated into a tile-based global routing methodology. We follow a traditional rip-up and re-route type of strategy. Our heuristic proceeds in four stages: (i) initial Steiner tree construction, (ii) wire congestion reduction, (iii) buffer allocation, and (iv) final post processing. The primary innovations are within stages 3 and 4 which handle buffer site assignment. Stages 1 and 2 deliver an initial congestion-aware global routing solution as a starting point. One could alternatively begin with the solution from any global router, e.g., the multi-commodity flow-based approach of [53].

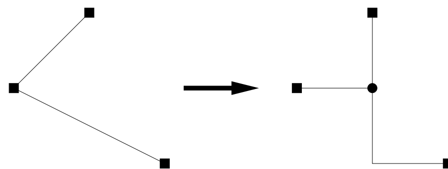


Figure 6.3: Example of spanning tree edge overlap removal.

6.3.1 Stage 1: Initial Steiner Tree Construction

At this stage, we want an initial routing of each net so that congested regions can be evaluated and reduced in each stage. As opposed to a pure minimum length construction, the tree construction needs to be timing-driven, yet timing constraints are not necessarily available. Hence, we adopt the Prim-Dijkstra construction [73] which generates a hybrid between a minimum spanning tree and shortest path tree. The result is a spanning tree which trades off between radius and wirelength. The spanning tree is then converted to a Steiner tree via a greedy overlap removal algorithm. The algorithm iteratively searches for the two tree edges with the largest potential wirelength overlap. A Steiner point is introduced to remove the overlap as shown in Figure 6.3. The algorithm terminates when no further overlap removal is possible.

6.3.2 Stage 2: Wire Congestion Reduction

The next step is to rip-up-and-reroute to reduce wire congestion. The tile graph $G(V_G, E_G)$ is constructed from the existing Steiner routes, and the congestion of each edge in E_G is computed. Instead of ripping up nets in congested regions, we rip-up and re-route every net, similar in spirit to Nair's method [55]. This approach is less likely to become trapped in a local minima. The net ordering is first fixed (we sort in order of smallest to largest delays), and each net is processed in turn according to the ordering. The advantage is that even nets which do not violate congestion constraints can be improved to further reduce congestion so that other nets can be successfully re-routed in subsequent iterations. The algorithm terminates after either three complete iterations or $d(\hat{e})/s(\hat{e}) \leq 1$ for all $\hat{e} \in E_G$. From experience, only nominal

potential improvement exists after the third iteration.

To re-route the net, the entire net is deleted and then re-routed using an approach similar to [46], as opposed to re-routing one edge. The new tree is constructed on the tile graph using the same Prim-Dijkstra cost function in Stage 1, except that the cost for each edge is not its Manhattan distance. The routing occurs across the tile graph using the following congestion-based cost function:

$$cost(\hat{e}) = \begin{cases} \frac{d(\hat{e})+1}{s(\hat{e})-d(\hat{e})} & : \quad d(\hat{e}) < s(\hat{e}) \\ \infty & : \quad d(\hat{e}) \geq s(\hat{e}) \end{cases} \tag{6.1}$$

The cost is the number of wires that will be crossing divided by the number of wires still available. The purpose of this cost is to have the penalty become increasingly high as the edge comes closer to full capacity. The procedure performs a wave-front expansion from the tile which contains the source, updating to the lowest tile cost with each expansion. When each sink in the net is reached, the algorithm terminates, and the tree is recovered by tracing back the edges to the source from each sink.

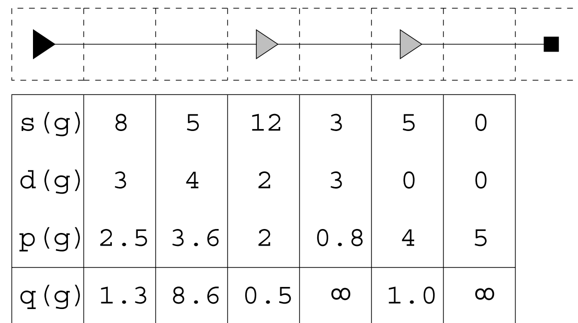
6.3.3 Stage 3: Buffer Allocation

Once a low congestion routing exists, the next step assigns buffer sites to each net. We perform this allocation iteratively in order of net delay, starting with the net with highest delay. Before buffers are allocated, we first estimate the probability of a net occupying a buffer site in a tile. For a net N^i passing through tile g, the probability of a buffer from g being inserted onto N^i is defined as $1/L^i$. Let $p(g)$ be the sum of these probabilities for tile g over all unprocessed nets. Recall that $s(g)$ is the number of buffer sites in and $d(g)$

is the current number of used buffer sites. We define the cost $q(g)$ for using a particular buffer site as

$$cost(g) = \begin{cases} \frac{d(g)+p(g)+1}{s(g)-d(g)} & : \quad d(g) < s(g) \\ \infty & : \quad d(g) \geq s(g) \end{cases} \tag{6.2}$$

Observe the similarity between Equations 6.2 and 6.1. Both significantly increase the penalty as resources become more contentious.



s(g)	8	5	12	3	5	0
d(g)	3	4	2	3	0	0
p(g)	2.5	3.6	2	0.8	4	5
q(g)	1.3	8.6	0.5	∞	1.0	∞

Figure 6.4: Example of how buffer costs are computed. For a value of $L^i = 3$, the optimal solution is shown, having total cost 1.5.

Figure 6.4 shows an example of how the buffer cost is computed. Note that the $p(g)$ values do not include the currently processed net. The cost $q(g)$ is computed for each tile, and $q(g)$ is included in the cost for a net if a buffer is inserted at g. In the example, if $L^i = 3$, the minimum cost solution has buffers in the third and fifth tiles, with cost $0.5 + 1.0 = 1.5$.

An optimal solution can be found in linear time in terms of the number of tiles spanned by the net (assuming that L is constant). The approach uses a Van Ginneken [21] style dynamic programming algorithm, but has lower time complexity because the number of candidates for each node is at most L.

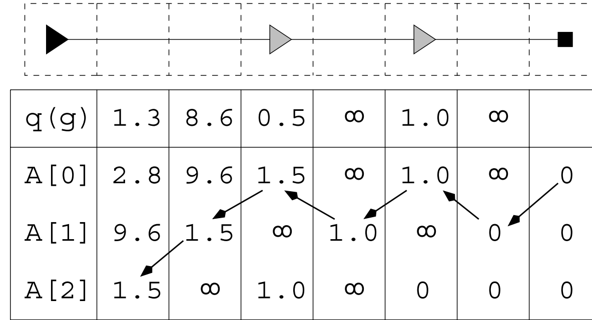
We begin with the simple case, a net N with a single source v_0 and sink v_t. Let $\pi(g)$ be the parent tile of tile g in the routing path, and assume that $q(g)$ has been computed for all tiles on the path between v_0 and v_t. At each tile g, the array A_g stores the cost of the solutions from g to v_t. The index of the array determines the distance downstream from g to the last buffer inserted. Thus, the array is indexed from 0 to $L-1$, since g cannot be at distance more than L from the last available buffer. The full algorithm is shown in Figure 6.5.

Single-Sink Buffer Sites Allocation
1. $g \leftarrow$ tile where sink v_t locates in
Set $A_g[i] = 0$ for $0 \leq i < L$
2. While v_0 not in g
For $i = 1$ to $L-1$
$A_{\pi(g)}[i] \leftarrow A_g[i-1]$
$A_{\pi(g)}[0] \leftarrow q(\pi(g)) + \min\{A_g[j] \vert 0 \leq j < L\}$
$g \leftarrow \pi(g)$
3. Let g be such that v_0 locates in $\pi(g)$
Return $\min\{A_g[j] \vert 0 \leq j < L\}$

Figure 6.5: Single-sink buffer sites allocation algorithm

The first step is to initialize the cost array A_g to zero for the tile g where sink v_t locates in. The algorithm then traverses up towards the source, iteratively setting the values for the cost array. Step 2 computes the values for $\pi(g)$ given the values for g. The value of $A_{\pi(g)}[i]$ for $i > 0$ is simply $A_g[i-1]$ since no buffer is being inserted at $\pi(g)$ for this case. If a buffer is to be inserted at $\pi(g)$, then the cost $A_{\pi(g)}[0]$ is computed by adding the

current cost for insertion, $q(\pi(g))$, to the lowest cost seen at g. One can recover the solution by storing at $\pi(g)$ the index in A_g which was used to generate the solution.



q(g)	1.3	8.6	0.5	∞	1.0	∞	
A[0]	2.8	9.6	1.5	∞	1.0	∞	0
A[1]	9.6	1.5	∞	1.0	∞	0	0
A[2]	1.5	∞	1.0	∞	0	0	0

Figure 6.6: Execution of the single source algorithm on the example in Figure 5. The optimal solution has cost 1.5 and the arrows show how this cost is obtained.

Figure 6.6 shows how the cost array is computed for the 2-pin example in Figure 6.4 (with $L = 3$), and the arrows show how to trace back the solution. Observe from the table that costs are shifted down and to the left as one moves from right to left, with the exception of entries with index zero.

The algorithm is optimal since each possible solution is preserved during the execution. One can take advantage of the fact that the number of possible candidates at each node is no more than L to give a space and time complexity of $O(nL)$, where n is the number of tiles spanned by the net. This is a significant advantage over similar dynamic programming approaches [21, 27, 44] which have at least $O(n^2)$ time complexity.

Extending the algorithm to multi-sink routing trees is fairly straightforward. One still keeps a cost array at each node, but updating the cost becomes a bit trickier when a node in a routing tree has two children. Let

$l(g)$ and $r(g)$ denote the two children of g in the routing tree. If g has only one child, let it be $l(g)$. When considering buffer insertion at a node with two children, there are three cases as shown in Figure 6.8. A buffer may be used to either (a) drive both branches, (b) decouple the left branch, or (c) decouple the right branch.

Multi-Sink Buffer Sites Allocation

1. $v \leftarrow$ an unvisited node whose descendents have been visited
   $g \leftarrow$ tile v locate
   While no visited node in g
2. If a leaf node v in g, $A_g[i] \leftarrow 0$ for $0 \leq i < L$
3. If g has one child $l(g)$ in routing tree
   For $i = 1$ to $L - 1$
   $A_g[i] \leftarrow A_{l(g)}[i-1]$
   $A_g[0] \leftarrow q(g) + \min\{A_{l(g)}[j] | 0 \leq j < L\}$
4. If g has two children $l(g)$ and $r(g)$

4.1 For $i = 2$ to to $L - 1$
   $A_g[i] \leftarrow \min\{A_{l(g)}[i_l] + A_{r(g)}[i_r] | i_l + i_r + 2 = i\}$

4.2 $A_g[0] \leftarrow q(g) + \min\{A_{l(g)}[i_l] + A_{r(g)}[i_r] | i_l + i_r + 2 \leq L\}$

4.3 $A_g[1] \leftarrow \infty$

4.4 For $i = 1$ to $L - 1$
   $A_g[i] \leftarrow \min\{A_g[i], q(g) + A_{l(g)}[i-1], q(g) + A_{r(g)}[i-1]\}$

5. Mark v as visited, $g \leftarrow \pi(g)$
6. Let g be tile that v_0 locates in, return $\min\{A_g[j] | 0 \leq j < L\}$

Figure 6.7: Multi-sink buffer sites allocation algorithm

The complete algorithm is shown in Figure 6.7. The algorithm flows from the sinks to the source in the same manner as the single-sink algorithm in Figure 6.5, except for the inclusion of Step 4. Step 4.1 handles the case where no buffer is inserted at the branch node g. A distance of one is driven

for both the left and right branches, hence no buffer implies that the cost array is updated only for indices 2 and above. Step 4.2 handles the case where a buffer is driving both children, taking the combined minimum cost left and right branches. Step 4.3 initializes the cost array for index 1 since it has not yet been set. Finally, Step 4.4 updates the cost array with a better solution from potentially decoupling either of the two branches. The multi-sink variation has $O(nL^2)$ time complexity due to step 4.2.

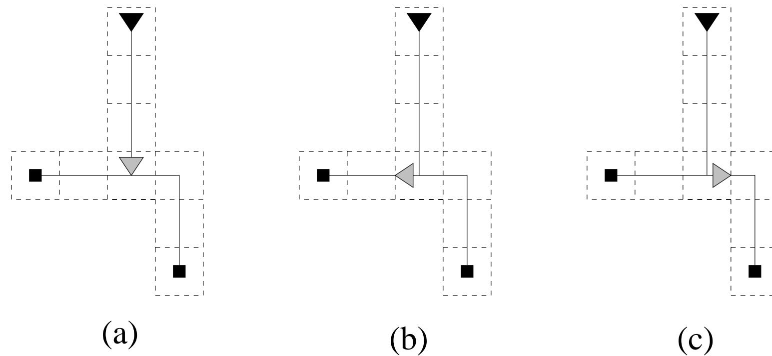


Figure 6.8: For a node with two children, a buffer may be used to either (a) drive both branches, (b) decouple the left branch, or (c) decouple the right branch.

6.3.4 Stage 4: Final Post Processing

The final stage of our algorithm attempts to reduce buffer congestion, wire congestion, and the number of nets which, up until now, have failed to meet their length constraint. Using the same flow as in stage 2, each net is ripped up and re-routed, and the buffers for the net are removed as well. However, for multi-pin nets, the net is ripped up one two-path at a time, where a two-path is a path in the tree which begins and ends at either a Steiner node, source, or sink and contains only vertices of degree two. The two ends of the

two-path are then reconnected via the path that minimizes the sum of wire and buffer congestion costs (Equations 6.1 and 6.2).

The minimum cost two-path is computed as follows. Call the endpoint of the original two-path that is in the same sub-tree as the source the head and the other endpoint the tail. The algorithm works in bottom-up fashion in a manner similar to the single-sink buffer insertion algorithm (and also the buffer insertion, maze-routing algorithms of [27, 44]). Starting from the tail, the algorithm visits the neighbors of the current minimum cost tile and updates the cost array. For each element in the cost array, a pointer is maintained back to the tile which was used to generate that cost. The algorithm iteratively expands the tile with lowest cost and updates the costs of neighboring tiles during wavefront expansion. The cost for the new tile also includes the wire congestion cost of crossing the tile boundary. Finally, when the head of the 2-path is reached, the minimum cost solution is recovered by tracing back out the path to the tail; the buffers used to derive this solution are also inserted during the trace.

6.4 Experimental Results

We implemented our heuristic in C++ on an RS6000/595 machine with 1 Gb of memory. We tested our code on ten benchmarks which we obtained from the authors of [68]. The first six are from the Collaborative Benchmarking Laboratory (CBL) and the other four are randomly generated. We also embed the designs in the same $0.18\mu m$ technology used in [68]. The statistics of the benchmarks are summarized in the first five columns of Table 6.1. The nets and sinks columns present slightly smaller values than in [68] to reflect the nets that Cong *et al.* did not optimize since they did not require buffers.

Table 6.1: Test circuit statistics and parameters for the first set of experiments.

circuit	# mod	# nets	# pads	# sinks	grid	L^i	# buf sites	$\%A_{buf}$	A_{tile}
apte	9	87	73	154	30 × 33	6	1200	0.13	0.38
xerox	10	178	2	375	30 × 30	5	3000	0.39	0.35
hp	11	70	45	189	30 × 30	6	2350	0.22	0.48
ami33	33	118	43	344	33 × 30	5	2750	0.24	0.46
ami49	49	351	22	466	30 × 30	5	11450	0.75	0.67
playout	62	1517	192	1970	33 × 30	5	27550	1.52	0.73
ac3	27	209	75	407	30 × 30	8	3550	0.32	0.49
xc5	50	857	2	1801	30 × 30	6	13550	1.11	0.53
hc7	77	359	51	1171	30 × 30	6	7780	0.33	1.04
a9c3	147	1138	22	1518	30 × 30	6	12780	0.52	1.08

6.4.1 General Performance

Our first set of experiments studies the performance of each of the four stages of our heuristic. The grid size and number of buffer sites used are presented in Table 6.1. We chose the grid size to have 30 tiles on the shorter side of the chip, then derived the number of tiles for the longest side, so that each tile was roughly square. The total number of buffer sites for each circuit were also chosen arbitrarily. The number was chosen to be large enough to so that buffer congestion is low, but small enough so that the percent of the total chip area occupied by buffer sites is less than 2% of the total chip area. The total buffer site area A_{buf} in percentage of chip area is given in column 9, we can observe that the percentage of the chip area is less than 1% for all but two cases. The tile area A_{tile} in mm^2 is shown in the rightmost column. Except for the last two random circuits, no tile is more than one millimeter long on a side. For each benchmark, a random nine by nine set of tiles were prohibited from having any inserted buffer sites to correspond to a large cache-like object. The buffer sites were randomly distributed among

the remaining tiles.

The floorplans were generated using the buffer block planning code supplied by the authors of [68]. The results for each stage and each CBL benchmark are summarized in Table 6.2. The statistics presented are:

- the maximum and average wire congestion over all edges in the tile graph,
- the sum of the wiring overflows $\mathcal{F}_{ov}$, i.e., the sum over all $\hat{e} \in E_G$ of $d(\hat{e}) - s(\hat{e})$, for whenever $d(\hat{e}) > s(\hat{e})$,
- the maximum and average buffer congestion,
- the number of buffers inserted,
- the number of nets for which the tile length constraint was not satisfied,
- total wirelength W in millimeters,
- maximum and average delays in picoseconds to each sink,
- and CPU time in seconds.

Note that no timing constraints are used, so we use average and maximum source-to-sink delays to give an indication for the quality of timing. We make several observations:

- The wire congestion constraint is always satisfied. If one ignores wire congestion, as is done in Stage 1, then the maximum wire congestion is typically a factor of two to three above capacity and there are several hundred overflows.

- The algorithm never violates the buffer site constraint, but typically utilizes at least one tile to full buffer capacity. As seen from the small buffer site area percentages in Table 1, the total number of buffer sites chosen is actually quite small relative to total area.

- The number of buffers, failures, and wirelength all decline from Stage 3 to Stage 4 (except for the wirelength for ami33), which shows that our final post-processing step is quite effective. The number of nets which fail to meet the length constraint is typically small, but not zero. This is caused almost exclusively by the existence of the large 9 by 9 tiled region with no buffer sites that was inserted into each design.

- Net delays increase significantly from Stage 1 to Stage 2 during re-routing to avoid congestion, but the insertion of buffers in Stage 3, reduces delay significantly even though the buffer insertion algorithm is *delay ignorant*. The maximum and average delay always is less after Stage 4 than Stage 1, with the exception of the maximum delay for playout.

- The CPU time is almost exclusively dominated by the two re-routing stages, 2 and 4. Thus, our buffer insertion algorithm in Stage 3 is indeed efficient in practice.

6.4.2 Variations

Our next experiments examines the behavior of our algorithm when the number of available buffer sites varies. We ran our algorithm on each of the six CBL circuits three times using small, medium, and large number of buffer sites that are randomly distributed (with the blocked 9 by 9 region). The

Table 6.2: Stage by stage experimental results for the 6 CBL circuits. The final results are shown for the last four random circuits.

		Wire Cngst			Buf Cngst					sink delay		
circuit	stage	max	ave	$\mathcal{F}_{ov}$	max	ave	#bufs	#fails	W	max	ave	CPU
apte	1	2.75	0.16	531	0.00	0.00	0	87	1519	5946	1687	0
	2	1.00	0.21	0	0.00	0.00	0	87	2036	7032	2462	12
	3	1.00	0.21	0	1.00	0.37	436	6	2036	5383	1079	0
	4	1.00	0.20	0	1.00	0.36	406	4	1974	2330	879	30
xerox	1	3.20	0.20	705	0.00	0.00	0	178	3101	4234	1600	1
	2	0.73	0.25	0	0.00	0.00	0	178	3875	5193	1920	41
	3	0.73	0.25	0	1.00	0.39	1140	17	3875	3492	1175	1
	4	0.73	0.23	0	1.00	0.35	967	9	3729	2564	859	61
hp	1	3.00	0.32	474	0.00	0.00	0	70	1563	8462	2869	0
	2	1.00	0.43	0	0.00	0.00	0	70	2133	9020	3570	8
	3	1.00	0.43	0	1.00	0.19	497	9	2133	4300	1035	0
	4	1.00	0.40	0	1.00	0.16	379	9	1992	3893	1026	29
ami33	1	3.33	0.34	532	0.00	0.00	0	118	2657	11179	5178	0
	2	0.83	0.41	0	0.00	0.00	0	118	3268	15476	6437	32
	3	0.83	0.41	0	1.00	0.31	865	7	3268	6031	1332	1
	4	1.00	0.40	0	1.00	0.31	765	3	3313	2906	1286	35
ami49	1	2.73	0.42	1236	0.00	0.00	0	351	6625	7644	2085	1
	2	1.00	0.58	0	0.00	0.00	0	351	9366	30614	3567	44
	3	1.00	0.58	0	0.55	0.17	1991	18	9366	12381	1300	0
	4	1.00	0.49	0	0.88	0.14	1475	13	7876	3007	981	81
playout	1	1.60	0.15	1032	0.00	0.00	0	1517	30323	8637	2395	0
	2	0.93	0.20	0	0.00	0.00	0	1517	39903	24117	3477	268
	3	0.93	0.20	0	0.83	0.28	7631	172	39903	18608	1399	4
	4	0.93	0.18	0	1.00	0.24	6378	73	36100	3144	1064	363
ac3	1-4	0.75	0.31	0	1.00	0.19	632	26	4726	6109	1005	103
xc5	1-4	0.87	0.41	0	1.00	0.24	3021	36	17997	5681	1214	331
hc7	1-4	1.00	0.57	0	1.00	0.27	2006	65	15885	8520	1477	209
a9c3	1-4	1.00	0.59	0	1.00	0.31	3796	48	31287	14911	1374	401

other parameters are the same as in the previous set of experiments. Results are summarized in Table 6.3. The three lines for each circuit correspond to small, medium, and large numbers of buffer sites.

Observe that when the number of buffer sites is small, several nets fail to meet their length constraint. Also, sometimes wire congestion constraints cannot be met, e.g., for ami33. Most notably, as the number of buffer sites increases, the maximum and average net delays decrease significantly. Having no more than one in every five buffer sites occupied is necessary to obtain good solutions.

Table 6.3: Summary of results when the number of available buffer sites varies.

circuit	buf sites	Wire Cngst max	Wire Cngst ave	$\mathcal{F}_{ov}$	Buf Cngst max	Buf Cngst ave	#bufs	#fails	W	sink delay max	sink delay ave	CPU
apte	280	1.00	0.21	0	1.00	0.79	218	36	2018	5544	1620	51
	700	1.00	0.21	0	1.00	0.57	379	9	2155	5876	1308	59
	3200	1.00	0.20	0	1.00	0.13	405	6	1923	1899	832	67
xerox	600	0.93	0.25	0	1.00	0.89	530	95	3997	8346	1509	132
	1300	0.87	0.25	0	1.00	0.77	962	25	4094	4551	1257	228
	3000	0.73	0.23	0	1.00	0.35	967	9	3729	2564	859	163
hp	300	1.00	0.44	0	1.00	0.73	214	27	2192	7849	2373	35
	600	1.00	0.43	0	1.00	0.60	346	13	2137	6007	1561	66
	2350	1.00	0.40	0	1.00	0.16	379	9	1992	3893	1026	58
ami33	500	1.50	0.41	4	1.00	0.78	385	50	3303	15327	6029	92
	850	1.17	0.43	1	1.00	0.74	613	24	3441	9782	3143	91
	2750	1.00	0.40	0	1.00	0.31	765	3	3313	2906	1286	107
ami49	850	1.27	0.48	42	1.00	0.78	656	221	7892	6519	1589	174
	1650	1.18	0.53	12	1.00	0.80	1279	128	8572	5302	1278	190
	11450	1.00	0.49	0	0.88	0.14	1475	13	7876	3007	981	195
playout	3250	0.93	0.18	0	1.00	0.93	2994	1093	37141	13191	1770	854
	6250	0.94	0.19	0	1.00	0.85	5253	612	39349	13191	1434	876
	27550	0.93	0.18	0	1.00	0.24	6378	73	36100	3144	1064	998

In our next set of experiments, we keep the number of buffer sites constant, but vary the size of the grid. The results are summarized in Table 6.4 for a sampling of the CBL circuits. Observe that the maximum wire congestion increases with the tile size. A finer-grained tiling implies a tighter wire congestion, e.g., dividing a tile into four equal sized tiles increases the number of congestion constraints by a factor of three. The increased wire congestion may cause an increase in the maximum delay because of long detours, though the average congestion can stay the same.

The finer-grained tile sizes give better insight into the quality of the floorplan. A coarser tiling can indicate that the design is fairly easily routable, but a finer tiling can better highlight areas of potential congestion. Thus, if one wants to use our algorithm to evaluate the quality of a particular floorplan, a finer-grained tiling is likely more useful for wire congestion evaluation. Finally, we observe that the CPU times roughly increase at a rate slightly

higher than linear with the number of tiles.

Table 6.4: Experimental results with varying grid sizes for three CBL benchmarks.

circuit	grid	Wire Cngst		Buf Cngst		#bufs	#fails	W	sink delay		CPU
		max	ave	max	ave				max	ave	
apte	10 × 11	0.44	0.19	1.00	0.25	283	4	1859	2219	933	4
	20 × 22	0.62	0.18	1.00	0.35	372	9	1918	2385	876	18
	30 × 33	0.89	0.18	1.00	0.45	493	17	2055	3050	881	40
	40 × 44	1.00	0.15	1.00	0.39	445	18	1951	2771	912	71
	50 × 55	1.00	0.19	1.00	0.39	450	25	1930	4265	1011	121
ami49	10 × 10	0.75	0.39	0.18	0.07	843	1	7610	3477	1113	11
	20 × 20	0.90	0.39	0.71	0.10	1124	3	7767	3185	1039	53
	30 × 30	0.92	0.40	1.00	0.13	1451	4	7801	2644	972	126
	40 × 40	1.00	0.39	1.00	0.14	1431	3	7741	3375	1024	250
	50 × 50	1.00	0.39	1.00	0.14	1464	7	7791	3309	973	457
playout	11 × 10	0.31	0.12	0.31	0.13	3704	5	34983	5369	1200	54
	22 × 20	0.48	0.13	0.63	0.18	4969	11	36551	3653	1136	256
	33 × 30	0.70	0.13	1.00	0.23	6387	69	36614	4529	1128	624
	44 × 40	0.77	0.13	1.00	0.23	6178	103	35257	3289	1045	1244
	55 × 50	0.96	0.13	0.88	0.24	6304	44	36080	11150	1140	2197

6.4.3 Comparisons with Buffer Block Planning

Our final experiments attempt to compare with previous work on buffer block planning, yet we are not using buffer blocks. Hence, it is not feasible to simply compare with previously published results. We need to run the code ourselves and implement routines to gather statistics from the data. For this comparison, Cong *et al.* [68] supplied us with the source code to their algorithm BBP/FR. Although other buffer block planning results exists (e.g., the work of [66] creates more buffer blocks to reduce wire congestion), we believe this comparison is sufficient to show that our proposed methodology can deliver timing solutions that are competitive with the buffer block planning methodology while better managing buffer and wire congestion.

As in [68], but unlike the previous experiments, we decomposed multi-pin net into several 2-pin nets. Our results were generated using randomly distributed buffer sites that altogether occupy less than 2% of the total chip area. Cong *et al.* [68] report timing results by measuring the number of nets which fail to meet their delay constraints. The timing constraint was chosen to be between 1.05 and 1.20 of the optimum achievable delay. We believe this constraint generation to be unrealistic because real timing constraints are path-based, and this implies that all constraints are tight, yet potentially satisfiable. In a practical situation, some of the $1.05\times - 1.20\times$ timing constraints will be so tight that buffer insertion is insufficient to satisfy timing. For these cases, feasible regions are not well defined. Also, other constraints are so loose that no buffer insertion is required or many detours can be taken to still meet delay constraints. Both our approach and BBP/FR insert buffers on all nets which require them, so we use maximum and average sink delay to quantify timing performance.

In addition to the previous statistics, we also measured MTP (maximum tile area percentage) and minimum wirelength. A percentage of the area of each tile is occupied by buffers; MTP denotes the maximum such percentage over all tiles. We used the same tiling for our algorithm (see Table 6.1) to measure the MTP for BBP/FR. The minimum wirelength is the sum of the minimum possible wirelength routing of all the nets. This enables one to see how much additional wirelength above the minimum was actually required.

Table 6.5 presents the comparisons with BBP/FR. Observe that sometimes the reported wirelength for BBP/FR is significantly higher than the minimum wirelength (11% on average), e.g., for xerox it is 5623 as opposed to a minimum of 4497. We believe that BBP/FR is not performing properly in these cases. Nevertheless, some differences between the approaches are

Table 6.5: Comparisons of our algorithm to BBP/FR.

circuit	algo.	Wire Cngst max	Wire Cngst ave	$\mathcal{F}_{ov}$	#bufs	MTP	W_{min}	W	sink delay max	sink delay ave	CPU
apte	BBP	2.00	0.15	73	224	2.0	2014	2259	4407	820	0
	Ours	0.92	0.14	0	525	0.3		2278	1817	697	49
xerox	BBP	1.50	0.14	20	565	5.1	4497	5623	3545	853	0
	Ours	0.72	0.12	0	1389	0.7		4878	1465	611	78
hp	BBP	2.33	0.20	92	274	1.6	2749	2869	2372	886	0
	Ours	0.75	0.21	0	631	0.2		3106	1613	791	40
ami33	BBP	1.28	0.14	62	723	4.4	5501	5867	6103	928	0
	Ours	0.69	0.14	0	1452	0.4		5978	1918	832	78
ami49	BBP	4.36	0.40	1067	987	4.3	7564	7967	7230	906	0
	Ours	0.93	0.41	0	1720	0.5		8500	2544	881	93
playout	BBP	1.27	0.18	192	4452	13.4	35275	35944	2864	972	0
	Ours	0.94	0.19	0	7367	1.1		37747	2868	928	515
ac3	BBP	1.81	0.19	152	757	3.7	5757	6187	3970	800	0
	Ours	0.58	0.19	0	1232	0.5		6243	1735	734	92
xc5	BBP	4.40	0.48	3880	3403	18.2	23684	30413	5188	973	0
	Ours	0.82	0.39	0	4763	1.1		25430	2087	680	325
hc7	BBP	3.50	0.49	1708	3051	5.7	20744	24614	18559	1259	0
	Ours	0.86	0.45	0	2810	0.4		22750	3023	1037	265
a9c3	BBP	2.67	0.34	1829	4335	5.6	31001	31882	14726	1110	1
	Ours	0.63	0.34	0	4118	0.4		32915	2687	1162	342

readily seen:

- Our algorithm is always able to meet the congestion criteria while BBP/FR does not. The results presented here even include a post-processing step which tries to minimize congestion for the current buffering solution without increasing wirelength.

- Our algorithm inserts significantly more buffers, due to a tight length constraint.

- Because our methodology invites spreading, the MTP is significantly less. In the worse case, BBP/FR has one tile with 18.2% of its area

devoted to buffers. This percentage climbs to at most 1.1% for our approach.

- The CPU time for BBP/FR is negligible. Stages 2 and 4 of our algorithm cause much greater runtimes, but they are clearly not prohibitive.
- The delays for our approach are less, though the gap is likely larger than it would be if BBP/FR was performing properly. Ultimately, we believe that our delay results will still prove competitive even though our objective function is length based.

6.5 Conclusion

We have proposed an alternative methodology for buffer and wire planning that uses pre-allocated buffer sites that are distributed throughout the design. This methodology enables one to model this planning problem via a tile graph and simultaneously plan both wires and buffers. Our four stage heuristic includes an efficient algorithm for length-based buffer insertion and also a technique for simultaneous optimization of buffer and wire congestion. Our experimental results assert that this approach can generate effective solutions in a reasonable amount of time.

Chapter 7

Conclusions

This thesis has developed novel methods for building high-performance interconnects. We have used the high-fidelity Elmore delay to predict the properties of the problem, and have then performed an optimization using accurate AWE-based timing metrics to guarantee the timing correctness of the routed net. We have developed a framework for global routing, starting with considering routing problems for a single net, and culminating in simultaneous routing of many global nets under constraints on the routing congestion and timing. A methodology for early interconnect and buffer planning has also been suggested.

We have integrated buffer insertion and driver sizing separately with the routing problem for a single net. We have departed from the traditional Hanan grid based optimization, and have proposed three algorithms: (1) BINO: We have considered the realistic situation where buffer locations are restricted to a limited set of spaces and simultaneously optimize the net topology and the buffer locations to meet timing constraints. (2) FAR-DS: We have used the convexity/concavity property of the sink delay to search

for the optimal net topology and driver size. (3) BBB: We have developed techniques to route a multi-pin net to avoid buffer blockages and seek buffer bays without large wiring detours. Experiments on FAR-DS and BINO have shown about 20% cost reductions compared to competing approaches, while guaranteeing that the net satisfies timing constraints. BBB, when applied on industry designs, reduces the average in-blockage wirelength by 33% with only 3% increase on total wirelength, allowing solutions with significantly improved timing and slew performance.

Two novel approach to simultaneous routing for a large number of global nets have been proposed to optimize both congestion and delay. Starting with an initial solution using the above methods, we have exploited routing flexibilities to improve the solution. The first algorithm, which is hierarchical and network flow based, successfully achieves timing-constrained congestion reduction and shows results that are superior to the traditional rip-up-and-reroute method. The second algorithm controls the number of bends on wires through probability-based gradual refinement in addition to meeting the congestion and delay constraints.

Finally, an integrated buffer and wire planning scheme has been developed. It provides an alternative approach to buffer sites planning to fit a more realistic design environment than buffer block method. A four-stage heuristic has been designed to minimize both wire and buffer congestion for multi-pin nets directly so that the quality of a floorplan can be evaluated according to interconnect effects. Experiments on both benchmark and randomly generated circuits validate the effectiveness of this method.

Publications

- H. Hou, J. Hu and S. S. Sapatnekar, "Non-Hanan routing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 18, N o. 4, pp. 436-444, April, 1999.

- J. Hu and S. S. Sapatnekar, "Simultaneous buffer insertion and non-Hanan optimization for VLSI interconnect under a higher order AWE model." *Proceedings of the ACM International Symposium on Physical Design*, pp. 133-138, 1999.

- J. Hu and S. S. Sapatnekar, "FAR-DS: Full-plane AWE routing with driver sizing," *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 84- 89, 1999.

- J. Hu and S. S. Sapatnekar, "Algorithms for non-Hanan-based optimization for VLSI interconnect under a higher order AWE model," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 19, No. 4, pp. 446-458, April 2000.

- C. J. Alpert, G. Gandham, J. Hu, J. L. Neves, S. T. Quay and S. S. Sapatnekar, "A Steiner tree construction for buffers, blockages, and bays," accepted by the *IEEE Transactions on Computer-Aided Design of Integrated Circuits and System.*

- J. Hu and S. S. Sapatnekar, "A timing-constrained algorithm for simultaneous global routing of multiple nets," *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 99-103, 2000.

- J. Hu and S. S. Sapatnekar, "Performance driven global routing through gradual refinement," submitted to *ACM/IEEE Design Automation Conference*, 2001.

- C. J. Alpert, J. Hu, S. S. Sapatnekar and P. G. Villarrubia, “A practical methodology for early buffer and wire resource allocation,” submitted to *ACM/IEEE Design Automation Conference*, 2001.

Bibliography

[1] G. E. Moore, “Cramming more components onto integrated circuits,” *Electronic Magazine*, vol. 38, pp. 114–117, Apr. 1965.

[2] “National technology roadmap for semiconductors.” Semiconductor Industry Association, 1997.

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

[4] M. Hanan, “On Steiner’s problem with rectilinear distance,” *SIAM Journal on Applied Mathematics*, vol. 14, no. 2, pp. 255–265, 1966.

[5] W. C. Elmore, “The transient response of damped linear networks with particular regard to wideband amplifiers,” *Journal of Applied Physics*, vol. 19, pp. 55–63, Jan. 1948.

[6] K. D. Boese, A. B. Kahng, B. A. McCoy, and G. Robins, “Near-optimal critical sink routing tree constructions,” *IEEE Transactions on Computer-Aided Design*, vol. 14, pp. 1417–36, Dec. 1995.

[7] J. Qian, S. Pullela, and L. T. Pillage, “Modeling the effective capacitance for the RC interconnect of CMOS gates,” *IEEE Transactions on Computer-Aided Design*, vol. 13, pp. 1526–1535, Dec. 1994.

[8] L. T. Pillage and R. A. Rohrer, "Asymptotic waveform evaluation for timing analysis," *IEEE Transactions on Computer-Aided Design*, vol. 9, pp. 352–366, Apr. 1990.

[9] A. B. Kahng and G. Robins, *On optimal interconnections for VLSI*. Boston, MA: Kluwer Academic Publishers, 1995.

[10] H. Hou and S. S. Sapatnekar, "Routing tree topology construction to meet interconnect timing constraints," in *Proceedings of the ACM International Symposium on Physical Design*, pp. 205–210, 1998.

[11] H. Hou, J. Hu, and S. S. Sapatnekar, "Non-Hanan routing," *IEEE Transactions on Computer-Aided Design*, vol. 18, pp. 436–444, Apr. 1999.

[12] J. Hu and S. S. Sapatnekar, "Algorithms for non-Hanan-based optimization for VLSI interconnect under a higher order AWE model," *IEEE Transactions on Computer-Aided Design*, vol. 19, pp. 446–458, Apr. 2000.

[13] J. Lillis and P. Buch, "Table-lookup methods for improved performance-driven routing," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 368–373, 1998.

[14] J. Lillis, C. K. Cheng, T. T. Lin, and C. Y. Ho, "New performance driven routing techniques with explicit area/delay tradeoff and simultaneous wire sizing," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 395–400, 1996.

[15] A. Vittal and M. Marek-Sadowska, "Minimum delay interconnect design using alphabetic trees," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 392–396, 1994.

[16] J. Cong and C. K. Koh, "Interconnect layout optimization under higher-order RLC model," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 713–720, 1997.

[17] F. J. Liu, J. Lillis, and C. K. Cheng, "Design and implementation of a global router based on a new layout-driven timing model with three poles," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 1548–51, 1997.

[18] J. Cong and B. Preas, "A new algorithm for standard cell global routing," *Integration: the VLSI Journal*, vol. 14, no. 1, pp. 49–65, 1992.

[19] J. Cong and C. K. Koh, "Simultaneous driver and wire sizing for performance and power optimization," *IEEE Transactions on VLSI Systems*, vol. 2, pp. 408–425, Dec. 1994.

[20] S. S. Sapatnekar, "RC interconnect optimization under the Elmore delay model," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 392–396, 1994.

[21] L. P. P. P. V. Ginneken, "Buffer placement in distributed RC-tree networks for minimal elmore delay," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 865–868, 1990.

[22] J. Lillis, C. K. Cheng, and T. Y. Lin, "Simultaneous routing and buffer insertion for high performance interconnect," in *Proceedings of the Great Lake Symposium on VLSI*, pp. 148–153, 1996.

[23] J. C. Shah and S. S. Sapatnekar, "Wiresizing with buffer placement and sizing for power-delay tradeoffs," in *Proceedings of the International Conference on VLSI Design*, pp. 346–351, 1996.

[24] C. C. N. Chu and D. F. Wong, "Closed form solution to simultaneous buffer insertion/sizing and wire sizing," in *Proceedings of the ACM International Symposium on Physical Design*, pp. 192–197, 1997.

[25] T. Okamoto and J. Cong, "Buffered Steiner tree construction with wire sizing for interconnect layout optimization," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 44–49, 1996.

[26] A. Salek, J. Lou, and M. Pedram, "A simultaneous routing tree construction and fanout optimization algorithm," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 625–630, 1998.

[27] H. Zhou, D. F. Wong, I.-M. Liu, and A. Aziz, "Simultaneous routing and buffer insertion with restrictions on buffer locations," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 96–99, 1999.

[28] R. Gupta, B. Krauter, B. Tutuianu, J. Willis, and L. T. Pileggi, "The Elmore delay as a bound for RC trees with generalized input signals," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 364–369, 1995.

[29] J. Rubinstein, P. Penfield, and M. A. Horowitz, "Signal delay in RC tree networks," *IEEE Transactions on Computer-Aided Design*, vol. CAD-2, pp. 202–211, July 1983.

[30] C. L. Ratzlaff, N. Gopal, and L. T. Pillage, "RICE: Rapid interconnect circuit evaluator," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 555–560, 1994.

[31] N. H. E. Weste and K. Eshraghian, *Principles of CMOS VLSI design: a system perspective*. Reading, MA: Addison-Wesley Publishing Company, 1993.

[32] J. Cong, L. He, C.-K. Koh, and P. H. Madden, "Performance optimization of VLSI interconnect layout," *Integration: the VLSI Journal*, vol. 21, pp. 1–94, 1996.

[33] C. J. Alpert, A. Devgan, and S. T. Quay, "Buffer insertion for noise and delay optimization," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 362–367, 1998.

[34] C. J. Alpert and A. Devgan, "Wire segmenting for improved buffer insertion," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 588–593, 1997.

[35] C. C. N. Chu and D. F. Wong, "A new approach to simultaneous buffer insertion and wire sizing," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 614–621, 1997.

[36] S. Dhar and M. A. Franklin, "Optimum buffer circuits for driving long uniform lines," *IEEE Journal of Solid-State Circuits*, vol. 26, pp. 32–38, Jan. 1991.

[37] C. J. Alpert, A. Devgan, and S. T. Quay, "Buffer insertion with accurate gate and interconnect delay computation," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 479–484, 1999.

[38] J. Lillis, "Timing optimization for multi-source nets: characterization and optimal repeater insertion," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 214–219, 1997.

[39] J. Lillis, C. K. Cheng, and T. Y. Lin, “Optimal wire sizing and buffer insertion for low and a generalized delay model,” *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 437–447, Mar. 1996.

[40] J. Hu and S. S. Sapatnekar, “Simultaneous buffer insertion and non-Hanan optimization for VLSI interconnect under a higher order AWE model,” in *Proceedings of the ACM International Symposium on Physical Design*, pp. 133–138, 1999.

[41] D. W. Hightower, “A solution to line routing problems on the continuous plane,” in *The Sixth Design Automation Workshop*, pp. 1–24, 1969.

[42] C. Y. Lee, “An algorithm for path connection and its applications,” *IRE Transactions on Electronic Computers*, vol. EC-10, no. 3, pp. 346–365, 1961.

[43] J. Cong, J. Fang, and K.-Y. Khoo, “Via design rule consideration in multi-layer maze routing algorithms,” in *Proceedings of the ACM International Symposium on Physical Design*, pp. 214–220, 1999.

[44] S.-W. Hur, A. Jagannathan, and J. Lillis, “Timing driven maze routing,” in *Proceedings of the ACM International Symposium on Physical Design*, pp. 208–213, 1999.

[45] E. W. Dijkstra, “A note on two problems in connection with graphs,” *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[46] C. Chiang and M. Sarrafzadeh, “Global routing based on Steiner min-max trees,” *IEEE Transactions on Computer-Aided Design*, vol. 9, pp. 1318–25, Dec. 1990.

[47] M. Burstein and R. Pelavin, "Hierarchical wire routing," *IEEE Transactions on Computer-Aided Design*, vol. CAD-2, pp. 223–234, Oct. 1983.

[48] M. Marek-Sadowska, "Global router for gate array," in *Proceedings of the IEEE International Conference on Computer Design*, pp. 332–337, 1984.

[49] J. D. Cho and M. Sarrafzadeh, "Four-bend top-down global routing," *IEEE Transactions on Computer-Aided Design*, vol. 17, pp. 793–802, Sept. 1998.

[50] P. Raghavan and C. D. Thompson, "Multiterminal global routing: a deterministic approximation scheme," *Algorithmica*, vol. 6, pp. 73–82, 1991.

[51] E. Shragowitz and S. Keel, "A global router based on a multicommodity flow model," *Integration: the VLSI Journal*, vol. 5, pp. 3–16, Mar. 1987.

[52] R. C. Carden, J. Li, and C.-K. Cheng, "A global router with a theoretical bound on the optimal solution," *IEEE Transactions on Computer-Aided Design*, vol. 15, pp. 208–216, Feb. 1996.

[53] C. Albrecht, "Provably good global routing by a new approximation algorithm for multicommodity flow," in *Proceedings of the ACM International Symposium on Physical Design*, pp. 19–25, 2000.

[54] B. S. Ting and B. N. Tien, "Routing techniques for gate array," *IEEE Transactions on Computer-Aided Design*, vol. CAD-2, pp. 301–312, Oct. 1983.

[55] R. Nair, "A simple yet effective technique for global wiring," *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, pp. 165–172, Oct. 1987.

[56] K. W. Lee and C. Sechen, "A global router for sea-of-gate circuits," in *Proceedings of the European Design Automation Conference*, pp. 242–247, 1991.

[57] D. Wang and E. S. Kuh, "Performance-driven interconnect global routing," in *Proceedings of the Great Lake Symposium on VLSI*, pp. 132–136, 1996.

[58] J. Huang, X.-L. Hong, C.-K. Cheng, and E. S. Kuh, "An efficient timing-driven global routing algorithm," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 596–600, 1993.

[59] J. Hu and S. S. Sapatnekar, "A timing-constrained algorithm for simultaneous global routing of multiple nets," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 99–103, 2000.

[60] M. Marek-Sadowska, "Route planner for custom chip design," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 246–249, 1986.

[61] K. Zhu, Y.-W. Chang, and D. F. Wong, "Timing-driven routing for symmetrical-array-based FPGAs," in *Proceedings of the IEEE International Conference on Computer Design*, pp. 628–633, 1998.

[62] K.-Y. Khoo and J. Cong, "An efficient multilayer MCM router based on four-via routing," *IEEE Transactions on Computer-Aided Design*, vol. 14, pp. 1277–1290, Oct. 1995.

[63] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network flows: theory, algorithms, and applications.* Upper Saddle River, NJ: Prentice Hall, 1993.

[64] K. D. Wayne and L. Feischer, "Fast and simple approximation schemes for generalized flow," in *Proceedings of Symposium on Discrete Algorithm*, pp. 981–982, 1999.

[65] H.-M. Chen, H. Zhou, F. Y. Yang, H. H. Yang, and N. Sherwani, "Integrated floorplanning and interconnect planning," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 354–357, 1999.

[66] P. Sarkar, V. Sundararaman, and C.-K. Koh, "Routability-driven repeater block planning for interconnect-centric floorplanning," in *Proceedings of the ACM International Symposium on Physical Design*, pp. 186–191, 2000.

[67] Kusnadi and J. D. Carothers, "A method of measuring nets routability for mcm's general area routing problems," in *Proceedings of the ACM International Symposium on Physical Design*, pp. 186–192, 1999.

[68] J. Cong, T. Kong, and D. Z. Pan, "Buffer block planning for interconnect-driven floorplanning," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 358–363, 1999.

[69] F. F. Dragan, A. B. Kahng, I. Mandoiu, and S. Muddu, "Provably good global buffering using an available buffer block plan," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 104–109, 2000.

[70] F. F. Dragan, A. B. Kahng, I. Mandoiu, and S. Muddu, "Provably good global buffering by multiterminal multicommodity flow approximation." to appear in *Asia and South Pacific Design Automation Conference*, 2001.

[71] X. Tang and D. F. Wong, "Planning buffer locations by network flows," in *Proceedings of the ACM International Symposium on Physical Design*, pp. 180–185, 2000.

[72] J. Cong, "An interconnect-centric design flow for nanometer technologies," in *Proceedings of International Symposium on VLSI Technology, Systems, and Applications*, pp. 54–57, 1999.

[73] C. J. Alpert, T. C. Hu, J. H. Huang, A. B. Kahng, and D. Karger, "Prim-dijkstra tradeoffs for improved performance-driven routing tree design," *IEEE Transactions on Computer-Aided Design*, vol. 14, pp. 890–896, July 1995.