Optimization of Post-Layout Area, Delay and Power Dissipation

Hirendu Vaishnav

CENG Technical Report 95-20

Department of Electrical Engineering - Systems
University of Southern
Los Angeles, California 90089-2562
(213)740-4458

August 1995

OPTIMIZATION OF POST-LAYOUT AREA, DELAY AND POWER DISSIPATION

by

Hirendu Vaishnav

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(Computer Engineering)

August 1995

Copyright 1995 Hirendu Vaishnav

Dedication

To my parents,
Jaya and Parshuram Vaishnav,
and to my beloved wife,
Tara

Acknowledgements

First and foremost, I would like to thank my advisor Prof. Massoud Pedram for his guidance, constant support and encouragement throughout my Ph.D. work. I would also like to thank him for always demanding the best from me, for being the toughest critic of my work, and for giving me the freedom and encouragement to pursue rather risky areas of research. I am grateful to Prof. Melvin Breuer and Prof. Doug Ierardi for serving on my dissertation and guidance committee. Thanks are also due to Prof. Sandeep Gupta and Prof. Viktor Prasanna for being part of my guidance committee.

During my graduate studies at USC, I have benefited greatly from my interaction with many people. It is impossible to list them all here but I still thank them all. In particular, I would like to thank Sasan Iman, Juergen Kasper, Bahman Nobandegani and Chun-li Pu for tolerating me as their office mate at different times and for providing intellectual and challenging environment in the office. I would also like to thank Jui-Ming Chang, Chihshun Ding, Yung-Te Lai, Diana and Radu Marculescu, Ricky Pan, Ishwar Parulkar, Chi-Ying Tsui and others who I had the good fortune to work with and who have provided constant feedback on my ideas. I am indebted to Lucille Stivers, Dawn Ernst and Mary Zittercob of the Electrical Engineering Department at USC for their help throughout my Ph.D.

Getting a Ph.D. is a long and winding road. In my case, this journey seems to have started when my grandparents, Maniben and Dwarkadas, moved an astronomical distance (in those days) of 70 miles to a village with school so that their children can go to school. In spite of being uneducated, they knew the value of education and I am grateful for their foresight. I grateful to my father and mother for instilling in me that desire to learn and for encouraging me in my pursuit of higher education.

I would like to take this opportunity to thank the government and people of India for providing excellent primary, secondary and undergraduate educational systems;

and the government and people of America for providing an outstanding graduate/research environment. In particular, I am grateful for the support provided by National Science Foundation's Research Initiation Award under contract No. MIP-9223812 and Young Investigator Award under contract No. MIP-9457392.

Last but not least, I would like to thank my wife Tara for her constant support, encouragement and understanding during the last 10 years that I have known her. I would also like to thank my son, Divyansh, for waking me up early and for motivating me to get my act together at the end of my Ph.D. work.

Contents

Dedication ii

Acknowledgements iii

List Of Figures x

List Of Tables xiii

Abstract xv

1 Introduction 1
- 1.1 Layout-Driven Logic Synthesis 2
- 1.2 Power-Driven Physical Design 5
- 1.3 Outline of the Proposal 7

I Layout-Driven Logic Synthesis 11

2 Approaches to Capture Post-Layout Costs 12
- 2.1 Placement Based Approaches for Layout-Driven Logic Synthesis . . . 14
- 2.2 Structure Based Approaches for Layout-Driven Logic Synthesis 17

3 Alphabetic Trees: Enumeration and Optimization 19
- 3.1 Introduction 19
- 3.2 Terminology and Notation 22
- 3.3 Alphabetic Tree Enumeration 23
  - 3.3.1 Bounded Height Trees 31

3.3.2 Bounded Degree Trees 33
3.3.3 Bounded Height and Bounded Degree Trees 34
3.4 Alphabetic Tree Optimization 35
3.4.1 Subtree and Subforest Optimality 37
3.4.2 ST-optimal Trees 42
3.4.3 ST-optimal Binary Trees 46
3.4.4 ST-optimality and Regular Tree Cost Functions 49
3.4.5 ST-optimality and Quasi-Linear/Schur-Concave Tree Cost Functions 51
3.4.6 SF-optimal trees 54
3.5 Conclusion: Alphabetic Tree Enumeration and Optimization 56

4 Alphabetic Trees: Applications in Layout-Driven Logic Synthesis 57
4.1 Introduction 57
4.2 Prior Work 59
4.2.1 Motivation for the Use of Alphabetic Trees for Layout-Driven Logic Synthesis 60
4.3 Layout-Driven Technology Decomposition 61
4.3.1 Optimal NAND Decomposition of Product Terms 62
4.3.2 Optimal Decomposition of Sum-Of-Product Expressions . . . 64
4.3.3 Implementation and Experimental Results 66
4.4 Layout-Driven Fanout Optimization 71
4.4.1 Handling Different Loads 75
4.4.2 Handling Multiple buffers 77
4.4.3 Implementation and Experimental Results 78

5 An Analysis of Placement Based Approaches 84

6 Structure Based Measures of Post-Layout Costs 89
6.1 Post-Layout Area Measures based on Pin Count 90
6.1.1 Estimating Equi-perimeter Netlength Ratios 93
6.1.2 Estimating Perimeter Ratios 94
6.1.3 Estimating Normalized Netlengths 98
6.1.4 Evaluation of the Netlength Cost Function 99

6.2 Post-Layout Area Measures based on Fanout Ranges 102
6.2.1 Evaluation of Fanout Ranges 103
6.2.2 Signal Localization using Fanout Ranges 104
6.2.3 Signal Distribution using Fanout Ranges 105
6.3 Post-Layout Area Measures based on Other Graph Properties 107

7 Application of Structure Based Measures to Logic Synthesis 110
7.1 Pin Count Based Extraction 114
7.1.1 Minimizing Normalized Netlengths during Extraction 114
7.1.2 Experimental Results 116
7.2 Fanout Range Based Extraction 119
7.2.1 Updating the Fanout Ranges during Extraction 119
7.2.2 Improving the Efficiency of Fanout Range Update 120
7.2.3 Effect of Low Weight Divisors and Single Divisors 122
7.2.4 Experimental Results 123
7.3 Fanin Range Based Extraction 128
7.3.1 Prior Work in Performance Optimization 128
7.3.2 Performance Optimization during Extraction 131
7.3.3 Experimental Results and Discussion 136

8 Conclusion: Layout-driven Logic Synthesis 147
8.1 Main Results and Contributions 148
8.1.1 Placement Based Approaches 148
8.1.2 Structure Based Approaches 148
8.2 Discussion and Future Directions 149
8.2.1 Placement Based Approaches 149
8.2.2 Structure Based Approaches 151
8.2.3 Post-Layout Logic Resynthesis 152

II Power-Driven Physical Design 160

9 Delay Optimal Circuit Partitioning for Low Power 161
9.1 Introduction 161

9.2 The Power Model 165
9.3 Background 168
9.4 Power Optimal Partitioning for Trees 169
9.4.1 Definitions and Notations 169
9.4.2 An Enumerative Clustering Algorithm 171
9.4.3 Number of Distinct Cluster Patterns 176
9.4.4 Cluster Pattern Enumeration 178
9.4.5 Optimality of POWER_CLUSTER 182
9.5 Low Power Partitioning for DAGs 183
9.6 Post-Clustering Area Recovery 185
9.6.1 Area Recovery by Forcing a Single Clustering Solution 186
9.6.2 Area Recovery by Relabeling Based on Unique Label Assignment 186
9.6.3 Area Recovery by Relabeling Based on Required Times 188
9.7 Experimental Results and Extensions 189
9.7.1 Effect of Cluster Size on Power Dissipation 189
9.7.2 Results without Area Recovery 191
9.7.3 Results with Area Recovery 193
9.7.4 Results Considering Internal Power Dissipation 195
9.8 Conclusion: Delay Optimal Circuit Partitioning for Low Power 195

10 Standard Cell Placement and Routing for Low Power **199**
10.1 PCUBE: Performance-Driven Placement for Low Power 200
10.1.1 Global Optimization Phase 202
10.1.2 Slot Assignment Phase 204
10.1.3 Placement for Low Power under a Real Delay Model 204
10.1.4 Experimental Results and Discussions 205
10.1.5 Conclusion: Placement for Low Power 210
10.2 Standard Cell Routing for Low Power 210
10.2.1 Low Power Global-Routing 210
10.2.2 Feed-through Assignment for Low Power 216
10.2.3 Conclusion: Circuit Routing for Low Power 218

11 Conclusion: Power-driven Physical Design **220**

11.1 Main Results and Contributions 221
11.1.1 Delay Optimal Circuit Partitioning for Low Power 221
11.1.2 Standard Cell Placement and Routing for Low Power 221
11.2 Discussion and Future Directions 222
Bibliography **224**

List Of Figures

1.1 Conventional synthesis flow and typical set of operations applied during logic synthesis and physical design. 2

2.1 Optimal alphabetic fanout tree as compared with optimal LT-tree under the unit fanout delay model. 15

2.2 The link between alphabetic trees and placement based logic synthesis. 16

3.1 Illustrations of operations JOIN and SPLIT. 23

3.2 An illustration of one-to-one correspondence between $\Psi_{i,i+3}$ and $\bigcup_{k=2}^{4} \Psi_{k:i,i+3}$. 25

3.3 An illustration of alphabetic tree generation equation. 27

3.4 Generating alphabetic trees for a) 1 leaf node, b) 2 leaf nodes, c) 3 leaf nodes, and d) 4 leaf nodes. 27

3.5 Examples of ST-optimal tree cost functions. 39

3.6 Examples of SF-optimal tree cost functions. 41

3.7 An illustration of ST-optimal alphabetic tree generation equation. . . 43

3.8 Example showing 8 possible ways to generate alphabetic trees with four leaf nodes under right branches. 45

4.1 An illustration of delay optimal alphabetic and non-alphabetic technology decomposition under the unit delay model. 64

4.2 An illustration of noncrossing and crossing product terms. 65

4.3 An illustration of cube ordering heuristic. 66

4.4 An illustration of incremental placement mechanism. 68

4.5 An instance of alphabetic fanout optimization problem and the required time at the source under different delay models. 72

4.6 Illustration of *Rule 1* and *Rule 2* for optimal alphabetic fanout optimization. 75
4.7 Generating optimal alphabetic fanout trees. 80

5.1 The script used for conformity analysis. 85

6.1 Average netlength for different pin counts for three benchmark circuits. 91
6.2 An illustration of equi-perimeter netlength ratios. 91
6.3 Effect of W on $\nu(n)$, $N_w(n)$ and $N_h(n)$. Solid line corresponds to $W = \infty$ whereas dotted line corresponds to $W = 10$. 98
6.4 Estimated Vs. Actual average netlengths. 100
6.5 Number of nets with different pin count and sum of their netlengths for circuit *cps*. 101
6.6 Illustration of fanout ranges and fanout intervals. 102
6.7 Average netlengths for different fanout ranges. 103
6.8 Number of nets with different fanout ranges and sum of their netlengths in circuit *cps*. 104
6.9 Illustration of the square overlap function. 105
6.10 Illustration of an extraction. 108

7.1 Identifying the nodes whose fanout ranges may change due to extraction. 120
7.2 Illustration of the DD edges, DD Signature Set and RD Edges. 121
7.3 Illustration of Lawler's algorithm for optimal depth clustering with size constraint 5. 130
7.4 2-input AND decomposition: Balanced Vs. Huffman. 131
7.5 Illustration of fanin ranges and fanin intervals. 132
7.6 Effect of a subsequent extraction on fanin range of a node. 135

8.1 Alternate/future design flow. 153

9.1 Two delay optimal clusterings under size constraint 8 for circuit *con1*. 163
9.2 An illustration of effect of delay model on switching activity. 166
9.3 An illustration of size 3 cluster patterns. 170
9.4 An illustration of PD-set. 174

9.5 One to one correspondence between cluster patterns on binary tree structures and binary alphabetic trees. 177
9.6 Multiple inclusion of a gate in a cluster due to a reconvergent fanout. 184
9.7 Area recovery during low power partitioning by forcing a single clustering solution. 186
9.8 Area recovery during low power partitioning by unique label assignment.187
9.9 Area recovery during low power partitioning by relabeling based on required times. 189
9.10 Effect of varying the maximum cluster size. 190

10.1 The dependency loop involved in minimizing power dissipation under a real delay model during placement. 205
10.2 Switching activity profile for benchmark circuit *misex3*. 207
10.3 Wire Length and power distribution comparison between PCUBE and RITUAL. 208
10.4 Net distribution profile for benchmark example *misex3*. 209
10.5 A scenario where power dissipation can be improved during global routing. 213

List Of Tables

3.1 Number of alphabetic trees. 30

3.2 Number of alphabetic trees with bounded height. 32

3.3 Number of alphabetic trees with exact height. 33

3.4 Number of alphabetic trees with degree restriction. 34

3.5 Number of alphabetic trees with degree and height restriction on 10 leaf nodes. 35

4.1 Network depth comparisons after TechDecomp (TD) and after Mapping (M). 69

4.2 Results for layout-driven technology decomposition. 70

4.3 Results for layout-driven fanout optimization. 82

5.1 X and Y conformity for benchmark circuit *duke2*. 86

6.1 Multiplicative factors to estimate wire lengths given the semi-perimeter of the bounding box. 93

6.2 Expected normalized width of the bounding-box. 96

6.3 Expected normalized width as $W \to \infty$ and the percentage error. . . 97

6.4 Expected normalized netlength as a function of n. 99

7.1 Results using *script.rugged* for comparison with normalized netlength based extraction. 117

7.2 Results using *script.rugged.pix* (normalized by corresponding values in Table 7.1). 118

7.3 Results using *script.rugged* for comparison with fanout range based extraction. 124

7.4 Results of extraction minimizing sum of fanout ranges (normalized by corresponding values in Table 7.3). 125
7.5 Results of extraction minimizing the square of the fanout range overlaps (normalized by corresponding values in Table 7.3). 127
7.6 Results using *script.rugged* for comparison with fanin range based extraction. 139
7.7 Results using *script.delay* for comparison with fanin range based extraction. 140
7.8 Results using *script.rugged.frx* (normalized by corresponding values in Table 7.6). 141
7.9 Results using *script.delay.frx* (normalized by corresponding values in Table 7.7). 144
7.10 Results using *script.delay.frx* with "eliminate 100" (normalized by corresponding values in Table 7.7). 145

9.1 Results using Lawler's clustering algorithm. 192
9.2 Results using POWER_CLUSTER without area recovery (normalized by corresponding values in Table 9.1). 194
9.3 Results using POWER_CLUSTER with area recovery (normalized by corresponding values in Table 9.1). 196
9.4 Results using POWER_CLUSTER minimizing total power dissipation (normalized by corresponding values obtained with Lawler's clustering algorithm). 197

10.1 Results obtained with RITUAL and low power placement tool PCUBE. 206
10.2 PCUBE: Timing Vs. Normal mode. 206

Abstract

With the move towards deep-submicron technology, circuit designers enter a strange new world in which interconnect becomes a dominant factor in determining all costs associated with VLSI chips. These interconnect dominating, multi-million transistor deep-submicron circuits pose a great challenge to the CAD community, the only solution to which is a stronger interaction between layout and synthesis tools. This thesis intends to be a step in that direction by providing mechanisms to optimize post-layout area during logic synthesis (namely, layout-driven logic synthesis), and by providing physical design tools to minimize post-layout power dissipation.

Layout-Driven Logic Synthesis: The idea behind layout-driven logic synthesis is to minimize the routing cost during logic synthesis. To capture routing cost during logic synthesis, two basic approaches are proposed: (1) A *placement based approach* that estimates routing overhead based on a preliminary placement of the circuit; (2) A *structure based approach* that estimates routing overhead based on the structure of underlying Boolean graph. An analysis of placement based approach indicates that a placement based approach is more appropriate for technology dependent phase of logic synthesis. For earlier phases of logic synthesis, structure based routing measures that capture the inherent routing difficulty of a Boolean network are proposed. A placement based mechanism is then proposed for technology dependent logic synthesis whereas structure based routing measures are effectively minimized during the technology independent operation of logic extraction.

Power-Driven Physical Design: Traditional approaches to physical design address chip area and circuit delay, ignoring the power dissipation. However, with the advent of high through-put portable devices, it has become very important to achieve high performance at minimal cost in power dissipation. In this thesis, tools for performance-driven physical design for low power, i.e., partitioning, placement, and routing tools for low power and high performance, are presented.

Chapter 1

Introduction

During high level design, factors such as constraints imposed by the physical media, layout, interconnect and packaging are ignored. Physical design which is expected to address these issues, comes much later in the design hierarchy. By then, many of the key architectural and structural decisions have been made, thereby, limiting capability of physical design tools to generate the "best" solutions in terms of area and performance. In addition, once some parameter at the physical design stage fails to satisfy a constraint imposed on it, synthesis must be modified (often repeated) to accommodate the constraint. Furthermore, recent advances in CMOS technology aiming to reduce device sizes and to put more functionality on a single chip give rise to circuits in which the interconnect contribution dominates the chip area, circuit delay and power dissipation. Hence, addressing routing issues at all levels of design abstraction has become a necessity. This thesis is a step toward achieving this objective by

- proposing and analyzing a number of cost measures that estimate post-layout[1] routing cost for use during logic synthesis.
- presenting logic synthesis procedures that minimize post-layout chip area and/or delay.
- proposing performance-driven physical design procedures that minimize post-layout power dissipation.

[1]In this dissertation, *post-layout* costs refer to the circuit costs (i.e. area, delay and power) calculated/estimated while taking into account the interconnect contribution.

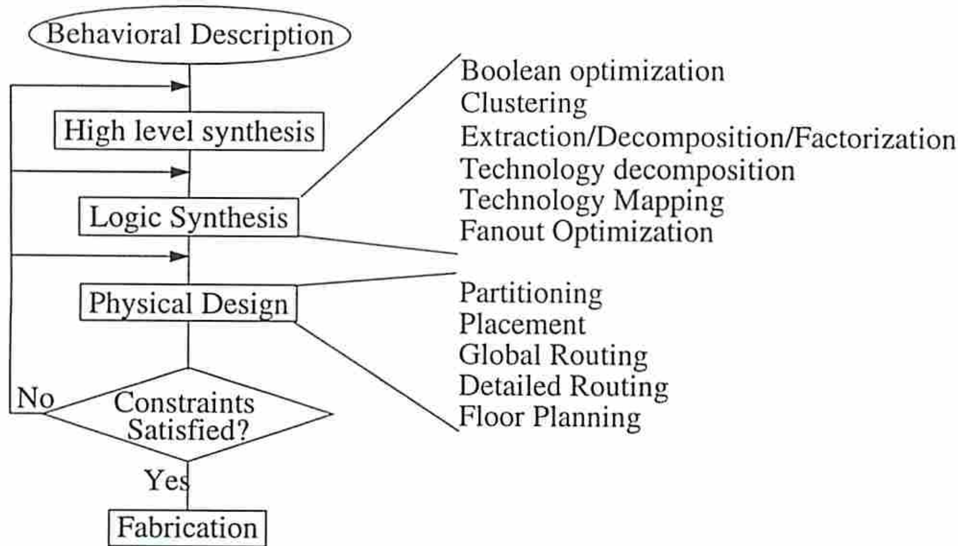


Figure 1.1: Conventional synthesis flow and typical set of operations applied during logic synthesis and physical design.

1.1 Layout-Driven Logic Synthesis

During the process of designing high performance VLSI circuits, designers often find that their designs do not meet the timing and/or power dissipation constraints after layout. This situation arises mainly due to the weak interaction between logic synthesis and physical design. Synthesis tools, which have the ability to significantly alter the timing and power dissipation of the circuits, do not have access to detailed layout information (e.g., interconnect delay, parasitic capacitances etc.) and hence cannot pinpoint the "problem areas" to apply the appropriate logic transformations. On the other hand, physical design tools lack the strength and flexibility to alter the circuit structure and hence cannot drastically influence the timing/power profile of the circuits. Conventional design flow that strictly separates the logic synthesis and physical design tools and typical operations applied are shown in Figure 1.1

Traditionally, objectives for logic synthesis operations have been gate area, circuit delay or power dissipation while ignoring the interconnect contribution to these objectives. During technology dependent phase of logic synthesis, values of gate area and gate-induced delay[2] can be determined exactly from the library implementation

[2]Gate-induced delay is delay due to the intrinsic delays of the gates and the load dependent delay taking into account the input loads of the fanout gates.

of the gates. Likewise, power dissipation, which is proportional to the load, is computed while ignoring the interconnect load. During technology independent logic synthesis (i.e, when the cell implementation is not available) literal count is used to approximate the active cell area and a simplified delay model (i.e., *unit delay model* that assigns unit delay to each gate ignoring the load values or *unit fanout delay model* that assumes that each gate has identical load) is used to estimate the circuit delay and power dissipation. These measures not only ignore the routing contribution, but they also provide only a rough estimate of the gate contribution to area, delay and power dissipation. Ignoring the interconnect contribution can lead to a circuit where minimization of gate contribution to area, delay and/or power dissipation has been achieved at a substantial increase in routing area and interconnect load resulting in an overall increase in chip area, circuit delay and/or power dissipation.

As the device sizes decrease, relative importance of interconnect in determining the circuit delay, chip area and power dissipation increases significantly. In fact, as characterized in [2], under an ideal scaling down by a factor of S, the gate capacitance decreases by a factor of $1/S$, whereas the capacitance of an interconnect remains the same. Furthermore, the interconnect resistance for global nets increase by a factor of $S^2 S_C$ where S_C is the scaling factor for the chip and accounts for the increase in chip size with a decrease in the device size. In fact, it has been shown in [2] that as the device sizes decrease, the critical length (i.e., a length beyond which the interconnect delay increases rapidly) decreases substantially. For example, for polysilicon interconnect with a resistance of 500 Ω, the critical netlength is reduced to 0.004 mm from 0.02 mm as the device sizes are reduced to 0.5 μm from 1 μm. Even for highly conductive aluminum interconnects, the critical netlength at 0.5 μm is reduced to 1.4 mm.

The problem posed by ignoring the interconnect contribution during synthesis is compounded further as feature sizes shrink to 0.5 μm and below, i.e., for deep-submicron technology. Here, IC designers enter a new world where interconnect delays become dominant, second-order effects such as cross-coupling become significant, and the shear complexity of million-transistor designs choke design tools. In fact, even in submicron circuits (i.e., feature sizes less than 1μ but greater than 0.5μ), routing accounts for about 60-80% of total chip area, 40-60% of the circuit delay

and a significant amount of total power dissipation. A combined effect of increasing contribution of interconnect and synthesis tools that ignore this contribution, is that the area, delay and/or power dissipation constraint violations are increased substantially after the interconnect contribution is taken into account, i.e., after placement and routing. This results in a dramatical increase in the number of synthesis-layout iterations required to satisfy these constraints, increasing the design time significantly. In fact, it has been reported that in many cases, in spite of a large number of synthesis-layout iterations, the circuit fails to converge to an implementation that satisfies the constraints [33]. This indicates a failure of existing synthesis tools to face the challenge posed by these multi-million transistor, interconnect dominating deep-submicron circuits.

To cope with these issues, a different design methodology is required where the sharp division between logic synthesis and physical design disappears. In this new design paradigm, synthesis tools work concurrently and interactively with floor-planning, placement and routing. Similarly, layout tools exploit the power of logic synthesis to eliminate design constraint violations without the need for costly, and often unpredictable, iterations between logical design and layout. Such links between synthesis and layout tools can be established by: 1) *layout-driven synthesis*, 2) *post-layout resynthesis*. The first mechanism provides more accurate estimates to synthesis tools either by utilizing the structural information or by referring to a companion floor-planning or placement solution whereas the second mechanism allows physical design tools to restructure the logic in order to meet the specifications. In this thesis, I address layout-driven logic synthesis, delegating the equally important topic of post-layout resynthesis to [113].

Research in logic synthesis for minimizing post-layout costs has been sparse. Pedram et al. [81, 80] proposed a mechanism in which an incrementally updated companion placement solution is used to estimate the routing area and delay during logic synthesis. However, as shown later in this thesis, this approach is effective during the technology dependent phase of logic synthesis but not during the technology independent phase. Saucier et al. [91] proposed a mechanism called *lexicographical extraction* to reduce the routing overhead of the circuit by deriving and maintaining an order amongst primary inputs of the circuit during extraction. This order ensures that a primary input with index i enters the logic cone of each primary output

at a depth greater than that of any other primary input with index less than i. This scheme results in a reasonable improvement in chip area for circuits in which the routing contribution of primary inputs is significant, and whenever the physical design tools react favorably to such modifications.

The problem of identifying generic cost functions that capture the post-layout cost of the circuit has been left unaddressed. Apart from being generic and hence applicable to any phase of logic synthesis, these measures should also be abstract enough to make them independent of the parameters beyond the control of logic synthesis tool, i.e., design style to implement the logic, type of placement tool used, technology using which the circuit will be fabricated, etc..

Thus, specific objectives of this research in layout-driven logic synthesis are the following:

- Propose and analyze generic and abstract measures of post-layout costs during logic synthesis.
- Propose optimization mechanisms to minimize these measures.

1.2 Power-Driven Physical Design

Minimizing power is the primary concern for portable, battery-powered devices like notebook computers, digital cellular telephones, and wireless networks as for such products, longer battery life translates to extended use and better marketability. With the convergence of telecommunications, computers, consumer electronics, and biomedical technologies, the number of low power applications is expected to grow rapidly. Another driving force behind design for low-power is that excessive power consumption is becoming the limiting factor in integrating more transistors on a single chip or on a multiple-chip module. Exploring the trade-offs between area, performance and power during synthesis and design is thus demanding more attention.

Many researchers (in solid state circuits and technology areas) have been studying low power/low voltage design techniques. For example, research is being conducted in low power DRAM and SRAM designs, aggressive voltage scaling and process optimization for active logic circuits, device modeling and simulation tools

for low power circuits, low power analog circuit design, etc. Other researchers (in computer architecture area) are exploring instruction set architectures and novel memory management schemes for low power, processor design using self-clocking, static and dynamic power management strategies, etc. The computer aided design community has recently started paying more attention to power estimation and low power design [16, 75, 30, 107, 69, 1, 49, 108].

Estimating or minimizing post-layout power dissipation is much more difficult than estimating or minimizing post-layout chip area during logic synthesis. Since switching activity of each gate can take any value between 0 and 1, power dissipation per unit length of interconnects may differ significantly across nets[3]. Hence, to accurately estimate post-layout power dissipation, very accurate estimation of interconnect lengths, specifically for the nets with high switching activity, is required. It is very difficult to accurately estimate absolute interconnect lengths (and hence, post-layout power dissipation) during logic-synthesis. However, since accurate interconnect lengths can be estimated effectively during physical design, post-layout power can be minimized effectively during physical design. Hence, it was concluded that to minimize post-layout power it is appropriate to focus on physical design. A further reason to address post-layout power dissipation during physical design is that minimization of post-layout power dissipation has not been attempted at physical design level.

The basic idea in these physical design tools for low power is to trade off the length/load of idle nets (i.e., nets with low switching activity) for reducing the length/load of highly active nets. However, since some of the delay critical nets may also have low switching activity, it is quite likely that a purely power minimizing tool could assign a longer length to delay critical nets and hence result in significant delay degradation. Hence, it is necessary that the performance of the resultant circuit should also be taken into account while performing physical design for low power.

Thus, the specific objective of this research in power-driven physical design is the following:

[3]The switching activity can also be greater than 1 in presence of hazards or glitches.

- Propose performance-driven physical design tools (i.e., partitioning, placement, routing, etc.) for low power.

1.3 Outline of the Proposal

The outline and a brief summary of remainder of the dissertation is as follows:

Part I: Layout-Driven Logic Synthesis

Part I presents approaches proposed in this dissertation for layout-driven logic synthesis. Specifically, two approaches for layout-driven logic synthesis are proposed: a placement based approach and a structure based approach. A brief outline and summary of Part I is as follows.

Chapter 2 introduces the concepts of placement based approach and structure based approach to capture post-layout circuit costs during logic synthesis. Chapter 2 also introduces the concept of *alphabetic trees.* Alphabetic trees are trees without any internal edge crossing given an imposed total order on their leaf gates. This chapter also describes how alphabetic trees can be used for placement based layout-driven logic synthesis.

Chapter 3 presents theory behind alphabetic tree enumeration and alphabetic tree optimization. It presents a systematic procedure for generating all alphabetic trees on a given number of leaf nodes under bounded degree and/or bounded height criteria. The number of such trees is calculated and used to derive upper bounds on the complexity of the corresponding alphabetic tree optimization problems. A general alphabetic tree optimization procedure is then described that has a time complexity of $O(6^n)$ for n leaf nodes. It is shown that this complexity reduces to $O(n^4)$ for the class of tree cost functions that satisfy certain properties (i.e., satisfy subtree or subforest optimality conditions). This is further reduced to $O(n^3t)$ for a tree where the the degree of every internal node is upper bounded by t and to $O(n^2)$ for a binary tree that satisfies the monotonicity principle. These results unify previous work on alphabetic

tree optimization and show that the polynomial time alphabetic tree optimization algorithms apply to a larger class of tree cost functions than previously thought. The proposed alphabetic tree generation mechanism is applicable to any tree optimization problem including those in logic synthesis.

Chapter 4 applies the alphabetic tree optimization procedures proposed in Chapter 3 to generate optimal alphabetical technology decomposition trees and optimal alphabetic fanout trees. Alphabetic order at fanins and fanouts of a complex gate is derived from a companion placement solution or from the circuit structure. Specifically, binary alphabetic tree generation mechanism is applied to generate optimal alphabetic technology decomposition of product terms. A mechanism to derive an order amongst product terms of a sum-of-product expression minimizing the crossing number is presented. Experimental result show about 6% improvement in delay and about 10% improvement in routing area compared to the standard area optimizing script in the logic synthesis tool SIS. In Section 4.4, generic non-binary alphabetic tree optimization procedures are applied to generate delay-optimal fanout buffer trees. The problem of fanout optimization is much difficult compared to technology decomposition as the resultant trees are no longer restricted to binary trees and the delay optimization needs to be performed under a load dependent library delay model. The fanout optimization problem is analyzed further and a set of rules that reduce size of the solution space while maintaining delay optimality are proposed. For fanout optimization, this procedure obtained 14% improvement in chip area for about the same delay.

Chapter 5 presents an analysis on the appropriateness of such a placement based approach during different stages of logic synthesis. This analysis confirms the belief that a placement based approach is not appropriate for earlier stages of logic synthesis unless a good incremental placement mechanism that produces circuits of quality comparable to that produced by a stand-alone placement tool (applied at the end of logic synthesis) is available. This lead to the development of structure based approaches to perform layout-driven logic synthesis that does not depend on a companion placement solution.

Chapter 6 introduces the concept of structure based approach for layout-driven logic synthesis. In this chapter, a few measures of post-layout costs are proposed. These measures are then analyzed to identify their effectiveness in capturing post-layout routing cost. Specifically, two structural parameters capturing the post-layout routing cost are proposed: *normalized netlengths* which uses the number of pins on the nets to determine relative routing cost of the nets in the circuit; and *fanout ranges* which captures the localization of a net. A routing measure based on normalized netlengths that captures the total routing cost of the circuit is proposed. Using the concept of fanout ranges, objective functions that lead to localization of each net as well as a distribution of nets across the circuit are proposed.

Chapter 7 summarizes the application of the normalized netlength based routing cost function and the fanout range based objective functions to the operation on logic extraction. I also present encouraging results obtained with both these experiments. This chapter also presents a parameter that is a variation of fanout ranges, namely fanin ranges, and proposes a fanin range based extraction procedure which leads to better delay and power dissipation in the circuit.

Chapter 8 presents concluding remarks based on my work in layout-driven logic synthesis and discusses future directions in the field of layout-synthesis interactions.

Part II: Power-Driven Physical Design

Part II of the thesis deals with performance-driven physical design tools targeting low power VLSI circuits.

Chapter 9 proposes an approach for delay optimal partitioning for minimizing the power dissipation of the circuit. Traditional approaches for delay-optimal partitioning are based on Lawler's clustering algorithm that, given a constraint on size of partitions, minimizes the circuit depth under a unit delay model. Lawler's algorithm guarantees a depth-optimal partitioning but it makes no attempt to explore alternative partitioning solutions that may have the same

depth but better power implementations. The algorithm proposed in this chapter provides a formal mechanism that implicitly enumerates alternate partitioning solutions and selects a partitioning solution that has the same depth but less power dissipation. The experimental results indicate that on average, 35% improvement in power dissipation was obtained at no loss in delay.

Chapter 10 presents a performance-driven placement tool for low power along with experimental results. It also presents a low power circuit routing technique and discusses the experimental results obtained with low power routing. For low power placement, a non-linear programming formulation based on a quadratic objective function minimizing power dissipation of the circuit under path based delay constraints is proposed. The distribution of gates across the chip is obtained by iteratively bisecting the circuit and introducing center of mass constraints that enforce gates to be equally distributed across the partitions. At each iteration, the objective function, embedded with center of mass constraints, is solved globally allowing gates to migrate across partitioning boundaries. Low power routing is then attempted by ordering the nets to be routed such that lengths of idle nets is traded off to reduce the lengths of power critical nets. A low power feed-through assignment phase is also proposed.

Chapter 11 presents concluding remarks for low power physical design.

Part I

Layout-Driven Logic Synthesis

Chapter 2

Approaches to Capture Post-Layout Costs

Many different parameters/factors affect the circuit routing, making it extremely difficult to arrive at a comprehensive and accurate post-layout cost measure. These factors could be broadly classified in two categories:

Circuit Independent or Environmental Factors : These factors include the technology factors (e.g., resistance and capacitance of an unit length of interconnect), design style factors (e.g., standard cell vs. FPGA), choice of placement/routing tools (e.g., quadratic placement vs. linear placement), etc..

Circuit Dependent Factors: These factors include the number of nets, number of pins on these nets, distribution of these nets across the circuit, etc..

The environmental factors can be handled using one of the following approaches: (1) they can be assumed fixed a-priori and the post-layout cost measures can be derived with respect to the corresponding environmental setup; or (2) the post-layout cost measures should be made abstract enough to be effective irrespective of these circuit independent factors. For the first approach, different post-layout cost measures need to be derived for each distinct environment[1]. The second approach is applicable if the proposed routing cost measure captures the inherent routing cost of the circuit, thereby making it effective irrespective of the environmental factors. In either case, environmental factors are effectively abstracted out of the actual expression of post-layout cost function. In this dissertation, the emphasis is on the latter approach which is more generic and promises to be more powerful.

[1]Some research has been conducted in this field to statistically estimate post-layout area and netlength for gate array based circuits [90, 94], standard cell based circuits [59, 83] etc..

Apart from abstracting out the environmental factors, a good post-layout cost measure should be able to capture effect of all circuit dependent factors. Also, such a cost measure should be simple enough to be calculated quickly during logic synthesis. Furthermore, the cost measure should be accurate enough to be of practical use. Thus, an ideal post-layout cost measure should be: (1) abstract enough to be effective irrespective of the circuit independent factors, (2) simple enough to be calculated efficiently during logic synthesis, and (3) accurate enough to be of practical use. It should be noted that depending on the application, it may be sufficient to measure only the relative post-layout costs instead of the absolute post-layout costs.

It is quite difficult to identify the parameters affecting the routing contribution of the circuit. Even when the parameters are identified and their individual effect on routing contribution is verified, interaction between the effect of different parameters may not be predictable, making it difficult to arrive at a comprehensive routing measure that captures all these parameters. On the other hand, assuming these parameters are orthogonal to each other (i.e., independent of each other), it is possible to arrive at simpler post-layout costs based on individual parameters. In this case, different post-layout cost measures can be optimized during different phases of logic synthesis depending on the ease of capturing/optimizing these costs. Hence, in this thesis, a number of routing costs are proposed based on different circuit parameters. I also identify logic synthesis procedures for which these costs can be estimated/optimized effectively and propose corresponding mechanisms to optimize post-layout chip area and delay during logic synthesis.

Two basic approaches to capture post-layout costs during logic synthesis are proposed: (1) placement based approaches, and (2) structure based approaches. For placement based approaches, post-layout cost is based on some placement parameters (e.g., placement locations) of the circuit whereas for structure based approaches the post-layout cost is based on some topological/structural parameters (e.g., number of gates connected by a net).

2.1 Placement Based Approaches for Layout-Driven Logic Synthesis

This concept was initially proposed by Pedram [80]. The basic idea behind the placement based approach is to integrate physical design tools within logic synthesis. The overall flow is as follows:

- A companion placement of the Boolean network is generated.
- This companion placement information is used to guide the logic synthesis tool to optimize post-layout costs.
- As the network is modified during logic synthesis, companion placement is incrementally updated to reflect the structural changes to the network.
- Eventually, companion placement is relaxed to the final placement.

The main advantage of such placement based approach is that during each operation of logic synthesis, exact placement information is available. Given the placement locations of the gates, it is very easy to calculate the routing contribution of the circuit. However, a disadvantage of this approach is that the companion placement locations are not very reliable during earlier stages of logic synthesis. Optimization of a cost function based on these exact but inaccurate locations can mislead the optimization procedure, giving inconsistent results. Furthermore, our experiments indicate that it is very difficult to propose an incremental placement update mechanism that produces circuits of a quality similar to those obtained by a stand-alone placement applied at the end of synthesis phase. Hence, an alternative placement based approach is proposed in this dissertation by using the companion placement in a more abstract (and hence, less error-prone) manner as described next.

In [80], the companion placement solution is used to extract exact information about the interconnect lengths which is then used to estimate post-layout area and delay. However, even during the later stages of logic synthesis, placement positions are not very reliable. Relying on these inaccurate locations to guide logic optimization procedures could lead to inconsistent results. However, though the exact placement locations may not conform to the final placement locations, it is more

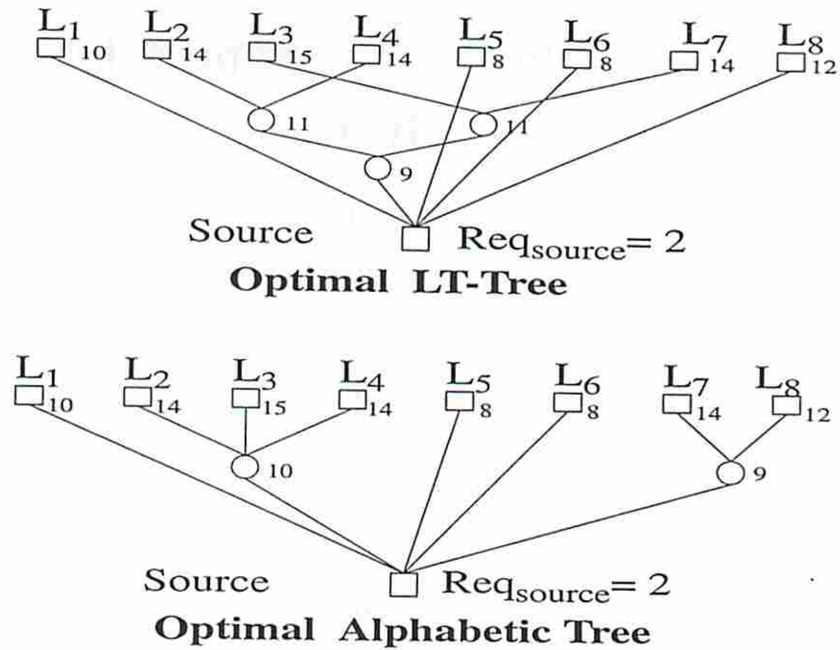


Figure 2.1: Optimal alphabetic fanout tree as compared with optimal LT-tree under the unit fanout delay model.

likely that the relative placement locations in companion placement solution conform to the final placement solution. In other words, when the pad locations remain the same, though it may be unreasonable to expect that two gates a and b are at the same X and Y locations in the final placement solution as in the companion placement solution; it is quite reasonable to expect that if gate a was on the left (bottom) of gate b in the companion solution, in the final placement solution also gate a would be on the left (bottom) of gate b, and vice versa. Hence, instead of relying on exact companion placement locations, I propose to use these locations only in a relative manner to improve the planarity of the underlying network. Specifically, relative directions of placed gates are used to synthesize the circuit such that no additional non-planarity is introduced (in terms of additional wire crossings) during logic synthesis procedures of technology decomposition and fanout optimization. This is achieved by using a special type of trees called alphabetic trees that produce wire crossing free technology decomposition trees and fanout trees given a linear order on the leaf nodes.

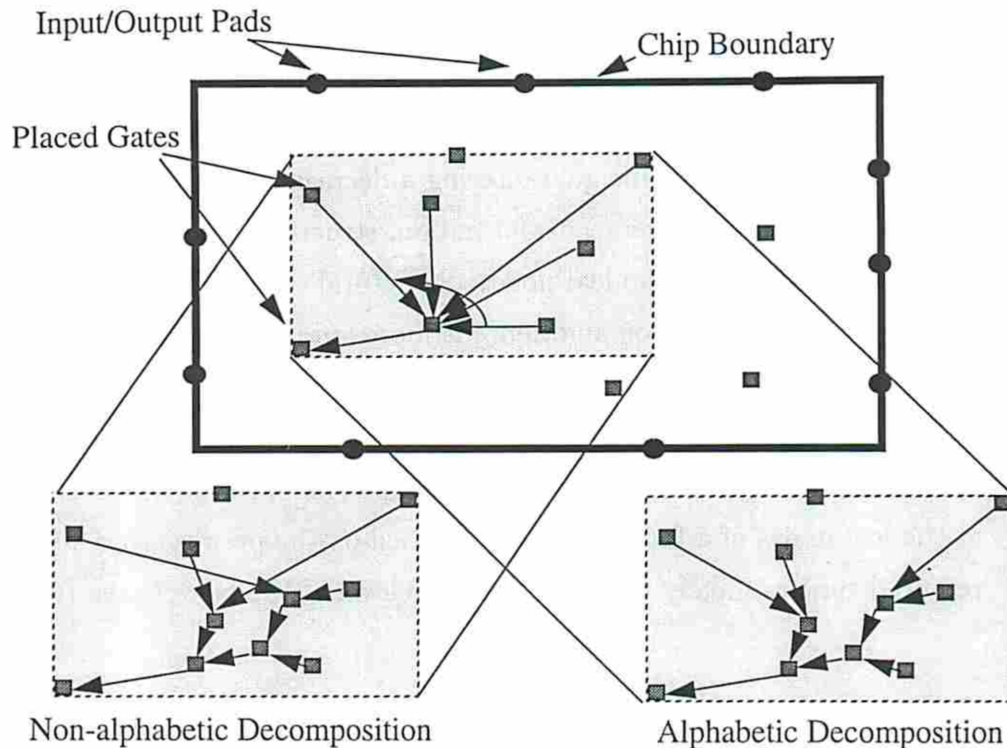


Figure 2.2: The link between alphabetic trees and placement based logic synthesis.

Alphabetic decomposition trees and alphabetic fanout trees provide a good trade-off between circuit performance and routability. These are the trees that minimize(maximize) arrival(required) time at the root of the decomposition(fanout) tree subject to a fixed linear order on the sinks, without creating any internal edge crossings (Figure 2.1). The penalty for using alphabetic trees is minimal. It has been shown that under the unit delay model, increase in depth is at most one, and increase in size is a constant multiplicative factor for optimal alphabetic fanout trees as compared to optimal non-alphabetic trees [42, 54]. In most cases, this is also true for other delay models as shown for fanout trees in Figure 2.1, where an optimal LT-tree[2] is compared to an optimal alphabetic tree under the unit fanout delay model.

Linear order on the input(output) nodes for decomposition(fanout) tree is derived from a companion placement solution of the circuit [82] as shown in Figure 2.2. This placement is incrementally updated during technology decomposition and

[2]A type of tree proposed in [104] for the problem of fanout optimization that is characterized by the procedure of constructing it.

relaxed after technology mapping and fanout optimization. Such wire crossing free technology decomposition and fanout optimization not only improve the planarity of underlying Boolean graph, but are also likely to improve the netlengths of the resultant circuit. In fact, it can be shown that any non-alphabetic tree can be converted to an alphabetic tree while guaranteeing a decrease in total netlength.

Instead of using placement information, structural information can also be used to derive the order on the leaf nodes. Structural information is more abstract than the placement information and hence is more appropriate for use during technology decomposition, i.e., when exact gate implementation of the circuit is not known. Other mechanisms to order leaf nodes during technology decomposition or fanout optimization can be used as well. For example, an ordering based on required times at the leaf nodes of a fanout tree can be used on the premise that sinks with similar required time are likely to be on the same level of the fanout tree [104].

2.2 Structure Based Approaches for Layout-Driven Logic Synthesis

The main idea behind the structure based approach is to identify some structural parameters based on the underlying graph structure of the circuit which are then used to derive post-layout cost measures that correlate well with the actual post-layout cost. The advantage of this approach is that it attempts to capture the inherent routing complexity/cost based on the network structure instead of emphasizing on the effects of one arbitrary planar embedding (e.g., a companion placement solution) of the circuit. Due to this dependence on the underlying network structure and not on a placement solution, this approach can lead to a more abstract routing cost measure that is independent of the environmental factors. Also, such abstract costs may be easier to compute and can be optimized effectively during earlier stages of logic synthesis. The main challenges associated with this approach are: identifying appropriate network parameters contributing to the routing cost of the circuit; and once identified, using them to derive a cost measure that correlates well with the actual post-layout area. A partial objective of this thesis is to address this challenge by deriving such cost functions and then optimizing them during logic synthesis to

improve the post-layout area, delay and/or power dissipation. Further discussion of structural parameters of routing cost and derivation of structure based routing measures is delegated to Chapters 6 and 7.

In the remainder of Part I, I present my work in both placement based as well as structure based approaches, starting with a placement based approach using alphabetic trees. The next chapter presents some theory behind alphabetic trees.

Chapter 3

Alphabetic Trees: Enumeration and Optimization

3.1 Introduction

The concept of tree has been around for centuries, for example, in family trees. The term "tree" first appeared in a paper by Cayley [9]. He is also generally credited for showing that the number of tree structures with n distinguished nodes is n^{n-2} [11]. This and other *tree enumeration* results have been well researched and are summarized by Moon [70].

Tree optimization differs from tree enumeration in the sense that tree optimization seeks to generate the best tree for a given application. If there are n nodes in a tree where each node i has a weight w_i, a measure of the quality of this tree can be obtained by defining a *tree cost function* in terms of the weights w_i and parameters associated with the tree structure. Using this tree cost function, it is determined whether one tree structure is better than another.

Pioneering work in tree optimization was done by Huffman [47]. He assumed that only leaf nodes were weighted. Weight of an internal node is obtained from weights of its immediate children using the weight *combining function*. For example, if nodes i and j with weights w_i and w_j are combined as children of node k, then $w_k = F(w_i, w_j)$, where F denotes the combining function. The application he was addressing was that of generating a prefix-free binary encoding of a set of symbols with minimum average codeword length given the probability of occurrence of each symbol, i.e., generating an optimal binary tree minimizing $\sum_{i=1}^{n} w_i l_i$ where n is the

number of leaf nodes, w_i is the weight of leaf node i and l_i is the length of the path from leaf node i to the root of the tree. The corresponding combining function was $F(w_i, w_j) = w_i + w_j$. Huffman also proposed generalization of the algorithm to generate optimal t-ary trees. It was discovered that his algorithm generated optimal trees not only for *additive* combining function given above, but also for other combining functions (e.g., *minimax* combining function where $F(w_i, w_j) = max(w_i, w_j) + 1$ and the tree cost function is $max_i(w_i + l_i)$). Glassey and Karp [32] provided the necessary and sufficient conditions for a combining function to generate optimal trees using Huffman's algorithm. Parker [52] provided a more precise characterization of combining functions as *quasi linear* functions and showed that these functions always generate optimal trees under Huffman's algorithm when the tree cost function is *schur* concave. Use of Huffman's algorithm for several applications has been reported in [79, 26, 27, 66].

Variations to tree optimization, e.g., generation of optimal trees with height constraint, generation of optimal trees minimizing the variance of tree depths, generation of optimal trees given an order on the leaf nodes etc., have been proposed for different applications. Trees generated under an order restriction on the leaf nodes are known as *alphabetic trees*. This name was coined in a paper by Gilbert and Moore [31] where they introduced the concept of alphabetic trees in the context of encoding an alphabet with a linear ordering relationship between the letters of the alphabet. Subsequently, alphabetic trees have found applications in fields of computer science (search trees, information storage and retrieval etc.) [58, 46, 28, 123, 50, 121, 36, 35, 19], matrix multiplication [45], information theory [31, 43, 76], fault diagnosis (and similar applications like medical diagnosis, laboratory analysis, genetic identification etc.) [27, 79, 127], and logic synthesis [84, 111].

Pioneering work in tree enumeration of alphabetic trees was done by Cayley in 1889 [10] where he provided the number of alphabetic trees on n leaf nodes. He calculated the number of alphabetic trees while allowing internal nodes to have one or more children. He also considered a case where each internal node bifurcates, i.e., where the resulting alphabetic tree is binary, and rediscovered the *catalan* numbers that give the number of alphabetic binary trees on $n+1$ leaf nodes. Catalan numbers were again derived by Gilbert and Moore in their classic paper [31] where they provided an $O(n^3)$ algorithm to obtain the best binary alphabetic tree. After these

initial works, the tree enumeration for alphabetic trees has been left unaddressed. Specifically, the problem of enumerating all alphabetic trees on n leaf nodes where each internal node at least bifurcates, has been left unresolved. This is an important problem as all applications of alphabetic trees listed above require that each internal node at least bifurcates.

The field of tree optimization of alphabetic trees has been relatively more active. Knuth [58] improved upon $O(n^3)$ algorithm of Gilbert and Moore for constructing an optimal alphabetic binary tree by proposing an $O(n^2)$ algorithm utilizing the concept of *monotonicity*. Hu and Tucker [46] proposed an algorithm similar to Huffman's algorithm with run time of $O(nlogn)$. Hu et al. [44] defined a class of combining functions called *regular functions* and showed that all combining functions that are regular functions generate optimal alphabetic binary trees using their algorithm. Kirkpatrick and Klawe [54] proposed a linear algorithm for alphabetic binary tree optimization under minimax combining function with integer leaf weights and an $O(n\log n)$ algorithm for real leaf weights. Wessner [123] and Itai [50] proposed $O(n^2h)$ algorithms to generate optimal alphabetic binary trees with a height restriction h.

For multi-way (i.e., non-binary) alphabetic trees, the work has focused mainly on additive or minimax cost functions. Gotlieb [35] and Vaishnavi et al. [121] independently provided an $O(n^3\log t)$ algorithm for generating optimal alphabetic trees where each internal node has at most t children. Vaishnavi et al. also identified a tree cost function for which $O(n^3t)$ is the best possible runtime and showed that $O(n^3\log t)$ complexity is possible only for a restricted class of tree cost functions. These approaches were proposed for the additive combining function (except for Coppersmith et al. [19], who have addressed minimax combining function proposing an $O(n\log n)$ algorithm for a unit delay based minimax combining function). However, runtimes of $O(n^3t)$ holds for a larger class of combining functions. Characterization of these combining functions has not been attempted in the past.

This work solves the important problem of counting the number of alphabetic trees where each internal node at least bifurcates. Recurrence equations for the number of alphabetic trees with bounded height or bounded degree or both are also derived. Next, tree optimization algorithms for alphabetic trees with general tree cost functions are provided and it is shown that, with appropriate restrictions, these algorithms reduce to the best known algorithms proposed for additive or minimax

tree cost functions. Finally, some applications of the tree optimization algorithms in logic synthesis are described.

The remainder of the chapter is organized as follows. In Section 3.2, the notations and definitions used in this chapter are introduced. This is followed by some enumeration results on alphabetic trees in section 3.3. Section 3.4 contains algorithms for alphabetic tree optimization and characterize the tree cost functions for which this algorithm generates optimal alphabetic tree in polynomial time. In the next chapter, applications of these algorithms to generate optimal alphabetic trees for technology decomposition and fanout optimization in the field of logic synthesis are presented.

3.2 Terminology and Notation

I follow the standard definitions of graphs and trees [23]. Trees are connected graphs with no cycles. Weighted graphs refer to node-weighted graphs. A forest refers to a set of trees built on some set of leaf nodes such that each tree is node disjoint from any other tree and the set of trees in the forest cover the set of leaf nodes. Support of a forest F refers to the set of leaf nodes corresponding to the forest F. The notation used in this dissertation is as follows. A k-tree forest with support $\{L_i, \ldots, L_{i+d}\}$ is a set of k trees such that support of these k trees partition $\{L_i, \ldots, L_{i+d}\}$. This k-tree forest is denoted by $F_{k:i,i+d}$. A 1-tree forest on leaf node i through $i+d$, namely, $F_{1:i,i+d}$ is denoted as $T_{i,i+d}$. A 1-tree forest on leaf nodes i through $i+d$ with a height bound h, a degree bound t and a root degree bound r (where $r \leq t$) is denoted by $T^{h:r,t}_{i,i+d}$.

A collection of alphabetic forests is denoted by Ψ. Specifically, a collection of k-tree forests on leaf nodes i through $i+d$ is denoted by $\Psi_{k:i,i+d}$. A collection of 1-tree forests on leaf nodes i through $i+d$, namely $\Psi_{1:i,i+d}$, is also denoted by $\Psi_{i,i+d}$. A collection of 1-tree forests on leaf nodes i through $i+d$ with a height bound h, a degree bound t and a root degree bound r (where $r \leq t$) is denoted by $\Psi^{h:r,t}_{i,i+d}$. The corresponding number of 1-tree forests in these collections of forests on d leaf nodes is denoted by Φ_d. For example, $|\Psi^{h:r,t}_{i,i+d}| = \Phi^{h:r,t}_d$. A collection of 1-tree forests on leaf nodes i through $i+d$ with an *exact* height restriction h, a degree bound t, and a root degree bound r (where $r \leq t$) is denoted by $\Psi^{[h]:r,t}_{i,i+d}$. With this notation, $\Psi^{h:t,t}_{i,i+d}$

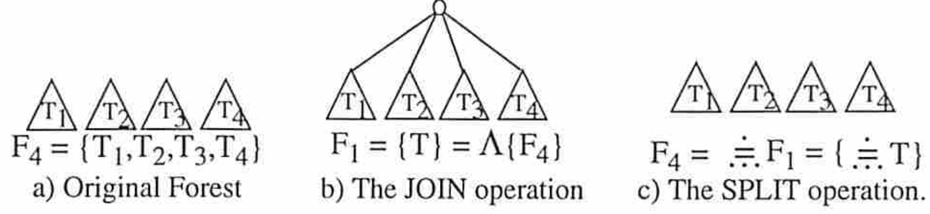


Figure 3.1: Illustrations of operations JOIN and SPLIT.

corresponds to collection of alphabetic 1-tree forests with height restriction h and degree restriction t. This is denoted as $\Psi_{i,i+d}^{h:t}$. Noting that $\Psi_{i,i+d}^{i:j,k} = \Psi_{i,i+d}^{d:d+1,d+1}$ $\forall i \geq d$ and $\forall j, k \geq d+1$, I substitute by ∞ each index for which there is no restriction, i.e., on leaf nodes i through $i+d$, collection of 1-tree forests without height restriction is denoted by $\Psi_{i,i+d}^{\infty:t}$ while collection of 1-tree forests without degree restriction is denoted by $\Psi_{i,i+d}^{h:\infty}$. A collection of 1-tree forests on leaf nodes i through $i+d$, namely, $\Psi_{i,i+d}$ is the same as $\Psi_{i,i+d}^{\infty:\infty}$. Finally, $\Psi_{i,i}$ gives the set of alphabetic 1-tree forests on a single leaf node, i.e., the leaf node itself.

3.3 Alphabetic Tree Enumeration

Definition 3.3.1 *Given a forest F with rooted trees,* JOIN *of F (denoted by $\wedge F$) is obtained by generating a rooted* 1*-tree forest T with the roots of trees in F as children of the root of the tree in* 1*-tree forest T (Figure 3.1).*

Definition 3.3.2 *Given a rooted* 1*-tree forest T,* SPLIT *of T (denoted by $.\dot{\div}.T$) is obtained by deleting the root making all its children rooted trees in a forest F (Figure 3.1).*

For example, consider a forest with 4 trees, i.e., $F_4 = \{T_1, T_2, T_3, T_4\}$ as shown in Figure 3.1. Applying JOIN operation to F_4 results in a 1-tree forest $F_1 = \{T\}$. By applying SPLIT on F_1 we get back $F_4 = \{T_1, T_2, T_3, T_4\}$. Thus, $\wedge(.\dot{\div}.F_1) = F_1$ and $.\dot{\div}.(\wedge F_4) = F_4$.

These operations are also applicable to a collection of forests. Given a collection of forests $\Psi_{i,i+d}$, $\wedge\Psi_{i,i+d}$ results in a collection of 1-tree forests such that each 1-tree forest corresponds to JOIN of some forest in $\Psi_{i,i+d}$. Likewise, given a collection of 1-tree forests $\Psi_{i,i+d}$, $.\dot{\div}.\Psi_{i,i+d}$ results in a collection of forests such that each forest in

this collection corresponds to SPLIT of some 1-tree forest in $\Psi_{i,i+d}$. For example, if $\Psi = \{\{T_{i,i+j'}, T_{i+j'+1,i+k'}, T_{i+k'+1,i+d}\}, \{T_{i,i+j''}, T_{i+j''+1,i+k''}, T_{i+k''+1,i+d}\}\}$, where $j' < k' < d$ and $j'' < k'' < d$, then $\wedge\Psi = \Psi' = \{\{T'_{i,i+d}\}, \{T''_{i,i+d}\}\}$ where $\{T'_{i,i+d}\} = \wedge\{T_{i,i+j'}, T_{i+j'+1,i+k'}, T_{i+k'+1,i+d}\}$ and $\{T''_{i,i+d}\} = \wedge\{T_{i,i+j''}, T_{i+j''+1,i+k''}, T_{i+k''+1,i+d}\}$ and $.\dot{\div}.\Psi' = \Psi$. SPLIT can also be applied on a collection of forests where each forest has different number of trees.

For the purpose of alphabetic tree enumeration, all distinct alphabetic trees on leaf nodes 1 through n need to be counted. In the rest of the dissertation, $\Psi_{1,n}$ is used to refer to this exhaustive collection of trees on n leaf nodes and Φ_n to refer to the number of such trees, i.e., $\Phi_n = |\Psi_{1,n}|$.

Lemma 3.3.1

$$\Psi_{i,i+d} = \bigcup_{k=2}^{d+1} \wedge(\Psi_{k:i,i+d}) \tag{3.1}$$

Proof $\Psi_{i,i+d}$ could be partitioned with respect to the arity of the root of each element of $\Psi_{i,i+d}$. The lemma follows from the observation that there is a one to one correspondence between each element of $\Psi_{k:i,i+d}$ and elements of $\Psi_{i,i+d}$ with k-ary roots. This is also shown in Figure 3.2. ∎

Corollary 3.3.2

$$.\dot{\div}.\Psi_{i,i+d} = \bigcup_{k=2}^{d+1} \Psi_{k:i,i+d} \tag{3.2}$$

Definition 3.3.3 *Given two sets Ψ' and Ψ'' their* Cartesian product *(denoted by $\times$) is defined as:* $\Psi' \times \Psi'' = \{F \mid F = F' \cup F'',\ F' \in \Psi',\ F'' \in \Psi''\}$.

Theorem 3.3.3

$$\begin{aligned}\Psi_{i,i+d} \quad = \quad & \bigcup_{j=0}^{d-2} (\wedge(\Psi_{i,i+j} \times \Psi_{i+j+1,i+d}) \cup \wedge(\Psi_{i,i+j} \times (.\dot{\div}.\Psi_{i+j+1,i+d}))) \\ & \bigcup \wedge(\Psi_{i,i+d-1} \times \Psi_{i+d,i+d}) \qquad\qquad \textit{for } d \geq 1\end{aligned} \tag{3.3}$$

Proof First, it is shown that

$$\Psi_{i,i+d} \quad = \quad \bigcup_{j=0}^{d-1} \wedge(\Psi_{i,i+j} \times [\bigcup_{k=1}^{d-j} \Psi_{k:i+j+1,i+d}]) \qquad \text{for } d \geq 1 \tag{3.4}$$

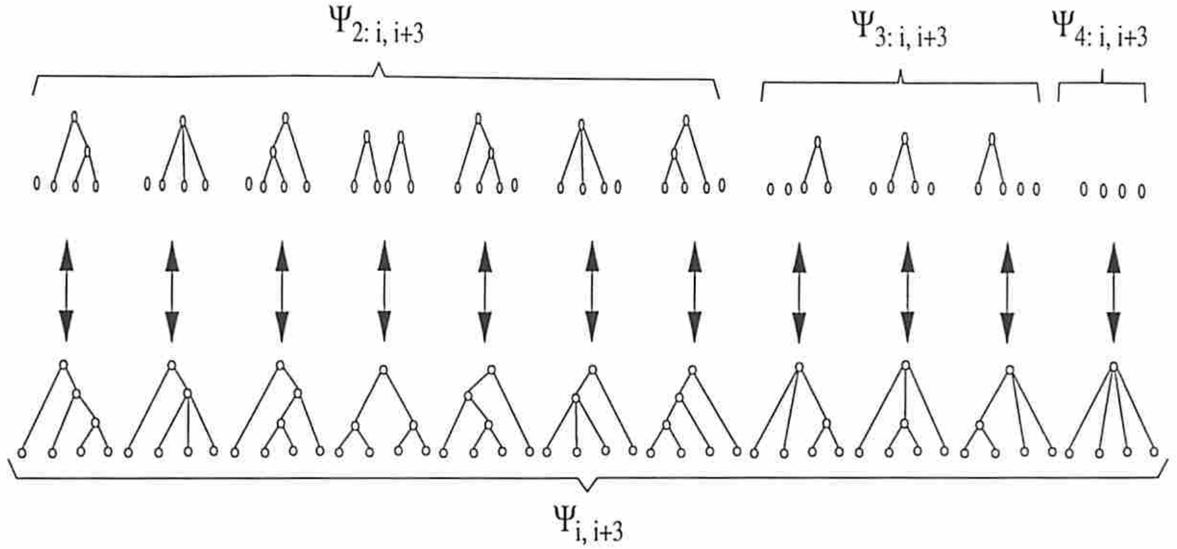


Figure 3.2: An illustration of one-to-one correspondence between $\Psi_{i,i+3}$ and $\bigcup_{k=2}^{4} \Psi_{k:i,i+3}$.

Examining equation (3.4), for some values of j and k, a 1-tree forest with support $\{i, \ldots, i+j\}$ is JOINed with a k-tree forest with support $\{i+j+1, \ldots, i+d\}$ to generate a 1-tree forest with support $\{i, \ldots, i+d\}$. Increasing value of j in this equation corresponds to incrementally adding more leaf nodes to the leftmost subtree of the tree root. Denote by $\Psi_{(i,i+j),i+d}$ the collection of alphabetic trees on leaf nodes i through $i+d$ such that $0 \leq j < d$ and leaf node $i+j$ is the rightmost leaf node of the leftmost subtree. Hence,

$$\Psi_{i,i+d} \quad = \quad \bigcup_{j=0}^{d-1} \Psi_{(i,i+j),i+d} \tag{3.5}$$

A generalization of Lemma 3.3.1 gives us

$$\Psi_{(i,i+j),i+d} \quad = \quad \bigcup_{k=2}^{d-j+1} \wedge(\Psi_{k:(i,i+j),i+d}) \tag{3.6}$$

However, all k-ary trees with node $i+j$ as the rightmost leaf node of the leftmost tree can be generated by JOINing each tree on leaf nodes i through $i+j$ with all

possible $k-1$ rooted forests on leaf nodes $i+j+1$ through $i+d$. This allows me to rewrite equation (3.6) as

$$
\begin{aligned}
\Psi_{(i,i+j),i+d} &= \bigcup_{k=2}^{d-j+1} \wedge(\Psi_{i,i+j} \times \Psi_{k-1:i+j+1,i+d}) \\
&= \bigcup_{k=1}^{d-j} \wedge(\Psi_{i,i+j} \times \Psi_{k:i+j+1,i+d})
\end{aligned}
$$

Substituting the above in equation (3.5):

$$
\begin{aligned}
\Psi_{i,i+d} &= \bigcup_{j=0}^{d-1} \bigcup_{k=1}^{d-j} \wedge(\Psi_{i,i+j} \times \Psi_{k:i+j+1,i+d}) \\
&= \bigcup_{j=0}^{d-1} \wedge(\Psi_{i,i+j} \times [\bigcup_{k=1}^{d-j} \Psi_{k:i+j+1,i+d}])
\end{aligned}
$$

The second term of the Cartesian product above corresponds to all k-tree forests for $1 \leq k \leq d-j$ on $d-j$ leaf nodes (leaf nodes $i+j+1$ through $i+d$). All possible k-tree forests on leaf nodes $i+j+1$ through $i+d$ for $2 \leq k \leq d-j$, namely $\Psi_{k:i+j+1,i+d}$, can be obtained by SPLITting each element of $\Psi_{i+j+1,i+d}$ with k-ary root. This gives the following $d-1$ equations.

$$
\begin{aligned}
\Psi_{2:i,i+d} &= \Psi_{i,i} \times \Psi_{i+1,i+d} \cup \cdots \cup \Psi_{i,i+d-2} \times \Psi_{i+d-1,i+d} \cup \Psi_{i,i+d-1} \times \Psi_{i+d,i+d} \\
\Psi_{3:i,i+d} &= \Psi_{i,i} \times \Psi_{2:i+1,i+d} \cup \cdots \cup \Psi_{i,i+d-2} \times \Psi_{2:i+d-1,i+d} \\
\vdots & \quad \vdots \\
\Psi_{d+1:i,i+d} &= \Psi_{i,i} \times \Psi_{d:i+1,i+d}
\end{aligned}
$$

Taking the union of RHS's and then JOINing each element gives $\Psi_{i,i+d}$. Performing the same operation on LHS's and applying Corollary 3.3.2 on terms of the last $d-2$ equations above gives the desired result. ∎

Equation (3.3) is illustrated for n = 1, 2, 3 and 4 in Figure 3.4.

For leaf nodes 1 through n, equation (3.3) can be rewritten as

$$
\Psi_{1,n} = \bigcup_{j=0}^{n-3} (\wedge(\Psi_{1,j+1} \times \Psi_{j+2,n}) \cup \wedge(\Psi_{1,j+1} \times (.\dot{\div}.\Psi_{j+2,n})))
$$

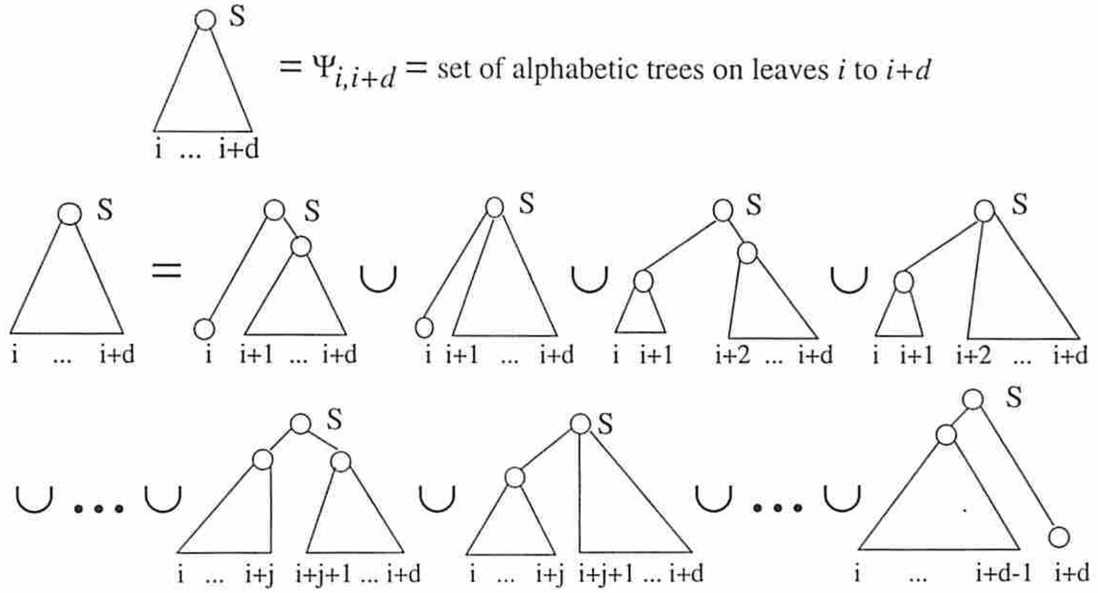


Figure 3.3: An illustration of alphabetic tree generation equation.

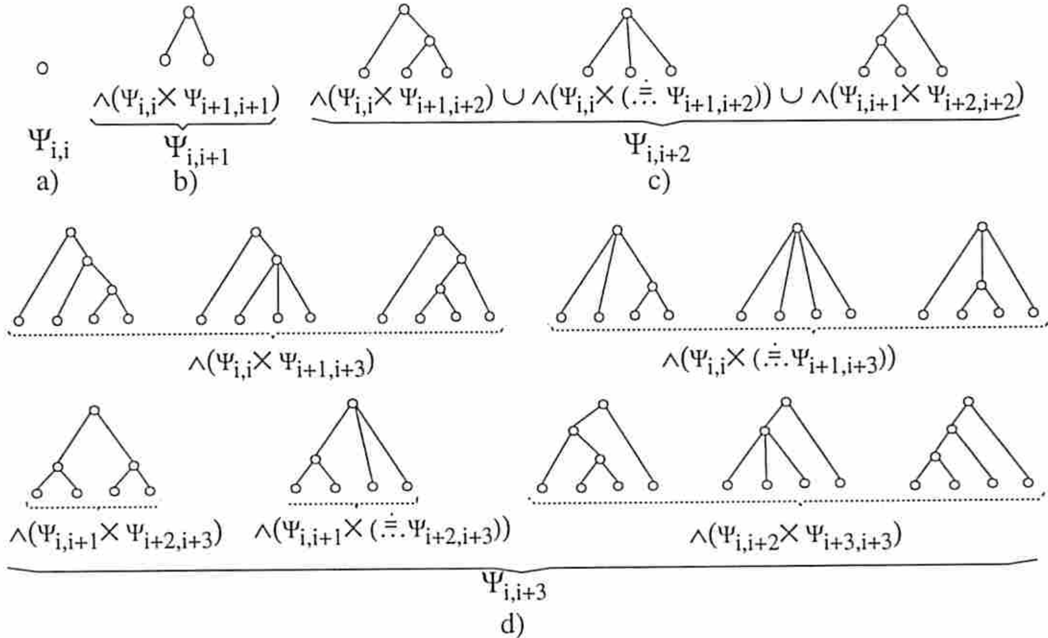


Figure 3.4: Generating alphabetic trees for a) 1 leaf node, b) 2 leaf nodes, c) 3 leaf nodes, and d) 4 leaf nodes.

$$\bigcup \wedge(\Psi_{1,n-1} \times \Psi_{n,n}) \qquad \text{for } d \geq 1$$

Hence,

$$\begin{aligned}
\Phi_n &= |\Psi_{1,n}| \\
&= |\bigcup_{j=0}^{n-3} (\wedge(\Psi_{1,j+1} \times \Psi_{j+2,n}) \cup \wedge(\Psi_{1,j+1} \times (.\dot{\div}.\Psi_{j+2,n})))| \\
&\quad +| \wedge (\Psi_{1,n-1} \times \Psi_{n,n})| \qquad \text{for } d \geq 1 \\
&= \sum_{j=0}^{n-3} (| \wedge (\Psi_{1,j+1} \times \Psi_{j+2,n})| + | \wedge (\Psi_{1,j+1} \times (.\dot{\div}.\Psi_{j+2,n}))|) \\
&\quad +| \wedge (\Psi_{1,n-1} \times \Psi_{n,n})| \qquad \text{for } d \geq 1 \\
&= \sum_{j=0}^{n-3} (|\Psi_{1,j+1}||\Psi_{j+2,n}| + |\Psi_{1,j+1}||.\dot{\div}.\Psi_{j+2,n}|) \\
&\quad +|\Psi_{1,n-1}||\Psi_{n,n}| \qquad \text{for } d \geq 1 \\
&= \sum_{j=0}^{n-3} (\Phi_{j+1}\Phi_{n-j-1} + \Phi_{j+1}\Phi_{n-j-1}) + \Phi_{n-1}\Phi_1 \qquad \text{for } d \geq 1 \\
&= 2\sum_{j=0}^{n-3} \Phi_{j+1}\Phi_{n-j-1} + \Phi_{n-1}\Phi_1 \qquad \text{for } d \geq 1 \\
&= 2\sum_{j=1}^{n-2} \Phi_j\Phi_{n-j} + \Phi_{n-1}\Phi_1 \qquad \text{for } d \geq 1
\end{aligned}$$

Thus, Φ_n for n leaf nodes is determined using the recursive difference equation given below.

$$\Phi_n = \begin{cases} 2\sum_{j=1}^{n-2} (\Phi_{n-j}\Phi_j) + \Phi_{n-1}\Phi_1 \text{for } n \geq 2 \\ 1 \qquad \text{for } n = 1 \end{cases} \tag{3.7}$$

The above equation is a convoluted recurrence equation whose solution is given below.

Lemma 3.3.4 *The number of alphabetic trees on n leaf nodes is given by*

$$\Phi_n = -\frac{1}{4} \sum_{j=0}^{\lfloor n/2 \rfloor} \begin{pmatrix} n-j \\ j \end{pmatrix} (-6)^{n-2j} \begin{pmatrix} 1/2 \\ n-j \end{pmatrix} \text{for } n \geq 2 \tag{3.8}$$

Proof Equation (3.7) can be rewritten as

$$\Phi_n = 2\sum_{j=1}^{n-1} \Phi_{n-j}\Phi_j - \Phi_{n-1} + [n = 1]. \qquad \text{for } n \geq 1$$

where $[n = 1]$ is 1 when $n = 1$ and 0 otherwise. Shifting this series left by one, a new series Φ' is obtained in which $\Phi'_i = \Phi_{i+1}$.

$$\Phi'_n = 2\sum_{j=0}^{n-1} \Phi'_{n-1-j}\Phi'_j - \Phi'_{n-1} + [n = 0]. \qquad \text{for } d \geq 0$$

Solution to the above can be obtained using generating function principles [37]. The generating function $\Phi(Z)$ for any series with series element Φ_i is given by $\sum_i \Phi_i Z^i$. Multiplying the above equation with Z^n and summing over n, we get

$$\Phi'(z) = \sum_n \Phi'_n Z^n = 2\sum_{n,j} \Phi'_{n-1-j}\Phi'_j Z^n - \sum_d \Phi'_{n-1} Z^n + \sum_{n=0} Z^n$$

$$\Phi'(z) = 2\sum_{n,j} \Phi'_{n-1-j} Z^{n-j} \Phi'_j Z^j - \sum_n \Phi'_{n-1} Z^n + 1$$

Using identity $\sum_i \Phi'_{i-1} Z^i = Z\Phi'(Z)$, we get

$$\Phi'(z) = 2Z\Phi'(Z)\Phi'(Z) - Z\Phi'(Z) + 1 = 2Z\Phi'(Z)^2 - Z\Phi'(Z) + 1$$

$$2Z\Phi'(Z)^2 - (1 + Z)\Phi'(Z) + 1 = 0$$

Coefficient of Z^i in the solution of above equation for $\Phi'(Z)$ gives ith element Φ'_i in the series Φ'. Solving the equation for $\Phi'(Z)$, we get

$$\Phi'(Z) = \frac{1 \pm \sqrt{(1 + Z)^2 - 8Z}}{4Z} = \frac{1 \pm \sqrt{1 - 6Z + Z^2}}{4Z}$$

However, $\frac{1+\sqrt{1-6Z+Z^2}}{4Z}$ can not be the solution as it does not satisfy the initial condition $\Phi'_0 = 1$. Expanding $\frac{1-\sqrt{1-6Z+Z^2}}{4Z}$ as $\sum_i \Phi'_i Z^i$ gives coefficients $\Phi'_i = \Phi_{i+1}$.

n	1	2	3	4	5	6	7	8	9	10	11	12
Φ_n	1	1	3	11	45	197	903	4279	20793	103049	518859	2646723

Table 3.1: Number of alphabetic trees.

Using binomial expansion $\sqrt{1-6Z+Z^2} = \sum_k \begin{pmatrix} 1/2 \\ k \end{pmatrix} Z^k(Z-6)^k$. Coefficients of Z^i in this equation are given by $\sum_{j=0}^{\lfloor i/2 \rfloor} \begin{pmatrix} 1/2 \\ i-j \end{pmatrix}$ {Coefficient of Z^j in $(Z-6)^{i-j}$}. (Substituting $i-j$ for k). Using binomial expansion, this is equal to

$$\Phi''_i = \sum_{j=0}^{\lfloor i/2 \rfloor} \begin{pmatrix} i-j \\ j \end{pmatrix} (-6)^{i-2j} \begin{pmatrix} 1/2 \\ i-j \end{pmatrix} \qquad (3.9)$$

Hence, $\Phi'(Z) = Z^{-1}/4 - \sum_i \Phi''_i Z^{i-1}/4$. The first term only affects the -1st element, and from the second term $-\Phi''_i/4$ is Φ'_{i-1}. However, since the sequence was initially shifted left, $\Phi'_{i-1} = \Phi_i = -\Phi''_i/4$. ∎

The number of distinct alphabetic trees on 1 to 12 leaf nodes derived using equation (3.8) is given in Table 3.1.

This sequence is identical to the sequence generated for dissection of a polygon by Motzkin [72]. The same sequence is also derived in [88] while solving equations for a pair of inverse series without reference to any particular application. The formula proposed there is given below

$$\Phi_n = \sum_{k=0}^{n} (-1)^j \begin{pmatrix} 2n-j \\ j \end{pmatrix} \begin{pmatrix} 2n-2j \\ n-j \end{pmatrix} \frac{2^{n-j}}{n-j+1} \qquad (3.10)$$

Indeed, I have formally shown that equations (3.8) and (3.10) are equivalent. Motzkin has also shown that for large n $\Phi_{n+1}/\Phi_n \to 5.83$, implying that Φ_n is $O(6^n)$.

It is interesting to note that if alphabetic trees are restricted to be binary, equation (3.3) is reduced to

$$\Psi_{i,i+d} = \bigcup_{j=0}^{d-1} \wedge(\Psi_{i,i+j} \times \Psi_{i+j+1,i+d}) \qquad \text{for } d \geq 1$$

This gives rise to a simpler convoluted recurrence equation of the form $\Phi_n = \sum_{j=1}^{n-1} \Phi_{n-j}\Phi_j$ for $n \geq 1$ whose solution gives the Catalan numbers. Thus, the number of alphabetic binary trees on n leaf nodes is given by

$$\begin{pmatrix} 2n-2 \\ n-1 \end{pmatrix} \frac{1}{n} \approx \frac{4^{n-1}}{\sqrt{\pi}(n-1)^{3/2}} \qquad \text{for large } n. \tag{3.11}$$

Cayley [10], and Gilbert and Moore [31] have independently derived expressions for the number of alphabetic binary trees that are equivalent to the Catalan numbers.

3.3.1 Bounded Height Trees

Height of a tree is the maximum number of edges on the path from root to any leaf node. Next, alphabetic trees with height restriction h are enumerated. The tree enumeration equation for bounded height alphabetic trees is derived from equation (3.3) as

$$\begin{aligned} \Psi^{h:\infty}_{i,i+d} = & \bigcup_{j=0}^{d-2} (\wedge(\Psi^{h-1:\infty}_{i,i+j} \times \Psi^{h-1:\infty}_{i+j+1,i+d}) \cup \wedge(\Psi^{h-1:\infty}_{i,i+j} \times (.\dot{\div}.\Psi^{h:\infty}_{i+j+1,i+d}))) \\ & \bigcup \wedge(\Psi^{h-1:\infty}_{i,i+d-1} \times \Psi^{h-1:\infty}_{i+d,i+d}) \qquad \text{for } d \geq 1 \end{aligned} \tag{3.12}$$

Correctness of the equation follows from the observation that alphabetic trees with maximum height h can be generated by JOINing all alphabetic trees with maximum height $h-1$. The number of such trees is computed using the following recursive difference equation:

# of leaves n	1	2	3	4	5	6	7	8	9	10
$\Phi_n^{1:\infty}$	1	1	1	1	1	1	1	1	1	1
$\Phi_n^{2:\infty}$	1	1	3	7	15	31	63	127	255	511
$\Phi_n^{3:\infty}$	1	1	3	11	37	117	357	1065	3129	9097
$\Phi_n^{4:\infty}$	1	1	3	11	45	181	703	2659	9875	36195
$\Phi_n^{5:\infty}$	1	1	3	11	45	197	871	3799	16297	68953
$\Phi_n^{6:\infty}$	1	1	3	11	45	197	903	4215	19673	91145
$\Phi_n^{7:\infty}$	1	1	3	11	45	197	903	4279	20665	100489
$\Phi_n^{8:\infty}$	1	1	3	11	45	197	903	4279	20793	102793
$\Phi_n^{9:\infty}$	1	1	3	11	45	197	903	4279	20793	103049

Table 3.2: Number of alphabetic trees with bounded height.

$$\Phi_n^{h:\infty} = \begin{cases} \sum_{j=1}^{n-2} (\Phi_j^{h-1:\infty}(\Phi_{n-j}^{h-1:\infty} + \Phi_{n-j}^{h:\infty})) + \Phi_{n-1}^{h-1:\infty} & \text{for } n \geq 2; h < n \\ 1 & \text{for } n = 1 \text{ or } h = 1 \end{cases} \tag{3.13}$$

This is a non-linear multi-variable recurrence equation. It is difficult to obtain a closed form solution to this equation. Indeed, no closed form solutions to such equations are known [24]. However, using the above equation for counting the number of 2-level to 10-level trees result in sequences shown in Table 3.2. Similar sequences can be derived for other height bounds.

Using $\Phi_n^{h:\infty}$ and $\Phi_n^{h-1:\infty}$, the number of alphabetic trees with exact height h can also be derived, i.e., $\Phi_n^{[h]:\infty} = \Phi_n^{h:\infty} - \Phi_n^{h-1:\infty}$ with the initial condition $\Phi_n^{[1]:\infty} = \Phi_n^{1:\infty}$. The number of alphabetic trees with exact height for upto 10 leaf nodes is given in Table 3.3.

Maximum height of a tree with n leaf nodes is $n-1$. Hence, the number of alphabetic trees on n leaf nodes is also given by

$$\sum_{h=1}^{n-1} \Phi_n^{[h]:\infty} = \sum_{h=1}^{n-1} (\Phi_n^{h:\infty} - \Phi_n^{h-1:\infty}) + \Phi_n^{1:\infty} = \Phi_n^{n-1:\infty} \tag{3.14}$$

# of leaves n	1	2	3	4	5	6	7	8	9	10
$\Phi_n^{[1]:\infty}$	1	1	1	1	1	1	1	1	1	1
$\Phi_n^{[2]:\infty}$	0	0	2	6	14	30	62	126	254	510
$\Phi_n^{[3]:\infty}$	0	0	0	4	22	86	294	938	2874	8586
$\Phi_n^{[4]:\infty}$	0	0	0	0	8	64	346	1594	6746	27098
$\Phi_n^{[5]:\infty}$	0	0	0	0	0	16	168	1140	6422	32758
$\Phi_n^{[6]:\infty}$	0	0	0	0	0	0	32	416	3376	22192
$\Phi_n^{[7]:\infty}$	0	0	0	0	0	0	0	64	992	9344
$\Phi_n^{[8]:\infty}$	0	0	0	0	0	0	0	0	128	2304
$\Phi_n^{[9]:\infty}$	0	0	0	0	0	0	0	0	0	256

Table 3.3: Number of alphabetic trees with exact height.

3.3.2 Bounded Degree Trees

Some applications require that internal nodes have no more than t children. The corresponding enumeration equation is

$$\begin{aligned}\Psi_{i,i+d}^{\infty:r,t} &= \bigcup_{j=0}^{d-2} \left(\wedge(\Psi_{i,i+j}^{\infty:t} \times \Psi_{i+j+1,i+d}^{\infty:t}) \cup \wedge(\Psi_{i,i+j}^{\infty:t} \times (.\doteq.\Psi_{i+j+1,i+d}^{\infty:r-1,t}))\right) \\ &\quad \bigcup \wedge(\Psi_{i,i+d-1}^{\infty:t} \times \Psi_{i+d,i+d}^{\infty:t}) \quad \text{for } d \geq 1;\ \ 2 \leq r \leq t \leq d+1 \end{aligned} \quad (3.15)$$

Note that $\Psi_{i,i+d}^{\infty:r,t} = \text{NULL}$ set for $r = 1$ or $t = 1$.

Hence,

$$\Phi_n^{\infty:r,t} = \begin{cases} \sum_{j=1}^{n-2} \left(\Phi_j^{\infty:t}(\Phi_{n-j}^{\infty:t} + \Phi_{n-j}^{\infty:r-1,t})\right) + \Phi_{n-1}^{\infty:t} & \text{for } n \geq 2;\ \ 2 \leq r \leq t \\ 0 & \text{for } r = 1 \\ 1 & \text{for } n = 1 \text{ and } r \neq 1 \end{cases} \quad (3.16)$$

Again, we are not aware of a closed form solution for this multi-variable recurrence equation. However, the above equation can be used to derive the number of

# of leaves n	1	2	3	4	5	6	7	8	9	10
${\Phi_n}^{2,\infty}$	1	1	2	5	14	42	132	429	1430	4862
${\Phi_n}^{3,\infty}$	1	1	3	10	38	154	654	2871	12925	59345
${\Phi_n}^{4,\infty}$	1	1	3	11	44	189	850	3951	18832	91542
${\Phi_n}^{5,\infty}$	1	1	3	11	45	196	894	4215	20377	100463
${\Phi_n}^{6,\infty}$	1	1	3	11	45	197	902	4269	20717	102531
${\Phi_n}^{7,\infty}$	1	1	3	11	45	197	903	4278	20782	102960
${\Phi_n}^{8,\infty}$	1	1	3	11	45	197	903	4279	20792	103037
${\Phi_n}^{9,\infty}$	1	1	3	11	45	197	903	4279	20793	103048
${\Phi_n}^{10,\infty}$	1	1	3	11	45	197	903	4279	20793	103049

Table 3.4: Number of alphabetic trees with degree restriction.

alphabetic trees if the number of leaf nodes and degree restrictions are known as shown in Table 3.4.

3.3.3 Bounded Height and Bounded Degree Trees

In the most general setting, alphabetic trees with both degree and height restriction can be enumerated. Corresponding enumeration equation and recurrence equations are as given below.

$$\begin{aligned}\Psi_{i,i+d}^{h:r,t} &= \bigcup_{j=0}^{d-2} (\wedge(\Psi_{i,i+j}^{h-1:t} \times \Psi_{i+j+1,i+d}^{h-1:t}) \cup \wedge(\Psi_{i,i+j}^{h-1:t} \times (.\dot{\div}.\Psi_{i+j+1,i+d}^{h-1:r-1,t}))) \\ &\bigcup \wedge(\Psi_{i,i+d-1}^{h-1:t} \times \Psi_{i+d,i+d}^{h-1:t}) \quad \text{for } d \geq 1; 2 \leq r \leq t \leq d+1; h < n \end{aligned} \tag{3.17}$$

Again, $\Psi_{i,i+d}^{h:r,t} = \text{NULL}$ set for $r = 1$ or $t = 1$.

$$\Phi_n^{h:r,t} = \begin{cases} \sum_{j=1}^{n-2} (\Phi_j^{h-1:t}(\Phi_{n-j}^{h-1:t} + \Phi_{n-j}^{h:r-1,t})) + \Phi_{n-1}^{h-1:t} & \text{for } n \geq 2; 2 \leq r \leq t; h \geq 2 \\ 0 & \text{for } r = 1 \\ 1 & \text{for } h = 1 \text{ or } n = 1 \text{ when } r \neq 1 \end{cases} \tag{3.18}$$

Height	Degree								
	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1
2	9	45	129	255	381	465	501	510	511
3	100	1869	5362	7700	8695	9012	9085	9096	9097
4	634	13222	27790	33885	35685	36106	36183	36194	36195
5	1814	32425	58054	66383	68435	68864	68941	68952	68953
6	3182	48705	79670	88559	90627	91056	91133	91144	91145
7	4222	56849	88982	97903	99971	100400	100477	100488	100489
8	4734	59089	91286	100207	102275	102704	102781	102792	102793
9	4862	59345	91542	100463	102531	102960	103037	103048	103049

Table 3.5: Number of alphabetic trees with degree and height restriction on 10 leaf nodes.

It should be noted that solution to above equation will automatically provide solutions to equation (3.7), equation (3.13) and equation (3.16) as special cases. Table 3.5 shows the number of alphabetic trees under different height and degree constraints for $n = 10$.

3.4 Alphabetic Tree Optimization

Alphabetic tree optimization implies finding the "best" alphabetic tree given the combining function and the tree cost function. Formally, the problem of alphabetic tree optimization can be defined as given below. Here, without loss of generality, it is assumed that the tree cost function is to be minimized.

Problem 3.4.1 ALPHABETIC_TREE_OPTIMIZATION

Instance: *A set of* n *ordered and weighted leaf nodes* $(L_1, L_2, \ldots, L_n)$ *with corresponding weights* W_{L_i} *and a combining function* F *that combines* $t \geq 2$ *nodes to generate an internal node* I_p *with weight* $W_{I_p} = F(W_{child_{I_{p,1}}}, \ldots, W_{child_{I_{p,t}}})$ *and a tree cost function* $C_T = C(W_{L_1}, \ldots, W_{L_n})$.

Problem: *Generate a minimum cost tree that has no internal edge crossing.*

The following algorithm enumerates all alphabetic trees and selects the best alphabetic tree with respect to the given tree cost function.

Algorithm 3.1 GenBestAlpTree ($\mathcal{N}$)
$\mathcal{N}$ is a set of n leaf nodes with weights
begin
1 **for** $d = 0$ to $n-1$ **do**
2 **for** $i = 1$ to $n-d$ **do**
3 **if** $d = 0$ $\Psi_{i,i}$ =SingleNodeTree(i)
4 **else**
5 **for** $l = 1$ to d $\Psi_{l:i,i+d} =$ NULL
6 **for** $j = 0$ to $d-1$ **do**
7 **ForEachElement** $F_{1:i,i+j}$ of $\Psi_{1:i,i+j}$ **do**
8 **for** $l = 1$ to $d-j$ **do**
9 **ForEachElement** $F_{l:i+j+1,i+d}$ of $\Psi_{l:i+j+1,i+d}$ **do**
10 $\Psi_{l+1:i,i+d} = \Psi_{l+1:i,i+d} \bigcup \{F_{1:i,i+j} \cup F_{l:i+j+1,i+d}\}$
11 $\Psi_{i,i+d} = \Psi_{i,i+d} \bigcup \{\wedge(F_{1:i,i+j} \cup F_{l:i+j+1,i+d})\}$
12 Result = FindBest($\Psi_{1,n}$)
end

Here d corresponds to the number of leaf nodes being considered in the main loop of the algorithm, i corresponds to the first leaf node of the set of leaf nodes being considered in the current loop. Thus, $i+d$ corresponds to the last leaf node in the set of leaf nodes being considered in the current loop. j corresponds to the number of leaf nodes under the leftmost branch. Thus, $d-j$ corresponds to the number of leaf nodes under all other branches of the tree. l corresponds to the number of siblings of the leftmost branch. The braces used in Line 10 of the signify a set generation operation. Line 10 results in $l+1$-tree forests by generating a set from the union of 1-tree forests on leaf nodes i to $i+j$ with l-tree forests on leaf nodes $i+j+1$ to $i+d$. Line 11 generates the corresponding 1-tree forest by JOINing these $l+1$-tree forest. The algorithm can be easily modified to generate optimal alphabetic trees with each internal node having at most t children by restricting $l < t$. With

small modifications, a generic algorithm for bounded-height and bounded-degree alphabetic tree enumeration can be obtained using equation (3.18).

Lemma 3.4.1 *The time and space complexity of Algorithm 3.1 are $O(n^3 6^{n+1})$ and $O(n^3 6^n)$, respectively.*

Proof Motzkin has shown that for large n, $\Phi_{n+1}/\Phi_n \to 5.83$. Hence, Φ_n is $O(6^n)$. In Line 7 of the algorithm, there are $O(6^{j+1})$ elements in $\Psi_{1:i,i+j}$. Likewise, there are totally $O(6^{d-j})$ elements in the right branches of the tree (corresponding to Line 8, 9, 10 and 11). Thus, the total time complexity of inner loop corresponding to Line 7-11 is $O(6^{d+1})$. Hence, the time complexity can be calculated as

$$\sum_{d=1}^{n-1}[\sum_{i=1}^{n-d}(\sum_{j=0}^{d-1}6^{d+1}+d)]+n=\sum_{d=1}^{n-1}(n-d)[d6^{d+1}+d^2]=O(n^3 6^{n+1})$$

Since there are $O(n^2)$ ordered subsets of leaf nodes (e.g., $n-1$ two leaf node sets, $n-2$ three leaf node sets, etc..), assuming that the information about an n leaf tree can be stored in $O(n)$ space, a naive implementation of this algorithm requires space complexity $O(n^3 6^n)$ for maintaining all alphabetic forests for every ordered subset of leaf nodes. ■

In general, depending on the combining and tree cost functions, complexity of determining the best tree may be reduced by considering only a subset of all tree structures that are non-inferior with respect to each other for the purpose of optimizing the tree cost. In particular, if the tree is *subtree optimal*, as is defined next, optimal cost alphabetic binary trees can be found in polynomial time for arbitrary tree cost functions. Similarly, optimal cost alphabetic nonbinary tree can be found in polynomial time if the tree is *subforest optimal* as is defined next.

3.4.1 Subtree and Subforest Optimality

Given a set of internal nodes $\mathcal{I}=\{I_1,I_2,\ldots,I_m\}$ and leaf nodes $\mathcal{L}=\{L_1,L_2,\ldots,L_n\}$ of a tree where I_m is the root of the tree, the cost of the tree is given by $C_T=C(W_{L_1},\ldots,W_{L_n})$ where W_N denotes weight of leaf node N. The objective is to solve problem 3.4.1. Consider two trees T and T' on leaf nodes $\{L_1,\ldots,L_n\}$. Assume that these trees only differ in subtrees rooted at internal node I, that is, T and T'

are identical except for differences between subtrees T_I and T'_I. Then, the tree is **subtree optimal** (ST-optimal) if a *subtree cost function* H can be defined for every such T_I and T'_I such that; $H_{T_I} > H_{T_{I'}} \Rightarrow C_T \geq C_{T'}$.

In other words, a tree is ST-optimal if the tree *cost* is monotone non-decreasing in subtree cost of each of its subtrees, that is, if the subtree cost of some subtree is increased (decreased), the tree cost can not decrease (increase).

Conditions for ST-optimality are

1. The tree cost function C_T is decomposable in terms of subtree cost function of each of its subtrees, i.e., $C_T = G(H_{T_I}, W_{\mathcal{L}})\ \ \forall I \in \mathcal{I}$.

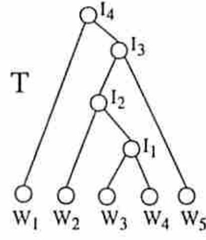
2. Function G is independent of the tree structure T_I at node I.

3. Function G is monotone non-decreasing in H_{T_I}.

This allows independent optimization of $H_{T_I}\ \ \forall I \in \mathcal{I}$ using dynamic programming. Hence, to determine whether the given tree optimization problem is ST-optimal, a function H_T satisfying the above conditions needs to be identified. Fortunately, for most ST-optimal trees, the tree cost is easily decomposable in terms of the tree cost of its subtrees (i.e., $H_T = C_T$). In Figure 3.5, let $\{L_1, L_2, L_3, L_4, L_5\}$ be the set of leaf nodes with weights $\{W_{L_1}, W_{L_2}, W_{L_3}, W_{L_4}, W_{L_5}\}$. Two alphabetic binary tree structures are shown in the figure. Both additive tree cost and minimax tree costs allow decomposition of the tree cost in terms of tree costs of its subtrees, resulting in ST-optimality of the additive and minimax tree cost functions.

The implication of ST-optimality is that if a tree is ST-optimal, with a dynamic programming based approach, only the optimal subtrees for each subset of leaf nodes need to be maintained. This is sufficient to reduce the time complexity of alphabetic binary tree optimization from exponential to polynomial as described in Section 3.4.2. However, for non-binary trees, this is not sufficient to reduce exponential time complexity to polynomial. Polynomial runtimes are achieved if the tree is subforest optimal as is described next.

Consider an internal node I of a tree T with V immediate children and leaf support $\{L_i, \ldots, L_{i+d}\}$. Consider any subset $\mathcal{D}$ of these V children such that these children have continuous leaf support $\{L_{i+j}, \ldots, L_{i+k}\}$ where $0 \leq j \leq k \leq d$. Let $D = |\mathcal{D}|$ where $1 \leq D \leq V$. Denote the corresponding tree structures



Additive Tree Cost function:

$C_T = W_1 + 3W_2 + 4W_3 + 4W_4 + 2W_5$

$C_{T_{I2}} = W_2 + 2W_3 + 2W_4$

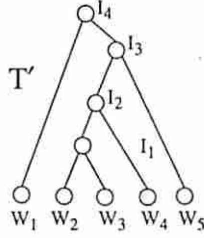
$C_T = G(C_{T_{I2}}, \mathcal{L}) = C_{T_{I2}} + W_1 + 2(W_2 + W_3 + W_4 + W_5)$

Minimax Tree Cost Function:

$C_T = max(W_1, max(max(W_2, max(W_3, W_4) + 1) + 1, W_5) + 1) + 1$

$C_{T_{I2}} = max(W_2, max(W_3, W_4) + 1) + 1$

$C_T = G(C_{T_{I2}}, \mathcal{L}) = max(max(C_{T_{I2}}, W_5) + 1, W_1) + 1$



Additive Tree Cost function:

$C_{T'} = W_1 + 4W_2 + 4W_3 + 3W_4 + 2W_5$

$C_{T'_{I2}} = 2W_2 + 2W_3 + 4W_4$

$C_{T'} = G(C_{T'_{I2}}, \mathcal{L}) = C_{T'_{I2}} + W_1 + 2(W_2 + W_3 + W_4 + W_5)$

Minimax Tree Cost Function:

$C_{T'} = max(W_1, max(max(max(W_2, W_3) + 1, W_4) + 1, W_5) + 1) + 1$

$C_{T'_{I2}} = max(max(W_2, W_3) + 1, W_2) + 1$

$C_{T'} = G(C_{T'_{I2}}, \mathcal{L}) = max(max(C_{T'_{I2}}, W_5) + 1, W_1) + 1$

Figure 3.5: Examples of ST-optimal tree cost functions.

as $T_1, T_2, \ldots, T_D$ with tree costs $C_{T_1}, C_{T_2}, \ldots, C_{T_D}$, respectively. This set of trees are defined as a **subforest** of the original tree T. Now, consider another tree T' that has identical tree structure except for the tree structures rooted at these D children of internal node I. The corresponding tree structures are denoted as $T'_1, T'_2, \ldots, T'_D$ with tree costs $C_{T'_1}, C_{T'_2}, \ldots, C_{T'_D}$. Then, the tree is **subforest optimal** (SF-optimal) if a *subforest cost function* H can be defined such that; $H(C_{T_1}, C_{T_2}, \ldots, C_{T_D}) > H(C_{T'_1}, C_{T'_2}, \ldots, C_{T'_D}) \Rightarrow C_T \geq C_{T'}$.

In other words, a tree is SF-optimal if the tree cost is monotone non-decreasing in subforest cost of each of its subforests, that is, if the subforest cost of some subforest is increased (decreased), the tree cost can not decrease (increase).

Conditions for SF-optimality are

1. The tree cost function C_T is decomposable in terms of the subforest cost function of each of its subforests, i.e., $C_T = G(H(C_{T_1}, C_{T_2}, \ldots, C_{T_D}), W_{\mathcal{L}})$ for all subforests $\mathcal{D}$ of T.

2. Function G is independent of the subforest structure of $\mathcal{D}$.

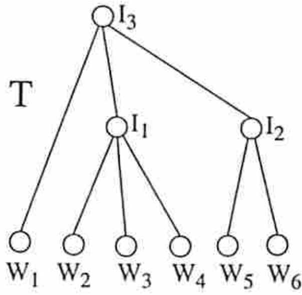
3. Function G is monotone non-decreasing in $H(C_{T_1}, C_{T_2}, \ldots, C_{T_D})$.

SF-optimality is a generalization of ST-optimality with ST-optimality being a special case of SF-optimality when $D = 1$. Indeed, a stronger definition of SF-optimality can be proposed that distinguishes between two forests of any number of trees on the same set of leaf nodes.

It is interesting to note here that conditions of SF-optimality are subsumed by the *principle of optimality* (see [65], for example) used in characterizing the decomposable problems for dynamic programming approach. Hence, conditions for SF-optimality can be viewed as specialization of the principles of optimality for tree optimization.

Examples of SF-optimal trees include the original Huffman trees with the additive cost function as shown in Figure 3.6. Minimax trees are also SF-optimal. A variation of Minimax trees, namely, trees with combining function $F(W_1, \ldots, W_n) = Max(W_1, \ldots, W_n) + n$ are SF-optimal. These minimax trees are very important in many applications, specially in logic synthesis where this latter combining function corresponds to the unit fanout delay model. SF-optimality of unit fanout delay model has allowed me to propose an optimal alphabetic fanout tree in polynomial time. An example of the minimax combining function with unit fanout delay model is shown in Figure 3.6.

Note that ST-optimality (or SF-optimality) is a property of the tree optimization problem being solved. Indeed, the combining functions and the tree cost functions determine ST-optimality or SF-optimality of trees generated and the underlying tree optimization problem. It should also be pointed out that ST-optimality and SF-optimality apply to alphabetic as well as non-alphabetic trees. In essence, ST-optimality characterizes binary trees that permit the use of dynamic programming approach while guaranteeing time complexity of $O(n^3)$ or better while strong SF-optimality characterizes non-binary trees that permit use of dynamic programming approach while guaranteeing time complexity of $O(n^3)$ or better. Trees that are SF-optimal but are not strongly SF-optimal allow use of dynamic programming approach while guaranteeing time complexity of $O(n^4)$ or better. The time and space complexity issues for ST-optimal and SF-optimal trees are discussed next.



Additive Trees:

$$C_T = W_1 + 2W_2 + 2W_3 + 2W_4 + 2W_5 + 2W_6$$

$$H(C_{T_{I1}}, C_{T_{I2}}) = (W_2 + W_3 + W_4) + (W_5 + W_6)$$

$$C_T = G(H(C_{T_{I1}}, C_{T_{I2}}), \mathcal{L})$$

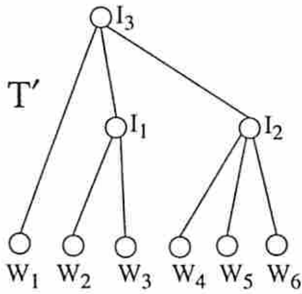
$$= H(C_{T_{I1}}, C_{T_{I2}}) + W_1 + W_2 + W_3 + W_4 + W_5 + W_6$$

Minimax Trees (With unit fanout delay):

$$C_T = max(W_1, max(W_2, W_3, W_4) + 3, max(W_5, W_6) + 2) + 3$$

$$H(C_{T_{I1}}, C_{T_{I2}}) = max(max(W_2, W_3, W_4) + 3, max(W_5, W_6) + 2)$$

$$C_T = G(H(C_{T_{I1}}, C_{T_{I2}}), \mathcal{L}) = max(W_1, G(C_{T_{I1}}, C_{T_{I2}})) + 3$$



Additive Trees:

$$C_{T'} = W_1 + 2W_2 + 2W_3 + 2W_4 + 2W_5 + 2W_6$$

$$H(C_{T'_{I1}}, C_{T'_{I2}}) = (W_2 + W_3) + (W_4 + W_5 + W_6)$$

$$C_{T'} = G(H(C_{T'_{I1}}, C_{T'_{I2}}), \mathcal{L})$$

$$= H(C_{T'_{I1}}, C_{T'_{I2}}) + W_1 + W_2 + W_3 + W_4 + W_5 + W_6$$

Minimax Trees (With unit fanout delay):

$$C_{T'} = max(W_1, max(W_2, W_3, W_4) + 3, max(W_5, W_6) + 2) + 3$$

$$H(C_{T'_{I1}}, C_{T'_{I2}}) = max(max(W_2, W_3) + 2, max(W_4, W_5, W_6) + 3)$$

$$C_{T'} = G(H(C_{T'_{I1}}, C_{T'_{I2}}), \mathcal{L}) = max(W_1, H(C_{T'_{I1}}, C_{T'_{I2}})) + 3$$

Figure 3.6: Examples of SF-optimal tree cost functions.

3.4.2 ST-optimal Trees

Lemma 3.4.2 *For ST-optimal trees, there exists an optimal tree with optimal subtrees.*

Proof Since the tree is ST-optimal, any non-optimal subtree could be substituted by an optimal subtree without increasing the tree cost. ∎

Corollary 3.4.3 *To generate an optimal alphabetic tree using equation (3.3), it is sufficient to consider only optimal trees as arguments to the* JOIN *operator.*

Let $\theta_{i,i+j}$ denote the optimal tree on the set of leaves $(i, \ldots, i+j)$ and $St\Psi_{i+j+1,i+d}$ denote the collection of alphabetic trees on leaves $(L_{i+j+1}, \ldots, L_{i+d})$ such that for each tree in $St\Psi_{i+j+1,i+d}$, every subtree is optimal. Then, equations (3.3) and (3.7) can be rewritten as

$$\begin{aligned} St\Psi_{i,i+d} &= \bigcup_{j=0}^{d-2} \Big((\wedge\{\theta_{i,i+j}, \theta_{i+j+1,i+d}\}) \cup (\wedge(\{\theta_{i,i+j}\} \times (.\dot{\div}.St\Psi_{i+j+1,i+d}))) \Big) \\ & \bigcup (\wedge\{\theta_{i,i+d-1}, \theta_{i+d,i+d}\}) \qquad \text{for } d \geq 1 \end{aligned} \tag{3.19}$$

An illustration of equation (3.19) is given in Figure 3.7.

$$St\Phi_n = \begin{cases} \sum_{j=1}^{j-2} [St\Phi_{n-d} + 1] + 1 & \text{if } n \geq 2 \\ 1 \quad \text{if } n = 1 \end{cases} \tag{3.20}$$

Lemma 3.4.4 *The number of ST-optimal alphabetic trees on n leaves ($n \geq 2$), $St\Phi_n$, is equal to $2^{n-1} - 1$.*

Proof This is proved by induction.

- For two leaf nodes $St\Phi_2 = 2^{2-1} - 1 = 1$ which is true since there is only one tree structure on two leaves.
- Assuming the above equation is true for $i \geq 2$, I prove its correctness for $i+1$:

$$St\Phi_{i+1} = \sum_{d=1}^{i-1} [St\Phi_{i+1-d} + 1] = \sum_{d=1}^{i-1} [2^{i-d} - 1 + 1] + 1 = \sum_{d=0}^{i-1} [2^d] + 1 - 2^0 = 2^i - 1$$

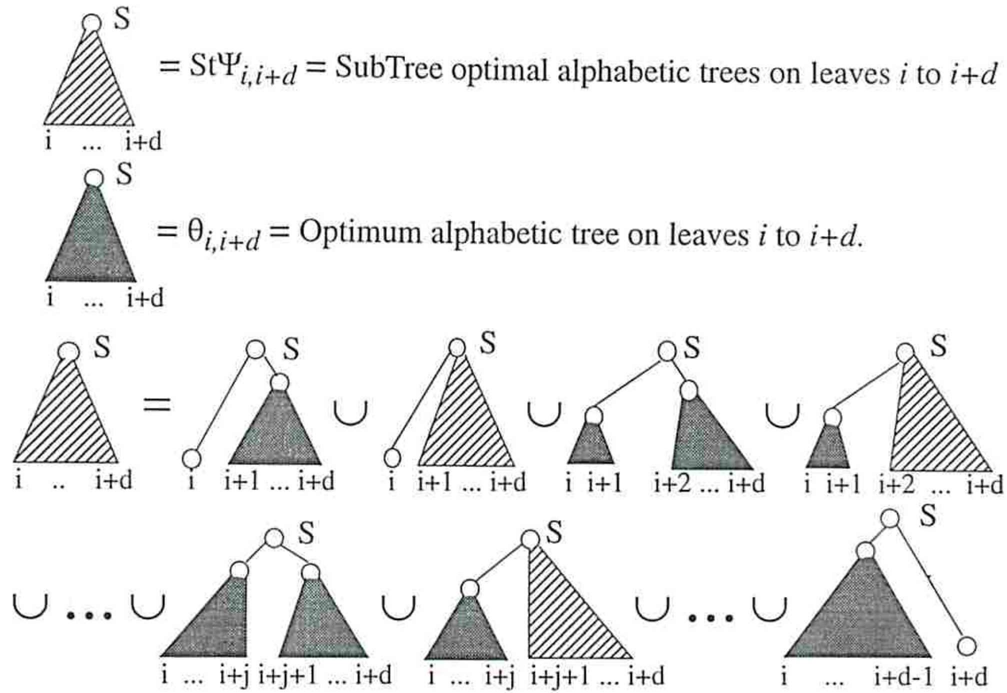


Figure 3.7: An illustration of ST-optimal alphabetic tree generation equation.

Thus, $St\Phi_n = 2^{n-1} - 1$. ∎

For all practical purposes, $St\Psi_{1,n}$ is the largest collection of trees any optimal alphabetic tree construction algorithm has to consider when the tree is ST-optimal. This leads to an upper bound on the complexity of any ST-optimal alphabetic tree problem. A corresponding algorithm is shown below.

Algorithm 3.2 GenBestAlpTree-STOptimal ($\mathcal{N}$)

$\mathcal{N}$ is a set of n leaf nodes with weights

begin

1 **for** d = 0 to $n-1$ **do**

2   **for** i = 1 to $n-d$ **do**

3   **if** $d=0$ $St\Psi_{i,i}$ =SingleNodeTree(i)

4   **else**

5       $\theta_{i,i+d}$ = NULL

6   **for** l = 1 to d $St\Psi_{l:i,i+d}$ = NULL

7   **for** j = 0 to $d-1$ **do**

8   **for** l = 1 to $d-j$ **do**

9   **ForEachElement** $F_{l:i+j+1,i+d}$ of $St\Psi_{l:i+j+1,i+d}$ **do**

10             $St\Psi_{l+1:i,i+d} = St\Psi_{l+1:i,i+d} \bigcup \{\{\theta_{i,i+j}\} \cup F_{l:i+j+1,i+d}\}$

11             $\theta_{i,i+d}$ = ChooseBest($\theta_{i,i+d}, \wedge(\{\theta_{i,i+j}\} \cup F_{l:i+j+1,i+d})$)

12 Result = $\theta_{1,n}$

end

Algorithm 3.2. is similar to Algorithm 3.1. However, Algorithm 3.2 does not generate through each element of $St\Psi_{i,i+j}$ corresponding to Line 7 of Algorithm 3.1 because $St\Psi_{1:i,i+j} = \{\{\theta_{i,i+j}\}\}$. Apart from this, in Line 11 Algorithm 3.2 maintains only the best tree while the corresponding line in Algorithm 3.1 maintained a collection of 1-tree forests.

Lemma 3.4.5 *The time and space complexities of Algorithm 3.2 are $O(2^{n+1})$ and $O(n^3 2^{n-1})$, respectively.*

Proof The exponential time complexity of this algorithm is due to exponentially large ways of selecting optimal subtrees to generate subforests at an internal node, i.e., $|St\Psi_{i+j+1,i+d} \cup .\dot{=}. St\Psi_{i+j+1,i+d}| = \sum_{l=1}^{d-j} |St\Psi_{l:i+j+1,i+d}|$. The number of subforests on $d-j-1$ leaf nodes is also given by the number of ways an ordered set of $d-j-1$ elements can be partitioned. This is shown in Figure 3.8. When the left most branch of the tree is already determined, there are $d-j-1$ ways of generating 3-tree

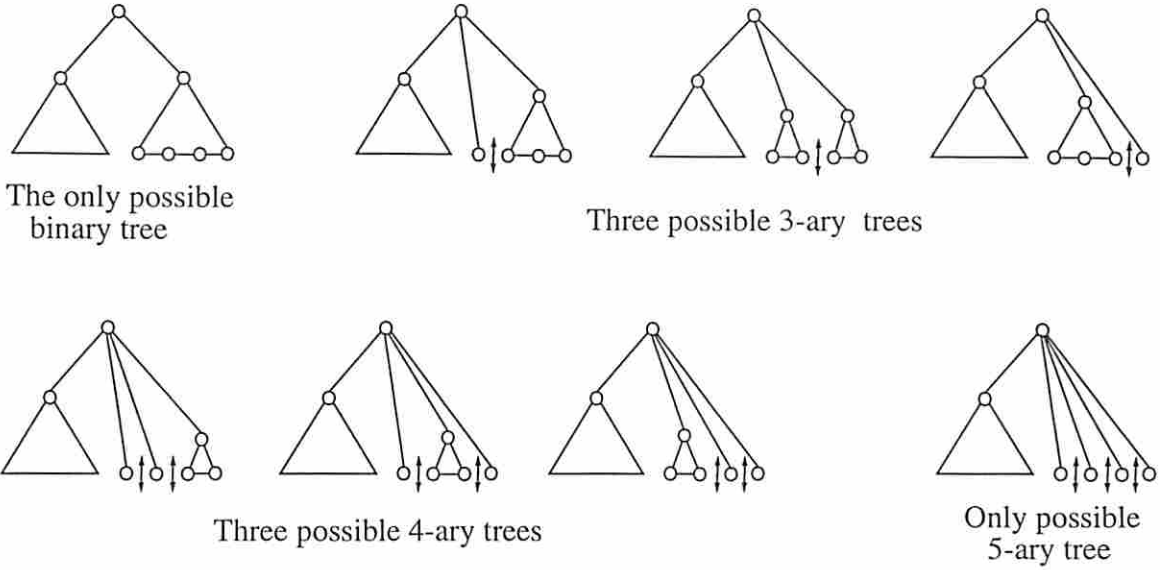


Figure 3.8: Example showing 8 possible ways to generate alphabetic trees with four leaf nodes under right branches.

subforests with optimal alphabetic trees, $(d-j-1)(d-j-2)/2$ ways of generating 4-tree subforests with optimal alphabetic trees, etc.. The recurrence equation for this problem is $\Phi'_{d-j} = \sum_{i=1}^{d-j-1} \Phi_i' + 1$ with the solution of $\Phi'_{d-j} = 2^{d-j-1}$, giving the runtime of inner most loops and the number of subforests to be maintained on every ordered subset of leaf nodes. Hence, the time complexity of Algorithm 3.2 is

$$\sum_{d=1}^{n-1} [\sum_{i=1}^{n-d} (\sum_{j=0}^{d-1} 2^{d-j-1} + d)] + n = 2^{n+1} + 5n^3/6 - 3n^2/2 + 2n/3 - 2 = O(2^{n+1})$$

Again, since there are $O(n^2)$ ordered subsets of leaf nodes, assuming that the information about an n leaf tree can be stored in $O(n)$ space, a naive implementation of this algorithm requires space complexity $O(n^3)$ for maintaining optimal subtrees for every ordered subset of leaf nodes. However, for each ordered subset of leaf nodes, a list of subforests consisting of ST-optimal subtrees needs to be maintained. Since each subtree is optimal, only a list of pointers to corresponding optimal subtrees need to be kept, requiring $O(n)$ space per subforest. As explained above, since there are 2^{n-1} ways to partition an ordered set of n nodes, and since since each partition corresponds to a subforest, the space complexity of the algorithm is $O(n^3 2^{n-1})$. ∎

Thus, for non-binary trees, ST-optimality of the trees has improved the run time and space complexity significantly although still not reducing them to polynomial. However, for alphabetic binary trees, ST-optimality is sufficient to reduce the run times and space complexity to polynomial as will be described next.

3.4.3 ST-optimal Binary Trees

Lemma 3.4.6 *The number of ST-optimal alphabetic binary trees on n leaf nodes is equal to $n-1$.*

Proof For ST-optimal binary trees, $\wedge(\{\theta_{i,i+j}\} \times (.\dot{\div}.St\Psi_{i+j+1,i+d}))$ drops out of equation (3.19), thus

$$Bi\Psi_{i,i+d} = \bigcup_{j=0}^{d-1} (\wedge\{\theta_{i,i+j}, \theta_{i+j+1,i+d}\}) \qquad \text{for } d \geq 1 \tag{3.21}$$

where $Bi\Psi_{i,i+d}$ denote the collection of apropos alphabetic binary trees. This gives

$$\Phi_n = \begin{cases} n-1 & \text{if } n \geq 2 \\ 1 & \text{if } n = 1 \end{cases} \tag{3.22}$$

■

Lemma 3.4.6 can be exploited to generate optimal alphabetic binary trees in $O(n^3)$ time complexity as follows.

```
Algorithm 3.3 GenBestAlpBinaryTree-STOptimal (𝒩)
𝒩 is given set of n leaf nodes with weights
begin
1  for d = 0 to n − 1 do
2    for i = 1 to n − d do
3      BiΨ_{i,i+d} = NULL
4      θ_{i,i+d} = NULL
5      if d = 0 BiΨ_{i,i} =SingleNodeTree(i)
6      else
7        for k = 0 to d − 1 do
8          BiΨ_{i,i+d} = BiΨ_{i,i+d} ⋃ ∧{θ_{i,i+k} ∪ θ_{i+k+1,i+d}}
9          θ_{i,i+d} = ChooseBest(θ_{i,i+d}, ∧{θ_{i,i+k} ∪ θ_{i+k+1,i+d}})
10  Result = θ_{1,n}
end
```

For each i, $n-i$ optimal binary trees are generated on subset of i leaf nodes. This has to be done n times; thus, time complexity of Algorithm 3.3 is

$$\sum_{i=1}^{i=n-1} i(n-i) = n^3/6 - n/6 = O(n^3).$$

Since for each of the $n(n-1)/2$ ordered subsets, only two pointers need to be maintained (one pointing to the ordered subset that corresponds to the best right subtree and the other pointing to the best left subtree), the space complexity of the algorithm is $O(n^2)$.

As mentioned in Section 3.3, the number of distinct alphabetic binary trees on n leaf nodes is given by the Catalan numbers that are $O(4^n)$. Subtree optimality of tree cost reduces the number of trees to be considered significantly, resulting in a polynomial time algorithm for finding an optimal alphabetic binary tree. This remains true irrespective of other characteristics of weights, tree cost function and combining functions. Previous researchers [46, 44, 31, 57, 54, 19] have generated optimal alphabetic binary trees, restricting leaf weights and/or some parameters of combining function to be integer, or only with respect to a specific tree cost

function. Algorithm 3.3 generates optimal alphabetic trees for ST-optimal trees in $O(n^3)$ without any such restriction. This run time can be further reduced to $O(n^2)$ for certain tree cost functions satisfying the *monotonicity* principle proposed by Knuth in [58]. Monotonicity property of a tree cost function guarantees that for optimal alphabetic trees, the leaf support of the left most branch of the root will not decrease if an additional leaf node is introduced to the right of all leaf nodes (and vice versa). The corresponding algorithm is given below.

Algorithm 3.4 GenBestAlpBinaryTree-STOptimalMonotone ($\mathcal{N}$)
$\mathcal{N}$ is given set of n leaf nodes with weights
begin
1 **for** d = 0 to $n-1$ **do**
2 **for** i = 1 to $n-d$ **do**
3 $Bi\Psi_{i,i+d}$ = NULL
4 $\theta_{i,i+d}$ = NULL
5 **if** $d=0$ $Bi\Psi_{i,i}$ =SingleNodeTree(i)
6 **else**
7 StartIndexOffset = IndexOfRightMostLeafOfLeftSubTree($\theta_{i,i+d-1}$) $-i$
8 StopIndexOffset = IndexOfLeftMostLeafOfRightSubtree($\theta_{i+1,i+d}$) $-i$
9 **for** k = StartIndexOffset to StopIndexOffset **do**
10 $Bi\Psi_{i,i+d} = Bi\Psi_{i,i+d} \bigcup \wedge\{\theta_{i,i+k} \cup \theta_{i+k+1,i+d}\}$
11 $\theta_{i,i+d}$ = ChooseBest($\theta_{i,i+d}, \wedge\{\theta_{i,i+k} \cup \theta_{i+k+1,i+d}\}$)
12 Result = $\theta_{1,n}$
end

The class of ST-optimal trees is the most general characterization of tree cost functions for which optimal alphabetic binary trees can be generated in $O(n^3)$. Likewise, the class of ST-optimal trees that satisfy the monotonicity principle is the most general characterization of tree cost functions for which optimal alphabetic binary trees can be generated in $O(n^2)$. Hu et al. [44] have proposed a generalization of tree cost functions called *regular* functions for which their $O(nlogn)$ algorithm produces optimal results. It is noteworthy that all trees constructed using a regular tree

cost function are also ST-optimal. Since the algorithm they used is based on dynamic programming, i.e., it assumes that optimal alphabetic subtrees are sufficient to generate optimal alphabetic trees, it is natural that all regular functions should be subtree optimal. A formal proof of this is presented next.

3.4.4 ST-optimality and Regular Tree Cost Functions

Before proving that all trees with regular tree cost function are ST-optimal, the definition of regular functions [44] is reproduced.

Definition 3.4.1 *Given a tree cost function C and a combining function F,* **regular** *functions satisfy the following conditions.*

P1: C cannot increase if any weight is replaced by a smaller weight.

P2: For all W_a, W_b and W_x,

- $F(W_a, W_b) = F(W_b, W_a)$
- $W_a \leq F(W_a, W_x)$
- $W_a \leq W_b \Longleftrightarrow F(W_a, W_x) \leq F(W_b, W_x)$

P3: If $W_b \leq W_c$ and $l_b \geq l_c$, where l_i denotes the number of edges from the root node to node i, then interchanging b and c cannot decrease the cost of the tree.

P4: Let T be a tree with n nodes in which a and b are merged. And let T^* be the tree with $n-1$ nodes resulting from the merger, with father of a and b having weight $F(W_a, W_b)$. Then there exists a function $\theta(W_a, W_b)$ such that

$$C_T = \theta(W_a, W_b) + C_{T^*}$$

P5: Functions $F(W_a, W_b)$ and $\theta(W_a, W_b)$ must satisfy the following conditions if $W_b \leq W_c$:

$$\begin{aligned} F[F(W_a, W_b), W_c] &\leq F[F(W_a, W_c), W_b] \\ \theta(W_a, W_b) + \theta[F(W_a, W_b), W_c] &\leq \theta(W_a, W_c) + \theta[F(W_a, W_c), W_b] \end{aligned}$$

Consider an internal node I_i in a tree T having children a and b. Now, consider an alternate tree structure T' in which node I' has the same leaf support. Let the children of I' be a' and b' with weight W'_a and W'_b, respectively. From the definition of regular functions, $W_I = F(W_a, W_b)$ and $W'_I = F(W'_a, W'_b)$.

Lemma 3.4.7 $\theta(W_a, W_b) < \theta(W'_a, W'_b) \Rightarrow F(W_a, W_b) \leq F(W'_a, W'_b)$ *and* $F(W_a, W_b) < F(W'_a, W'_b) \Rightarrow \theta(W_a, W_b) \leq \theta(W'_a, W'_b)$

Proof From definition, F and θ are symmetric and monotone nondecreasing in both variables. First, it is proven that $\theta(W_a, W_b) < \theta(W'_a, W'_b) \Rightarrow F(W_a, W_b) \leq F(W'_a, W'_b)$. There are three possible cases of weight changes.

Case 1 ($W_a \leq W'_a$ **and** $W_b \leq W'_b$)**:** In this case, since both θ and F are monotone in their variables $F(W_a, W_b) \leq F(W'_a, W'_b)$.

Case 2 ($W_a \geq W'_a$ **and** $W_b \geq W'_b$)**:** This contradicts the assumption that $\theta(W_a, W_b) < \theta(W'_a, W'_b)$.

Case 3 ($W_a < W'_a$ **and** $W_b > W'_b$) **or** ($W_a > W'_a$ **and** $W_b < W'_b$)**:** Without loss of generality it is assumed that $W_a > W'_a$ and $W_b < W'_b$. Since θ is symmetrical in its variables $\frac{\partial \theta}{\partial W_a} = \frac{\partial \theta}{\partial W_b}$ This implies that $W_a - W'_a < W'_b - W_b$. However, since F is also symmetrical in its variables, i.e., $\frac{\partial F}{\partial W_a} = \frac{\partial F}{\partial W_b}$, this implies that $F(W_a, W_b) \leq F(W'_a, W'_b)$.

Similarly, it can be proven that $F(W_a, W_b) < F(W'_a, W'_b) \Rightarrow \theta(W_a, W_b) \leq \theta(W'_a, W'_b)$
■

Theorem 3.4.8 *All regular functions are subtree optimal.*

Proof For this, it will be shown that $C_{T_I} < C'_{T_I} \Rightarrow C_T \leq C'_T$ for any internal nodes I and I' in two tree structures T and T' as described above.

From property P4 of regular functions, the tree cost of an internal node I in terms of its children can be written as $C_{T_I} = \theta(W_a, W_b) + C_{T^*_I}$. However, since there is only one leaf node in T^* with weight $W_I = F(W_a, W_b), C_{T^*_I}$ can be written

as $f(W_I)$. Since the rest of the tree structure remains the same, C_T can also be written as $C_T = \theta(W_a, W_b) + g(W_I)$. Hence we have,

$$C_{T_I} = \theta(W_a, W_b) + f(W_I)C_T = \theta(W_a, W_b) + g(W_I)$$

where from property P1 of regular functions, both f and g are monotone in W_I (i.e., weight of a leaf node for tree T^*).

With respect to another tree structure T', there are three cases to consider. I will show for each of them that $C_{T_I} < C'_{T_I} \Rightarrow C_T \leq C'_T$.

Case 1: $W_I \leq W'_I$ **and** $\theta(W_a, W_b) \leq \theta(W'_a, W'_b)$ In this case, since g is monotone in W_I, $g(W_I) \leq g(W'_I)$ which implies that $C_T \leq C'_T$.

Case 2: $W_I \geq W'_I$ **and** $\theta(W_a, W_b) \geq \theta(W'_a, W'_b)$ This lead to contradiction of the assumption that $C_{T_I} < C'_{T_I}$

Case 3: $W_I > W'_I$ **and** $\theta(W_a, W_b) < \theta(W'_a, W'_b)$ **or** $W_I < W'_I$ **and** $\theta(W_a, W_b) < \theta(W'_a, W'_b)$ Since $W_I = F(W_a, W_b)$, this case leads to direct contradiction of Lemma 3.4.7 for regular functions.

Thus, only case 1 is feasible for regular functions; under which $C_{T_I} < C'_{T_I} \Rightarrow C_T \leq C'_T$. ∎

I am not aware of any previous work to characterize tree cost functions that satisfy the monotonicity principle. However, I conjecture that all regular functions satisfy the monotonicity principle, in which case the run time of $O(n^2)$ is not much worse than $O(nlogn)$ run time of Hu-Tucker algorithm for regular functions. However, my algorithm will run in $O(n^2)$ for a much larger class of tree cost functions.

3.4.5 ST-optimality and Quasi-Linear/Schur-Concave Tree Cost Functions

Next, it is shown that the most general sets of combining and tree cost functions known so far for which Huffman's algorithm generates optimal trees – namely, all quasi linear combining functions with Schur concave tree cost functions [52] – result

in ST-optimal trees. It should be noted that these conditions were proposed for non-alphabetic binary trees. Thus, this work encompasses not only the alphabetic trees, but also other non-alphabetic tree optimization problems. However, the proposed algorithm is only applicable to alphabetic trees. This is because, unlike alphabetic trees, ST-optimality is not sufficient to guarantee polynomial runtimes for optimal non-alphabetic binary tree generation.

Before I show that quasi linear combining function with Schur concave tree cost function always results in a ST-optimal tree, some definitions and theorems from [32, 52] are restated.

Definition 3.4.2 *A weight combining function $F : U^2 \rightarrow U$, where U is a connected interval of nonnegative real values $\mathcal{R}_+$, is* **Quasilinear** *if $F(W_i, W_j) = \rho^{-1}[\lambda\rho(W_i) + \lambda\rho(W_j)]$ where λ is a nonzero constant and $\rho : U \rightarrow \mathcal{R}$ is invertible.*

Definition 3.4.3 *A weight sequence $\mathcal{W}$ is a row of nonnegative numbers $[W_1, W_2, \ldots, W_m]$ such that $W_1 \leq W_2 \leq \cdots \leq W_m$.*

Definition 3.4.4 *Given two weight sequences $\mathcal{W}$ and $\mathcal{W}'$, $\mathcal{W} \leq \mathcal{W}'$ if $\sum_{i=1}^{k} W_i \leq \sum_{i=1}^{k} W'_i$ holds for all $k, 1 \leq k \leq m$.*

Theorem 3.4.9 *(Glassey and Karp [32]) Let $\mathcal{W} = [W_1, W_2, \ldots, W_m]$ be the weight sequence for internal nodes in a tree constructed by the Huffman algorithm with the additive tree cost function, and let $\mathcal{W}' = [W'_1, W'_2, \ldots, W'_m]$ be the weight sequence for internal nodes of any other tree on the same leaf nodes. Then $\mathcal{W} \leq \mathcal{W}'$.*

Theorem 3.4.9 characterizes the trees constructed by Huffman algorithm with an additive tree cost function with fixed arity, namely such Huffman trees will always result in a weight sequence $\mathcal{W}$ such that $\mathcal{W} \leq \mathcal{W}'$ for any weight sequence $\mathcal{W}'$ generated from any tree structure on the same leaf nodes.

Theorem 3.4.10 *(D.S.Parker [52]) Let $F(W_i, W_j) = \rho^{-1}[\lambda\rho(W_i) + \lambda\rho(W_j)]$ be the weight combining function of the tree construction where ρ is convex, positive and strictly monotone and $\lambda \geq 1$. If $\mathcal{W}$ and $\mathcal{W}'$ are the weight sequences for internal nodes of the trees, constructed respectively by the Huffman algorithm and by any other algorithm, then $\mathcal{W} \leq \mathcal{W}'$.*

Theorem 3.4.10 generalizes the combining function from an additive combining function to a quasilinear function for which the Huffman algorithm will continue to generate optimal binary trees. However, tree cost function is still the addition of all internal node weights (i.e., weighted path length in terms of leaf weights). Fortunately, generalizing tree cost function to Schur concave maintains optimality of Huffman algorithm.

Definition 3.4.5 *A function $C : U^m \rightarrow \mathcal{R}$ is* **Schur Concave** *if*

$$(W_i - W_j)(\frac{\partial C}{\partial W_i} - \frac{\partial C}{\partial W_j}) \leq 0 \qquad \forall \; W_i, W_j \in U, \quad i, j \in \{1, 2, \ldots m\}$$

Theorem 3.4.11 *(Schur [93]) $C(\mathcal{W}) \leq C(\mathcal{W}')$ for all weight sequences $\mathcal{W} \leq \mathcal{W}'$ iff C is Schur concave.*

This condition is both necessary and sufficient, and hence exactly characterizes the tree cost functions for which the Huffman algorithm for binary trees generate optimal solution. Finally, combining quasi linear combination function with Schur concave tree cost function characterizes the most general class of trees for which Huffman algorithm for binary tree generates optimal solution.

Theorem 3.4.12 *(D.S.Parker) Let F be a function as defined in Theorem 3.4.10. The Huffman tree will have the least cost if the tree cost function C is any Schur concave function of internal node weights.*

The next theorem proves that ST-optimal trees are more general then the class of trees with quasi linear combining function and Schur concave tree cost function.

Theorem 3.4.13 *A tree with quasi linear combining function and Schur concave tree cost function is subtree optimal.*

Proof Consider a tree T with internal nodes $\{I_1, I_2, \ldots, I_m\}$ and corresponding weight sequence $\{W_{I_1}, W_{I_2}, \ldots, W_{I_m}\}$. Consider a subtree T_{I_i} rooted at an internal node I with tree cost $C_{T_{I_i}}$. T_{I_i} to T'_{I_i} are restructured such that $C_{T_{I_i}} > C_{T'_{I_i}}$. Then,

- Within the subtree structure, Schur concavity of the tree cost function implies that the weight sequence associated with T_{I_i} must be lexicographically greater

than the weight sequence associated with T'_{I_i}. Since quasi linear functions are monotonic, this implies that $W_{I_i} > W'_{I_i}$.

- For rest of the tree structure, only the effect of the change in weight of I_i will be propagated. However, since $W_{I_i} > W'_{I_i}$, monotonicity of quasi linear function will guarantee that weight of all internal nodes whose weight may change due to the weight change at node I_i (ancestors of node I_i), will only decrease.

This implies that for the new tree T' the new weight sequence will be lexicographically less than the weight sequence for tree T. Since the tree cost function is Schur concave this will imply that $C_T \geq C_{T'}$. ∎

3.4.6 SF-optimal trees

As described above, for non-binary ST-optimality of tree cost is not sufficient to guarantee polynomial runtime of alphabetic tree optimization. However, if $|\sum_{l=1}^{d-j} St\Psi_{l:i+j+1,i+d}|$ is restricted to a constant or even to a polynomial in $d-j$, Algorithm 3.2 will run in polynomial time, e.g., if $|\sum_{l=1}^{d-j} St\Psi_{l:i+j+1,i+d}| \leq c$, runtime of Algorithm 3.2 will be

$$\sum_{d=1}^{n-1}[\sum_{i=1}^{n-d}(\sum_{j=0}^{d-1} c+d)]+n = \sum_{d=1}^{n-1}[(c+1)(n-d)d]+n = (c+1)n^3/6-(c-5)n/6 = O(n^3).$$

If $|\sum_{l=1}^{d-j} St\Psi_{l:i+j+1,i+d}| = 1$ (as is the case for binary trees as well as for non-binary trees when the stronger definition of SF-optimality is satisfied) as the left most branch will never have more than one set of siblings. Thus, inner loop in Algorithm 3.2 is a constant time operation leading to a $O(n^3)$ run time for the algorithm.

According to the definition of SF-optimality, if the tree is SF-optimal, $|St\Psi_{l:i+j+1,i+d}| = 1; 1 \leq l \leq d-j$, giving $|\sum_{l=1}^{d-j} St\Psi_{l:i+j+1,i+d}| = d-j$. In this case Algorithm 3.2 will run in time complexity given by

$$\sum_{d=1}^{n-1}[\sum_{i=1}^{n-d}(\sum_{j=0}^{d-1}(d-j)+d)]+n = (n^4+6n^3-n^2-6n)/24 = O(n^4)$$

Intuitively, if a tree is SF-optimal, the cost contribution due to every subset of siblings is identifiable in terms of function H, allowing one to determine a priori if one k-tree subforest is better than another k-tree subforest on the same set of leaf nodes. Fortunately, conditions of SF-optimality are often satisfied, e.g., in case of Huffman's original additive combining function and tree cost function.

For generating degree-restricted optimal alphabetic trees, the above is modified to

$$\begin{aligned} \sum_{d=1}^{n-1}[\sum_{i=1}^{n-d}(\sum_{j=0}^{d-1} min[d-j,t]+d)]+n &\leq \sum_{d=1}^{n-1}[\sum_{i=1}^{n-d}(\sum_{j=0}^{d-1} t+d)]+n \\ &= (t+1)n^3/6-(t-5)n/6 = O(n^3t) \end{aligned}$$

In this special case, however, algorithm proposed here has reduced to the algorithm proposed by Vaishnavi et al. [121]. Application they were considering was t-ary alphabetic tree optimization with optimum average weighted search time. They also showed that for that application, no further reduction in run time is possible. Thus, their algorithm was derived for a specific tree cost function the proposed algorithm was arrived at from a more general algorithm. Hence, I can characterize tree cost functions for which the proposed algorithm can achieve a run time of $O(n^3t)$. An improved run time of $O(n^3logt)$ has been achieved by Itai [50] and Gotlieb [35]. However, that algorithm is applicable to a restricted set of tree cost functions for which there is always an optimal tree with maximum allowable number of branches at the root.

It is intuitive to note that ST-optimality was sufficient for binary trees because the only proper subsets of a set of two children of an internal node contain only one subtree. As a matter of fact, even for SF-optimality it is sufficient to consider only proper subsets of the set of children of an internal node. For example, for trees with degree bounded by t, it is sufficient that SF-optimality is satisfied for upto $(t-1)$-tree subforests.

3.5 Conclusion: Alphabetic Tree Enumeration and Optimization

In this chapter, the number of alphabetic trees on n leaf nodes was derived and a generic algorithm to produce alphabetic trees for any application was proposed. It was shown that the exponential run time of this algorithm can be significantly improved when the tree is ST-optimal. A set of tree cost functions that result in ST-optimal trees was classified. ST-optimal tree cost function allows generation of optimal alphabetic binary trees with the same time complexity as the best known algorithms. However, since this algorithm was derived using a top down approach as opposed to deriving optimal alphabetic binary tree with respect to a specific tree cost function, it can be generalized and applied to a large class of tree cost functions. To derive optimal alphabetic nonbinary trees in polynomial time, a further classification of tree cost functions was proposed, namely, that of SF-optimal trees. Again, it was shown that in most cases, the proposed algorithm reduces to the best known algorithm while being applicable for a much larger set of tree cost functions.

In the next chapter, applications of alphabetic trees in the field of logic synthesis are considered. The optimal alphabetic binary tree generation mechanism and optimal alphabetic nonbinary tree generation are applied to two different problems in logic synthesis. For further information on these applications see [84, 111].

Chapter 4

Alphabetic Trees: Applications in Layout-Driven Logic Synthesis

4.1 Introduction

Technology dependent phase of logic synthesis mainly consists of three stages: technology decomposition, technology mapping and fanout optimization. Technology decomposition (the procedure for converting an optimized Boolean network into a 2 input NAND-decomposed network) is a precursor to the technology mapping step. Technology mapping problem is the optimization problem of finding a minimum cost covering of this decomposed "subject graph" by choosing from a collection of "primitive graphs" for all gates in the library. In past years, technology independent logic synthesis research has concentrated on techniques that maximize logic sharing resulting in circuits with high number of fanouts per net. Excessive loads at such high fanout gates after technology mapping results in considerable performance degradation. Fanout optimization which generates buffer/inverter trees at the output of such heavily loaded gates is thus necessary to improve the circuit performance.

Both technology decomposition and fanout optimization are essentially tree optimization problems with different combining and tree cost functions[1]. Hence, algorithms developed for tree optimization are directly applicable to technology

[1]As described in Chapter 3, tree cost function is the cost of the tree defined on weights of tree nodes, e.g., arrival time at the root of the tree. Combining function is a function using which weight of a parent node is calculated, given weights of its children.

decomposition and fanout optimization. In Chapter 3, alphabetic tree generation/optimization algorithms were proposed that are directly applicable to the alphabetic version of technology decomposition and fanout optimization problems.

For technology decomposition the circuit is traversed in topological order (i.e., starting from primary inputs processing a gate only after each of its input has been processed), generating a binary tree corresponding to a 2 input NAND decomposition for each gate processed. For performance driven technology decomposition, given the arrival times at the leaf nodes, a binary tree with minimum arrival time at the root is generated. During technology decomposition, exact arrival times of signals are not known as the circuit is not mapped. Also, every node in a decomposition tree has only one output. Hence, it is appropriate to use the *unit* delay model of circuit delay during decomposition[2]. This implies that in a technology decomposition tree, leaf nodes have integer weights, where leaf weights corresponds to the maximum depth of the corresponding leaf node from primary inputs. During a post-map speed up operation, it is likely that exact arrival times are available at the input to the technology decomposition tree. In this case, technology decomposition algorithm should be able to handle real valued weights (i.e., arrival times) at the leaf nodes. Thus, for performance-driven technology decomposition, tree optimization algorithm should be able to generate a binary tree with minimum arrival time at the root (under a unit delay model), given integer or real valued arrival times at the leaf nodes.

For fanout optimization the arity of a fanout tree is not restricted to be two. Also, for fanout optimization unit delay model is not appropriate since the fanout tree structure depends on the load and relative required times at the leaf nodes. Hence, either unit fanout delay model or library delay model should be used. Since the circuit is mapped and fanout optimization is done in reverse topological order (i.e., starting from primary outputs processing gates such that all outputs of a gate are processed before processing the gate.), exact required times and loads at the leaf nodes are available under the library delay model. Thus, for fanout optimization,

[2]Unit delay model assumes that every gate has delay of 1 unit irrespective of the load. *Unit fanout* and *library* delay models are more accurate measures of circuit delay. Under the *unit fanout* delay model, delay of the gate is given by $1 + \alpha \cdot$ *number_of_fanout*, where $0 < \alpha \leq 1$. *Library* delay model uses accurate, pin-dependent values for intrinsic delay and drive of the source as well as accurate load values for the sinks.

tree optimization algorithm should be able to generate a non-binary tree under a load dependent delay model.

4.2 Prior Work

Conventional method for technology decomposition is to construct a *balanced* binary tree for each complex gate in the network. Wang [122] proposed a mechanism similar to Huffman's algorithm[3] [47] to speed up circuits. This technique can also be applied to technology decomposition. He proved its optimality (in terms of minimizing the signal arrival time at the root of the decomposed binary tree) for sum-of-products (SOP) expressions with orthogonal (disjoint variable support) cubes. Chen et al. [14] proposed a similar procedure for FPGA technology decomposition. Morrison et al. [71] proposed a heuristic top-down procedure that generates an unbalanced NAND decomposition of a complex gate in order to minimize the signal arrival time at the root.

Golumbic [34] and Hoover et al. [42] addressed fanout optimization under a *unit* delay model with a restriction on maximum number of fanouts per buffer. Golumbic's "combinatorial merging" algorithm [34] is an application of *t-ary* Huffman tree generation procedure to fanout optimization. Hoover et al. [42] present an algorithm to obtain networks of bounded fanin and fanout, so that both size and depth are not increased by more than a constant factor. Unit delay model, however, ignores the load and hence is not adequate for fanout optimization. Berman et al. [4], Singh et al. [98], and Touati [104] used more realistic *unit fanout* and *library* delay models. It was shown in [4, 104] that with these delay models, even under very simplistic assumptions, fanout problem is NP-hard. These results have motivated various heuristic solutions. Berman's algorithm generates optimal fanout trees for a restricted set of trees with identical required times for all sinks. Singh's heuristics consists of three operations: repowering, critical signal isolation, and load balancing. Touati's work on fanout optimization is the most comprehensive to date. He extended Golumbic's work to take into account varying loads and variable node

[3]Huffman's algorithm was originally proposed to construct a prefix-free encoding of a set of symbols with minimum average length given the probability of occurrence each symbol.

degrees for internal nodes of the fanout tree. To integrate *critical signal isolation* with *load balancing*, Touati proposed "LT-Trees" which balance loads and isolate critical signals simultaneously by grouping signals with similar required times at similar depths of the fanout tree.

4.2.1 Motivation for the Use of Alphabetic Trees for Layout-Driven Logic Synthesis

Previous work on technology dependent phase trades off active cell area for performance without regard to other circuit parameters such as routing area or wire load. Routing area currently accounts for up to 70% of the chip area and wire load accounts for up to 40% of total load driven by gates. Thus, routing plays an important role in determining total circuit area and circuit performance and must be addressed as early as possible during the design process. Using any of the existing technology decomposition or fanout optimization mechanisms, even if original circuit graph is planar, resultant circuit graph might be far from being planar. Even if the original circuit is nonplanar, it is desirable to have a technology dependent phase that does not increase the nonplanarity and hence does not create more routing difficulties.

Quality of the technology mapping when an optimal technology mapper is used, is highly influenced by the quality of technology decomposition. It is an open problem to determine which of the possible NAND-decomposed networks yield an optimum solution when an optimum covering algorithm is applied [6]. It is my belief that decomposition trees that minimize the signal arrival time at primary outputs of the network (or fanout trees that maximize the required time at primary inputs of the network) while maintaining the *crossing number*[4] of the network will generate better mapped circuits from a routability point of view. However, this should be achieved at minimal cost. As mentioned in Chapter 2, alphabetic decomposition trees and alphabetic fanout trees provide a good trade-off between circuit performance and routability.

[4] *Crossing number* is the minimum number of edge crossings for a drawing of the graph on a plane. A related concept is *planar thickness* that is, the minimum number of planar graphs whose union is the graph itself. These two definitions relate to the following problem: Given that a graph is not embeddable on a plane, how far is it from embeddability?

Linear order on the input(output) gates for decomposition(fanout) tree is derived from a companion placement solution of the circuit [82]. This placement is incrementally updated during technology decomposition and relaxed after technology mapping and fanout optimization. Instead of using placement information, topological (structural) information to derive the order on the leaf nodes can be used. Topological information is more abstract than the placement information and hence is more appropriate for use during technology decomposition, i.e., when exact gate implementation of the circuit is not known. Other mechanisms to order leaf nodes during technology decomposition or fanout optimization can be used as well. For example, an ordering based on required times at the leaf nodes of a fanout tree can be used on the premise that sinks with similar required time are likely to be on the same level of the fanout tree [104].

The remainder of this chapter is organized as follows. Section 4.3 describes my approach to generate optimal alphabetic technology decomposition. In Section 4.4, alphabetic fanout optimization problem and a set of rules that reduce the problem space are proposed. Sections 4.3 and 4.4 also describe the implementation and contain experimental results.

4.3 Layout-Driven Technology Decomposition

A *Boolean network* $\mathcal{N}$, is a directed acyclic graph (DAG) such that for each gate in $\mathcal{N}$ there is a corresponding Boolean function f_i, and a Boolean variable y_i, where $y_i = f_i$. There is a directed edge (i, j) from y_i to y_j if f_j depends explicitly on y_i or $\bar{y}_i$. A gate y_i is a *fanin* of a gate y_j if there is a directed edge (i, j) and a *fanout* if there is a directed edge (j, i). A gate y_i is a *transitive fanin* of a gate y_j if there is a directed path from y_i to y_j and a *transitive fanout* if there is a directed path from y_j to y_i. *Primary inputs* (PIs) are inputs of the Boolean network and *primary outputs* (POs) are its outputs. *Intermediate gates* of the Boolean network have at least one fanin and one fanout.

Intermediate gates in a *Boolean network* $\mathcal{N}$ could have simple functions (i.e., constant, buffer, inverter, AND, OR) or complex functions (i.e., arbitrary sum-of-products). During alphabetic technology decomposition, an n input AND function

$c = x_1 x_2 \ldots x_n$ is written as $s = \bar{c}$ and then s is decomposed into a tree of two-input NAND gates. An n input OR function $f = c_1 + c_2 + \cdots + c_n$ is written as $f = \overline{s_1 s_2 \ldots s_n}$ where $s_i = \bar{c_i}$ and then NAND-decomposed. A complex function with n product terms, $f = c_1 + c_2 + \cdots + c_n$, can be decomposed as follows: First, each product term c_i is written as $s_i = \bar{c_i}$ and then is NAND-decomposed; next, f is written as $f = \overline{s_1 s_2 \ldots s_n}$ and is NAND-decomposed. Thus, in all cases, the technology decomposition problem can be represented as NAND-decomposition of a function $\overline{x_1 x_2 \ldots x_m}$

Since the number of gates is same under any decomposition, the main objective of performance oriented technology decomposition is to minimize the arrival time at the root of the tree. Hence, arrival time at the root is the tree cost function for the tree constructed during technology decomposition.

4.3.1 Optimal NAND Decomposition of Product Terms

Here, the technology decomposition needs to be performed given a linear ordering λ on the leaf nodes, i.e., alphabetic technology decomposition. A formal definition of alphabetic technology decomposition problem is as given next.

Problem 4.3.1 ALPHABETIC_TECHNOLOGY_DECOMPOSITION

Instance: *A set of n ordered inputs* $(x_1, x_2, \ldots, x_n)$ *with arrival times* W_{x_i} *and a combining function* F *corresponding to the delay of a two input NAND gate that combines 2 inputs generating an internal node* I_p *with arrival time* $W_{I_p} = F(W_{child_{I_{p,1}}}, W_{child_{I_{p,2}}}) = max(W_{child_{I_{p,1}}}, W_{child_{I_{p,2}}}) + d,$ *where d is the delay of the NAND gate; and a tree cost function* $C_T = C(W_{x_1}, \ldots, W_{x_n}) = max_i(W_{x_i} + d\ l_i) = W_r$*, where r is the root of the decomposed tree and* l_i *is the number of NAND gates on the path from root to input node* x_i.

Problem: *Construct a NAND-decomposed tree with a minimal arrival time at the root without any internal edge crossing.*

Lemma 4.3.1 *Alphabetic technology decomposition trees are ST-optimal, independent of the delay model used.*

Proof The delay d is set to one for the unit delay model. Under the unit fanout delay model, since there is only one fanout for an internal node in the technology decomposition tree, d reduces to a constant value given by $d = 1 + \alpha \cdot number_of_fanouts = 1 + \alpha \cdot 1 = const.$ Furthermore, during the technology independent phase, it can be safely assumed that we are dealing with only one implementation of the NAND gate resulting in constant α. Hence, under any delay model the combining function reduces to $F = max(W_{child_{I_{p,1}}}, W_{child_{I_{p,2}}}) + const.$ As shown in Figure 3.5, this combining function results in in ST-optimal alphabetic trees. ∎

This allows use of Algorithm 3.3 to generate optimal alphabetic technology decomposition in $O(n^3)$. This runtime can be further reduced to $O(n^2)$ noting that the tree cost function is monotone as defined in [58]. From now on, modified version of Algorithm 3.3 to run in $O(n^2)$ is referred to as ALGALPTD[5]. The proof of the theorem presented next follows from Theorem 4.3.1. It is also possible to give an alternative proof based on the *monotone speed up property* for tree decompositions (see Lemmas 6.5.2 and 6.5.3 of [122]).

Theorem 4.3.2 *For an expression of the form* $\overline{x_1 x_2 \ldots x_m}$ *and given a linear ordering* λ *on* $\{x_i\}$, ALGALPTD *generates an alphabetic NAND-decomposed tree with minimum arrival time at the root, independent of the delay model used.*

Alternately, Huffman's algorithm can be used to perform a purely performance driven technology decomposition that ignores the edge crossings. An illustration comparing the delay optimal technology decomposition tree with delay optimal alphabetic technology decomposition is given in Figure 4.1. In this figure, the square boxes correspond to the original inputs to the 9-input NAND gates with the values inside the boxes (and circles) denoting the arrival times at corresponding nodes.

[5]Furthermore, since alphabetic technology decomposition combining function is a regular function, Hu-Tucker algorithm can be used as in [84] to reduce the runtime to $O(nlogn)$. See [46] for further details.

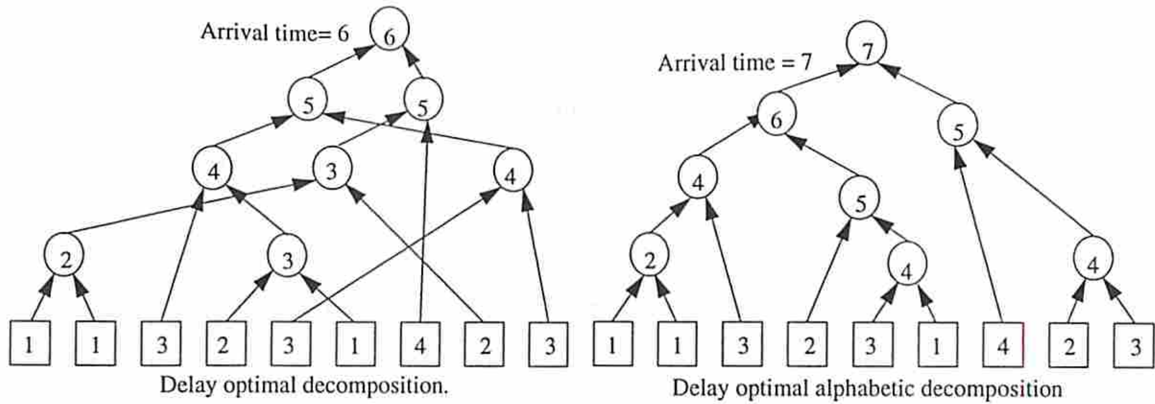


Figure 4.1: An illustration of delay optimal alphabetic and non-alphabetic technology decomposition under the unit delay model.

4.3.2 Optimal Decomposition of Sum-Of-Product Expressions

For complex gates with an SOP expression, the claims of optimality do not extend directly either for Huffman or for alphabetic technology decomposition. In general, a complex gate with an arbitrary SOP logic function could have the same variable appear more than once in the logic equation across different product terms. Huffman's algorithm generates area optimal technology decomposition as long as no variable appears in more than one product terms, i.e., if the product terms are orthogonal [122]. However, in the case of alphabetic trees, to guarantee alphabeticity of resulting tree, variable support for each product term should consist of consecutive elements of the linear order λ. Such product terms are referred to as *noncrossing.* This is shown in Figure 4.2. The following theorem guarantees optimality of alphabetic trees for functions with *noncrossing* product terms. It should be noted that if the product terms are noncrossing, it guarantees that consecutive products terms will share at most on literal, thus guaranteeing an area as well as delay optimal alphabetic technology decomposition.

Theorem 4.3.3 *For any gate whose SOP expression consists of noncrossing product terms and given λ a linear ordering on the input signals,* ALGALPTD *generates an alphabetic NAND-decomposed tree with minimum arrival time at the root, independent of the delay model used.*

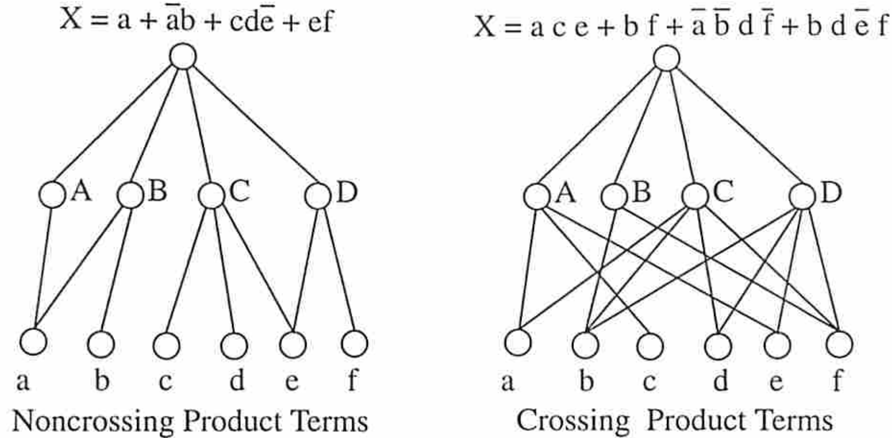


Figure 4.2: An illustration of noncrossing and crossing product terms.

A complex gate n with function $f = c_1 + c_2 + \cdots + c_m$ is decomposed as follows. The NAND decomposition of $\{s_i = \bar{c_i}\}$ is done using ALGALPTD. However, a linear order on variables $\{s_i\}$ should be derived in order to apply ALGALPTD to decompose $f = \overline{s_1 s_2 \ldots s_m}$. In case of non crossing cubes this order can be derived based on the order amongst the variable support for each product term as described above.

However, when the product terms are crossing, it is not possible to generate a technology decomposition tree without edge crossing. The problem then reduces to generating a technology decomposition with minimum number of edge crossings. As can be seen in Figure 4.2, the crossings occur only between the product terms and the literals belonging to these product terms. Since the order between the literals (leaf nodes) is already given, the problem reduces to that of deriving an optimal order amongst the product terms such that the crossing number is minimized, i.e., the linear order should be derived such that the edge crossing in the bipartite graph formed between $\{s_i\}$ and $\{x_j\}$ variables is minimized. A formal definition of the cube ordering problem for minimal crossing number is as follows.

Problem 4.3.2 OPTIMAL_CUBE_ORDERING_PROBLEM

Instance: *A bipartite graph (X, Y, E) where $X = \{x_j\}$ is the ordered set of variables appearing in the set of cubes $Y = \{s_i\}$ and there is an edge from x_j to s_i iff x_j is in the variable support of s_i.*

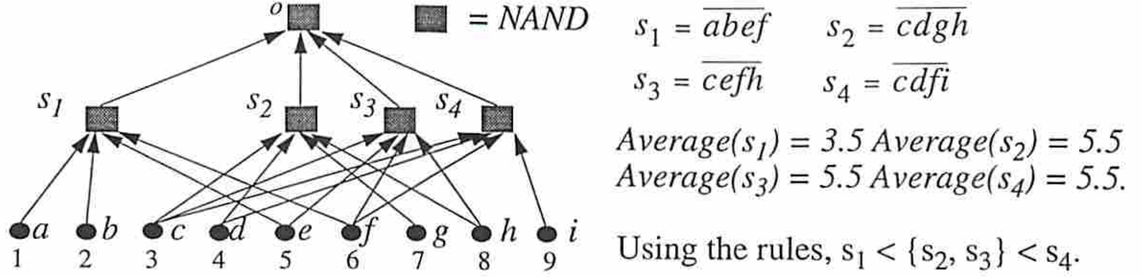


Figure 4.3: An illustration of cube ordering heuristic.

Problem: *Derive an order amongst $\{s_i\}$ such that the number of edge crossings in the resultant bipartite graph is minimized.*

The crossing problem which is shown to be NP-Hard [21] can be reduced to optimal cube ordering problem in linear time and hence the optimal cube ordering problem is also NP-hard. I have proposed the following heuristic [84] to derive the order between the product terms. Assign positions 1 to n to variables $x_1, \ldots, x_n$. Let r_i denote the column span of the variable support of s_i. Let $left(r_i)$ denote the leftmost column, $right(r_i)$ the rightmost column, and $average(r_i)$ the average of the column indices of the variables in the support of s_i. Let $s_i < s_j$ indicate that s_i is to the left of s_j. $\{s_i\}$ are sorted from left to right according to the following rule: if $average(r_i) < average(r_j)$, then $s_i < s_j$; In case of a tie, if $left(r_i) < left(r_j)$, then $s_i < s_j$; In case of a further tie, if $right(r_i) < right(r_j)$, then $s_i < s_j$; In case of yet another tie, an arbitrary order is imposed between s_i and s_j. Later, Eades et al. [21] showed that the crossing number achieved with this method is at most $\sqrt{|X|} \cdot Optimum_crossing_number$. An example of this ordering mechanism is shown in Figure 4.3.

4.3.3 Implementation and Experimental Results

Alphabetic technology decomposition of a Boolean network Γ, given a vector of arrival times Θ at the network primary inputs, is done as follows. First, the network size is estimated and the network is placed in the estimated boundary. Placement program is based on quadratic optimization where uniform distribution of gates over the layout plane is achieved by alternating global optimization and bi-partitioning steps over several levels of hierarchy [55].

For each gate n, let $\{x_i\}$ and $\{y_j\}$ denote its fanin and fanout gates in Γ. Next, the angle formed between edges connecting $\{x_i\}$ to n and the horizontal axis is calculated. This results in a *cyclic* order on the fanin signals. This is illustrated in Figure 2.2. Since alphabetic tree decomposition requires a linear order, a linear alphabetic order is produced by visiting x_i's in a counter-clockwise fashion and breaking the cycle at the angle corresponding to the (average) angle of $\{y_j\}$ with respect to n. This is a heuristic that tries to convert the cyclic order into a linear ordering. Alternatively, the cycle can be broken exhaustively between each pair of input gates and the order corresponding to the best decomposition can be selected.

After NAND decomposing n, new gates will replace it. Let η signify the subgraph containing the newly created gates. Subnetwork η is placed with respect to the original fanin and fanouts of n in Γ (i.e., $\{x_i\}$ and $\{y_j\}$). This is accomplished by finding a placement solution that minimizes the number of wire crossings subject to the linear order derived above. However, the region where the new gates should be placed must be determined as described next. First, n's *bounding* rectangle which is the smallest rectangle enclosing all $\{x_i\}$ and $\{y_j\}$ is calculated. I also find the n's *hierarchical region*[6]. In order to maintain the uniform distribution of gates on the placement plane, η is restricted to the intersection of its bounding rectangle and its hierarchical region. The placement mechanism is illustrated in Figure 4.4.

Finally, since the structure of Γ has changed, the signal arrival time for gates in Γ are recalculated before proceeding to the next gate. The process is repeated until all intermediate gates of Γ are decomposed.

All the benchmarks were first optimized for minimum area using the *rugged script.* Table 4.1 shows the network depth after technology decomposition and mapping for balanced, Huffman and alphabetic NAND decomposition procedures. It can be seen that the network depth for both Huffman and alphabetic decompositions is smaller than that for the balanced decomposition. In most cases, this trend is maintained after technology mapping. Furthermore, the difference between the network depths in both schemes is small. Thus, the cost (in terms of increase in network

[6]Suppose K bi-partitioning steps were performed during the initial placement of Γ, then the layout area is divided into 2^K equal-sized hierarchical regions such that each gate n is assigned to a unique region.

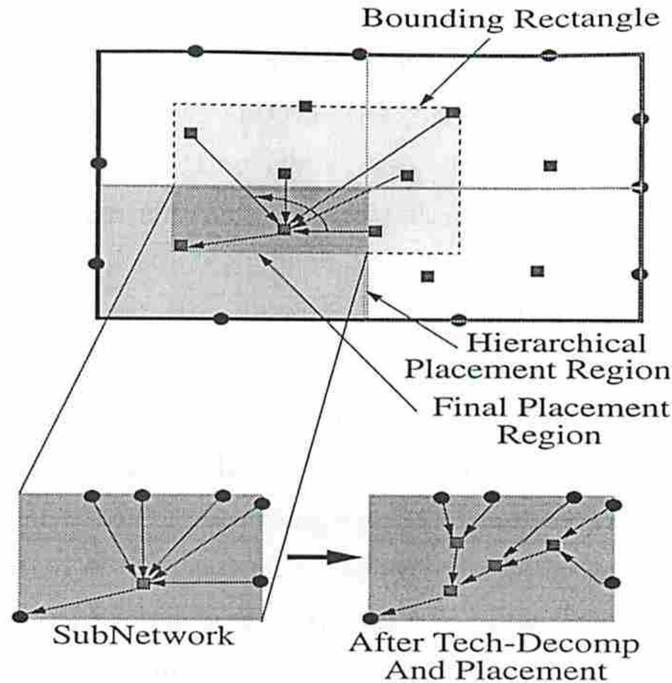


Figure 4.4: An illustration of incremental placement mechanism.

depth) of imposing a local linear order on inputs of the gates during decomposition is negligible.

The NAND-decomposed network was mapped using a 1 micron version of the *mcnc* library (with 28 gates) and SIS mapper (in timing mode). The circuits were placed and routed using the GORDIAN placement tool [55], TimberWolf's global router [63], and YACR2 detailed router [87] to generate results.

Table 4.2 shows comparisons between various NAND decomposition procedures in terms of total chip area, total routing area and delay after routing. When compared to balanced decomposition, alphabetic decomposition gives an average 6% improvement in circuit performance while reducing the chip area by 5% and routing area by 4%. When compared to Huffman decomposition, alphabetic decomposition reduces the chip and routing area by 4% but degrades the circuit performance by 4%. The running times for various decomposition schemes are comparable. For example, on a Sun Sparcstation 2, for C1355, balanced, Huffman and alphabetic decompositions took 2.5, 3.7 and 8.6 seconds, respectively. It should be noted that these results are generated using a 1 μm library. As the significance of interconnect increase with a reduction in device sizes [2], the results are likely to improve further.

Circuit	Network Depth						
		Balanced		Huffman		Alphabetic	
	Init	TD	M	TD	M	TD	M
C1355	10	19	13	18	14	19	14
C1908	16	35	28	30	25	32	27
C2670	10	34	27	26	24	28	25
C3540	24	48	39	43	39	45	41
C432	14	45	30	35	30	40	28
C6288	72	118	104	110	100	111	103
C7552	17	49	35	40	31	43	33
9symml	17	22	15	20	13	21	14
apex6	9	23	14	20	13	20	15
apex7	7	17	12	14	11	14	11
b9	6	9	9	8	8	9	9
k2	14	19	18	17	19	18	19
rot	11	25	22	20	18	22	20

Table 4.1: Network depth comparisons after TechDecomp (TD) and after Mapping (M).

Circuit	Balanced Decomp			Huffman Decomp			Alphabetic Decomp		
	Chip Area	Net Area	Delay	Chip Area	Net Area	Delay	Chip Area	Net Area	Delay
C1355	2152	836	12.86	2214	868	12.36	2147	844	12.44
C1908	2683	1075	21.24	2833	1117	19.59	2846	1122	20.64
C2670	12277	4943	24.64	12126	4839	22.00	12144	4705	23.74
C3540	19540	7663	34.25	20824	8139	32.71	18258	7318	33.15
C432	1737	682	25.58	1800	723	22.74	1567	646	23.52
C6288	24814	8708	84.67	17933	6531	77.34	17700	6439	77.63
C7552	25441	10080	34.01	26567	10610	31.81	26709	10731	32.87
C880	2265	879	22.11	2682	1030	20.55	2651	1002	20.74
9symml	1197	479	11.08	1201	488	9.49	1181	468	9.96
apex6	5850	2298	13.01	5463	2195	11.23	5505	2223	11.33
apex7	1246	518	8.37	1312	530	7.57	1208	500	7.31
b9	769	313	6.09	767	315	5.49	728	300	6.00
k2	11077	4360	15.80	10612	4283	14.83	10368	4172	15.88
rot	5751	2282	16.36	5768	2358	13.96	5501	2230	15.40
%	100	100	100	99.1	100	90.7	95.3	96.2	94.2

Table 4.2: Results for layout-driven technology decomposition.

4.4 Layout-Driven Fanout Optimization

Alphabetic fanout optimization problem may be formulated as follows.

Problem 4.4.1 ALPHABETIC_FANOUT_OPTIMIZATION (ALPFANOUT)

- **Instance:**

  1. *A set of n sinks $(L_1, L_2, \ldots, L_n)$ in a given, fixed order with 2-dimensional weights $W_{L_i} = (r_{L_i}, \gamma_{L_i})$ where r_{L_i} = required time at L_i and γ_{L_i} = input load of sink L_i.*

  2. *A 2-dimensional combining function (f_{req}, f_{load}) that combines l nodes generating an internal node I_m with weight vector W_{I_m} where,*

$$\begin{aligned} r_{I_m} &= f_{req}(W_{child^1_{I_m}}, W_{child^2_{I_m}}, \ldots, W_{child^l_{I_m}}) \\ &= min_i(r_{child^i_{I_m}}) - \beta_{buf} \sum_i \gamma_{child^i_{I_m}} - \alpha_{buf} \qquad (4.1) \\ \gamma_{I_m} &= f_{load}(W_{child^1_{I_m}}, W_{child^2_{I_m}}, \ldots, W_{child^l_{I_m}}) \\ &= \gamma_{buf} \qquad (4.2) \end{aligned}$$

  $\beta_{buf}, \alpha_{buf}$ and γ_{buf} denote drive resistance, internal delay and input load of the buffer, respectively.

  3. *A tree cost function $C(T) = r_{I_m}$ where I_m is the root of the fanout tree T.*

- **Problem:** *Generate a tree T such that the cost (required time at the root) is maximum and the tree has no internal wire crossings*[7].

Combining function in ALPFANOUT corresponds to the *library* delay model. The problem can be easily modified to consider other delay models by restricting the values of different parameters. For example, if unit fanout delay model is used, $\alpha_{buf} = 0$, $\beta_{buf} = 1$ and $\gamma_{buf} = 1$. An instance of alphabetic fanout optimization problem and the required time at the source under different delay models are given

[7] Proposed method can be extended to limit the number of fanouts per buffer. However, for the sake of simplicity, here it is assumed that buffers have with no fanout limit.

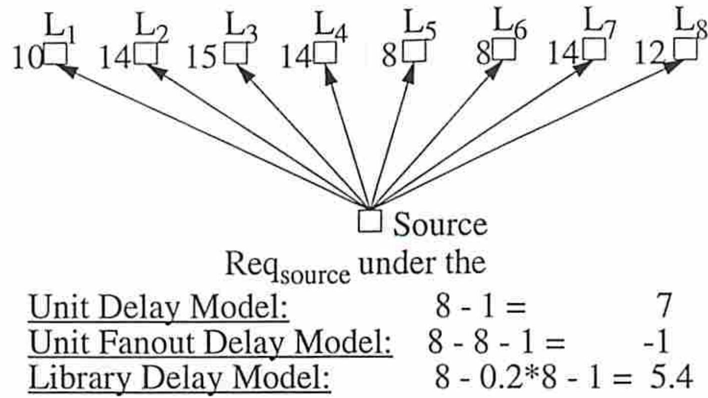


Figure 4.5: An instance of alphabetic fanout optimization problem and the required time at the source under different delay models.

in Figure 4.5. Under the library delay model more than one buffers may be available with different $\beta_{buf}, \alpha_{buf}$ and γ_{buf}, adding an extra degree of complexity to the problem. However, since the number of buffers is small in most libraries, to simplify the problem only one buffer type was used in this implementation. However, implications of allowing multiple buffers are discussed in Section 4.4.2.

Lemma 4.4.1 *Alphabetic fanout trees are SF-optimal under unit delay model and unit fanout delay model.*

Proof For the unit delay model, the proof follows from the constant value of f_{load} ($f_{load} = 0$). The optimal solution in this case is trivial, namely, a tree in which the root directly connects to all the sinks. Even under unit fanout delay model, value of f_{load} is a priori known for a D-rooted forests $\forall D$ ($f_{load} = c\ D$ = constant multiplied by number of outputs). Hence in either case, it is sufficient to keep the D-rooted forest with the maximum required time. The corresponding tree cost decomposition is shown in Figure 3.6. ■

Above lemma implies that under the unit delay or unit fanout delay, alphabetic fanout optimization problem is optimally solvable in polynomial time. Even under library delay model the above lemma holds if the loads are identical. However, with different loads or multiple buffers the alphabetic fanout trees are no longer SF-optimal. Fortunately, even with different loads alphabetic fanout trees are still ST-optimal if there is only one buffer in the gate library.

Lemma 4.4.2 *Alphabetic fanout trees are ST-optimal under library delay model if the library contains only one buffer/inverter.*

Proof According to the definition of ST-optimality, a tree is subtree optimal if the tree cost is monotone in the tree cost of each of its subtrees. Since there is only one internal node, all internal nodes have the same load. Thus, only required time at an internal node may change due to restructuring of the subtree structure at that internal node. As per the combining equation of alphabetic fanout optimization problem, increasing the required time at an internal node while maintaining the load at constant may never result in a decrease of the tree cost function (required time at the root). ∎

As a result of above lemma and Lemma 3.4.2, Algorithm 3.2 with $O(2^n)$ complexity can be used for solving the ALPFANOUT problem optimally. Otherwise, one needs to resort to Algorithm 3.1. The specialization of Algorithm 3.2 or Algorithm 3.1 (as the case maybe) for the ALPFANOUT problem is referred to as ALGALP-FANOUT. However, before applying this algorithm, the ALPFANOUT problem is analyzed further in order to simplify the problem space and to reduce the number of trees that need to be considered in order to generate an optimal alphabetic fanout tree. This reduced set of trees is referred to as the set of *apropos trees* for alphabetic fanout trees, i.e., trees that are sufficient for the purpose of alphabetic fanout optimization.

The delay through a buffer is given by $\alpha_{buf} + \beta_{buf} \sum_{j \in FO_{buf}} \gamma_j$ where FO_{buf} denotes fanouts of the buffer . The wiring load is estimated dynamically based on the number of fanouts. This load can then be included in γ_j. Using this mechanism, the required time at an intermediate buffer k is given by $r_k = min_{j \in FO_k}(r_j) - \alpha_{buf} - \beta_{buf} \sum_{j \in FO_k} \gamma_j$.

The fanout tree generating rules given below do not undermine the optimality of the algorithm but improve its efficiency. Given a set of original sinks, these rules generate a modified set of sinks. An additional requirement for these rules to be valid is that the $\gamma_j \geq \gamma_{buf}$ for each sink L_j. This requirement is satisfied in most cases as input capacitance of the inverter is less than most other gates. Similar rules for the unit delay model were proposed in [19] for generation of t-ary minimax tree under the unit delay model.

Lemma 4.4.3 *If the input capacitance of the inverter is less than all the sinks, given n internal or sink nodes, application of the following rules does not undermine optimality of* ALGALPFANOUT.

Rule 1: *If* $n \geq 1$ *and* $r_i \geq max(r_{i-1}, r_{i+1}), 1 \leq i \leq n$, *make* $r_i = max(r_{i-1}, r_{i+1})$.

Rule 2: *If* $max(r_i, r_{i+s+1}) \leq min(r_{i+1}, \ldots, r_{i+s}) - \alpha_{buf} - \beta_{buf} \sum_{j=1}^{s} \gamma_{i+j}$, *replace the sequence* $L_{i+1}, ..L_{i+s}$ *of nodes by one node with required time* $min(r_{i+1}, ...r_{i+s}) - \alpha_{buf} - \beta_{buf} \sum_{j=1}^{s} \gamma_{i+j}$.

Proof Instead of providing an exhaustive case enumeration, only an outline of the proof is provided. Each rule takes a set of sinks and produces a modified set of sinks.

Rule 1: For *Rule 1*, r_i will either get JOINed with r_{i-1}, r_{i+1}, or an ancestor or r_{i-1} or r_{i+1}. In each case, the required time of r_i will get dominated by it's sibling, as long as $r_i \geq max(r_{i-1}, r_{i+1})$

Rule 2: Consider an optimal alphabetic tree on leaf nodes 1 to n. *Rule 2* can be proved by considering all subtrees on some proper subset of leaf nodes $i+1, i+2, \ldots, i+s$ with a sibling that is either r_i or r_{i+s}, or any ancestor of r_i or r_{i+s}. Each such subtree can be replaced by a subtree with load γ_{buf} and a required time given by $min(r_{i+1}, ...r_{i+s}) - \alpha_{buf} - \beta_{buf} \sum_{j=1}^{s} \gamma_{i+j}$, without changing the required time at the root. Finally, keeping only one of these subtrees, while discarding the rest can never result in a decrease in the required time the the root. This exactly corresponds to application on *Rule 2*. Thus, application of *Rule 2* never undermines optimality of ALGALPFANOUT. ∎

Applications of these rules are illustrated in Figure 4.6. To simplify the presentation, it is assumed the unit fanout delay model, i.e., $\alpha_{buf} = 1, \beta_{buf} = 1$ and $\gamma_j = 1$ for all j. As shown in the figure, let the required time at sinks be given by vector r = (10, 14, 15, 14, 8, 8, 14, 12). *Rule 1* is applied on L_3, while *Rule 2* is applied on (L_2, L_3, L_4) and on (L_7, L_8), generating internal nodes I_1 and I_2, respectively. At this stage, an *impasse* is reached as neither of the rules can be applied. Instead, if the unit delay model is used, these rules would have generated optimal alphabetic fanout trees in O(n) time complexity without encountering any impasse. For the *unit fanout* or the *library* delay model, impasse is likely to occur, in which case the impasse is resolved by resorting to the tree generation part of ALGALPFANOUT.

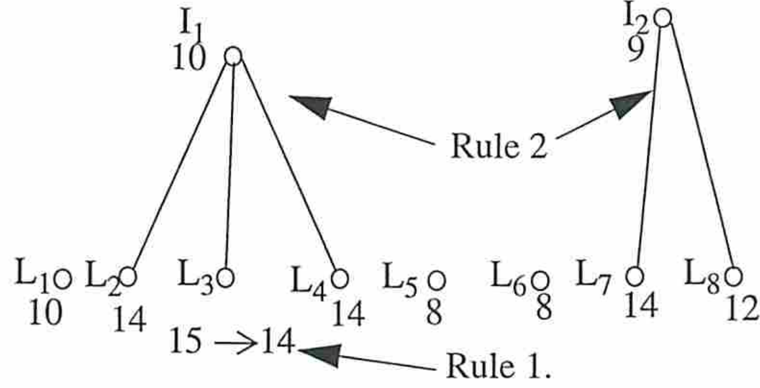


Figure 4.6: Illustration of *Rule 1* and *Rule 2* for optimal alphabetic fanout optimization.

Application of these rules, however, has reduced the solution space by reducing the number of sinks from 8 to 5.

4.4.1 Handling Different Loads

Let the set of apropos trees on leaf nodes i through j be denoted as $Ap\Psi_{i,j}$. As seen from equation (3.19), because of the tree SPLITting term, all subtrees in $Ap\Psi_{i,j}; \forall i,j \leq n$ must be generated. This number can be reduced significantly using the following theorem.

Let R_T, L_T, req_T and $load_T$ denote minimum of the required times at immediate children of the root of tree T, cumulative load offered by immediate children of the root of tree T, required time at the root of the tree T, and the load at the root of the tree T, respectively. Using this notation, required time for a tree T generated by $\wedge(\{T_l\} \times (.\dot{\div}.T_r))$ is given by:

$$req_T = min(req_{T_l}, T_{T_r}) - \beta_{buf}(load_{T_l} + L_{T_r}) - \alpha_{buf}$$

Definition 4.4.1 *Given two trees T' and T'' of the same set of leaf nodes with $R_{T'} \geq R_{T''}$,*

- *T' and T'' are* non-inferior *with respect to each other if $(R_{T'} - R_{T''}) > \beta_{buf}(L_{T'} - L_{T''}) > 0$.*

- *T' is* superior *to T'' if $(R_{T'} - R_{T''}) \geq 0 \geq \beta_{buf}(L_{T'} - L_{T''})$.*

otherwise, T' is inferior *to T''.*

Theorem 4.4.4 *For fanout optimization, when generating the set of apropos trees on sinks j through m, it is sufficient to maintain a set of trees that are non-inferior with respect to each other, but superior to all other trees.*

Proof As per the notation proposed earlier, the set of apropos trees on sinks $j+1$ through m is denoted by $Ap\Psi_{j+1,m}$ and the set of all alphabetic trees on sinks $j+1$ though m is denoted by $\Psi_{j+1,m}$. When a tree $T_r \in \Psi_{j+1,m}$ is SPLIT, it generates a forest F_r with minimum of the required time at the roots given by R_{T_r} and sum of loads at the roots given by L_{T_r}. It needs to be shown that apropos trees are sufficient to generate optimal alphabetic fanout tree in any tree structure.

According to tree generation procedure, while generating a tree $T \in \Psi_{i,m}$, F_r must be JOINed with the best tree $T_l = \theta_{i,j}$. In this case, the sibling of F_r consist only of T_l. The required time at the root of the tree T is given by

$$req_T = min(req_{T_l}, R_{T_r}) - \beta_{buf}(load_{T_l} + L_{T_r}) \tag{4.3}$$

Consider any two trees $T'_r, T''_r \in \Psi_{j+1,m}$. Without loss of generality, it is assumed $R_{T'_r} \geq T_{T''_r}$. From Definition 4.4.1, T'_r is either superior, inferior or non-inferior with respect to T''_r. Next, it is shown that if T'_r is superior or inferior with respect to T''_r, only one of the two needs to be considered to guarantee optimality.

T'_r is superior: Given $(R_{T'_r} - R_{T''_r}) > 0 \geq \beta_{buf}(L_{T'_r} - L_{T''_r})$, the following must be proven:

$$min(req_{T_l}, R_{T'_r}) - \beta_{buf}(load_{T_l} + L_{T'_r}) > min(req_{T_l}, R_{T''_r}) - \beta_{buf}(load_{T_l} + L_{T''_r})$$

This is true from the definition of superior tree irrespective of the value of $req_{T_l}, load_{T_l}$ and β_{buf}.

T'_r is inferior: Given $\beta_{buf}(L_{T'} - L_{T''}) \geq (R_{T'} - R_{T''}) > 0$, we need to prove the following:

$$min(req_{T_l}, R_{T'_r}) - \beta_{buf}(load_{T_l} + L_{T'_r}) < min(req_{T_l}, R_{T''_r}) - \beta_{buf}(load_{T_l} + L_{T''_r}) \tag{4.4}$$

From equation (4.4) the following can be obtained.

$$min(req_{T_l}, R_{T'_r}) - min(req_{T_l}, R_{T''_r}) < \beta_{buf}(L_{T'_r} + L_{T''_r})$$

From Definition 4.4.1, this is true for each of the three cases, $T''_r \leq T'_r \leq T_l$, $T''_r \leq T_l \leq T'_r$, and $T_l \leq T''_r \leq T'_r$.

Thus, when the siblings of F_r consist only of a single tree, i.e., T_l, only those trees from $\Psi_{j,m}$ need to be considered that are non-inferior with respect to each other.

When the sibling of F_r consist of more than one tree, the above can be proved similarly by considering req_{T_l} to be the minimum of the remaining siblings and $load_{T_l}$ to be the total of their load. ∎

Theorem 4.4.4 enables the reduction of the number of apropos trees for ALGALP-FANOUT by only generating the set of non-inferior trees. As each tree structure is generated, it is compared with trees in the current set of non-inferior trees, deleting the inferior trees during the comparison. A rule that exploits this fact is proposed next.

Rule 3: During tree generation on sinks i through m, current tree T is added to the set of non-inferior trees on nodes i through m only if T is non-inferior to all the trees in the current set of non-inferior trees. If T is superior to some trees currently in the set of non-inferior trees, those trees are removed from the current set of non-inferior trees before adding T.

The correctness of *Rule 3* follows from Theorem 4.4.4.

4.4.2 Handling Multiple buffers

If the library of gates contains multiple buffers/inverters, ST-optimality of alphabetic fanout optimization is not guaranteed. Consider subtrees generated on leaf nodes $i, \ldots, i+d$. For each buffer/inverter in the library the best alphabetic tree on these leaf nodes can be generated with the corresponding buffer/inverter being the root of the tree. Let us denote by $\mathcal{B}$ the set of buffers in the gate library. Let the gate driving the root of the fanout tree be G. So far, it was assumed that the root of the tree is also the same buffer. Here, along with allowing multiple buffers, a sink is also allowed to be driven directly by the library gate corresponding to the function.

Then, the definition of superior and non-inferior trees can be extended between pairs of 1 tree forests.

Definition 4.4.2 *Given two trees on T' and T'' on same leaf nodes with either $req_{T'} \geq req_{T''}$ or $R_{T'} \geq R_{T''}$,*

- *T' and T'' are* non-inferior *with respect to each other if $(req_{T'} - req_{T''}) > \beta_{buf}(load_{T'} - load_{T''}) > 0$ or if $(R_{T'} - R_{T''}) > \beta_{buf'}L_{T'} - \beta_{buf''}L_{T''} > 0$.*
- *T' is* superior *to T'' if $(req_{T'} - req_{T''}) > 0 \geq \beta_{buf}(load_{T'} - load_{T''})$ and if $(R_{T'} - R_{T''}) > 0 \geq \beta_{buf'}L_{T'} - \beta buf'' L_{T''}$.*

$\forall \ buf, buf', buf'' \in \mathcal{B} \cup \{G\}$.

In context of Definition 4.4.2, Theorem 4.4.4 can be extended and correspondingly *Rule 3* can be modified. Since these are simple modifications, they are not discussed in detail. However, a direct implication of allowing multiple buffers is that, for each subset of leaf nodes, instead of maintaining only one subtree, as many subtrees as number of buffers may be required. Also, as per Definition 4.4.2, the set of non-inferior tree may increase now as in the worst case, the best subtree for each buffer type may need to be kept. However, since number of buffers is usually small and since all the inferior subtrees are filtered out during tree generation, the run times do not grow substantially due to multiple buffers.

Finally, it should be noted that though Definition 4.4.2 and Rules 1 through 3 are proposed for alphabetic fanout trees, they can be used as effectively to compare two non-alphabetic fanout trees with identical set of leaf nodes.

4.4.3 Implementation and Experimental Results

The algorithm ALPFANOUT_ALG was implemented in the *SIS* environment. Mapped networks were optimized with ALPFANOUT_ALG after deriving an order on the fanouts of each gate using the ordering mechanism specified.

Algorithm 4.1 AlpFanout_Alg (Γ, Θ, Ω)
Γ is an optimized mapped Boolean network
Θ is a vector of required times
Ω is the ordering mechanism
begin
  for each gate $n \in \Gamma$ (in preorder) **do**
    L = Order_sinks (Γ, n, Ω)
    L' = Reduce_sinks (L, Θ)
    η = Generate_best_AlpFanout_tree (n, L')
    update_network (Γ, n, η)
  end
end

Reduce_sinks reduces the number of sinks using *Rule 1* and *Rule 2*. Given a set of ordered sinks, Generate_best_AlpFanout_tree returns an optimal fanout tree on the ordered sinks L' using *Rule 3* and the apropos tree generation equation (3.19). Application of *Rule 3* during tree generation significantly improves efficiency of ALPFANOUT_ALG. This is illustrated in Figure 4.7 and described next.

Continuing with the previous example, an optimal fanout trees on the modified set of sinks $(L_1, I_1, L_5, L_7, I_2)$ needs to be generated. From these 5 sinks, using the tree generation mechanism, all alphabetic trees on every subset of ordered sinks of size 2 to 5 is generated. According to *Rule 3*, every tree in the list of current apropos tree should be *non-inferior* to all others. Inferior trees are dropped from the current list of trees and are excluded from further consideration.

For this particular example, from equation (3.8), there are 4279 alphabetic fanout tree structures. Due to the monotone tree cost function, the number of apropos trees (i.e., trees that must be considered to find the optimal solution) is 127. Use of *Rules 1* and 2 reduces the number of sinks to 5, hence lowering the number of apropos trees to 31. *Rule 3* eventually reduces the total number of apropos trees on final set of sinks to 9. Note that *Rule 3* also reduces the number of apropos trees during the generation of subtrees. Since, on average, number of sinks for fanout optimization ranges between 3-6 sinks, in spite of being exponential in the worst

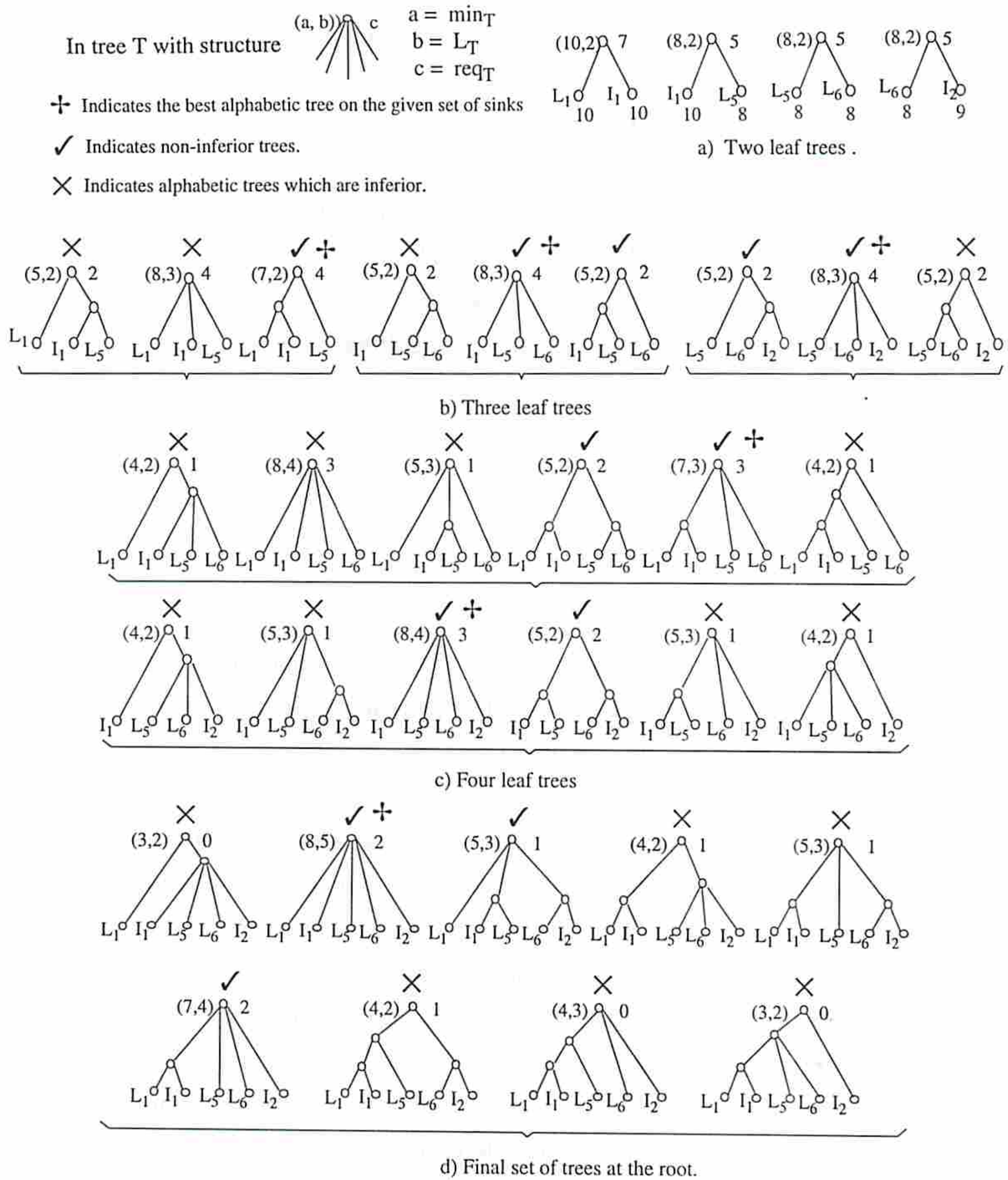


Figure 4.7: Generating optimal alphabetic fanout trees.

case, ALGALPFANOUT is quite fast in practice. This also shows how a detailed analysis of an exponential algorithm can lead to better runtimes of the optimal algorithm, alleviating of any need for non-optimal heuristic procedures.

Next, a description of how different polarities of the sinks were handled and how the order was derived amongst the sinks is presented. Previous algorithms considered sinks with differing polarities separately. However, to maintain alphabetic order on the sinks irrespective of their polarities, the following mechanism was used. For every set of the sinks, two fanout trees were generated: one with positive polarity at the root and the other with negative polarity at the root. Apart from this, during every JOIN operation, only the subtrees with identical polarities were joined.

The ordering on the sinks were derived using different mechanisms[8]. The PLACE_ORDER imposes a sink ordering based on the sink positions derived from a global placement solution for the Boolean network. The idea is that since the placement solution captures the connectivity structure of the network and the addition of fanout tree does not modify the network structure to a great extent, this placement solution can be relied upon for estimating the relative positions of the sinks after fanout optimization and placement. By using the placement information, due to the non-crossing property of the alphabetic trees, the crossing number of the network is preserved during fanout optimization. Here the performance of the circuit is traded for improved circuit routability.

The REQUIRED_ORDER generates a sink ordering based on the required times of the sinks. The rationale behind this ordering is that, in general, it is desirable (from the performance point of view) to put sinks with similar required times at the same depth in the fanout tree. This option has been adopted by others as well [104]. However, this fanout optimization algorithm is provably better than other algorithms as shown in the previous sections.

Table 1 compares ALPFANOUT with SISFANOUT [104] for all multi-level logic benchmarks recommended in [126]. All circuits were first optimized in SIS [95] using area optimizing script. The circuits were mapped using the SIS mapper in timing mode [103] and then optimized using two fanout optimization algorithms. After the

[8]Depending on the secondary objective of fanout optimization (routing congestion, power efficiency, etc), other ordering mechanisms can be proposed and implemented.

circuit	SisFanout		AlpFanout			
			Req		Place	
	Area 10^3	Delay ns	Area 10^3	Delay ns	Area 10^3	Delay ns
C1355	2838	30.62	2566	30.56	2440	31.39
C1908	3803	45.02	3049	43.17	2855	46.16
C2670	5229	33.65	4805	30.59	4790	32.61
C3540	11531	65.99	10512	64.69	10600	66.71
C432	1307	40.31	1184	39.43	1008	39.32
C6288	27060	165.67	25939	167.72	21693	167.59
C7552	21209	80.91	22133	71.20	19090	67.06
9symml	1304	21.53	1162	21.59	1143	23.53
b9	593	11.23	561	10.94	559	11.56
dalu	11408	69.23	11497	70.84	10021	73.92
k2	11925	38.25	11570	36.82	10497	38.30
rot	5043	29.55	4699	29.51	4224	34.28
t481	6698	30.57	5930	29.56	5828	31.00
% Gain	100	100	92.9	97.3	86.3	101.6

Table 4.3: Results for layout-driven fanout optimization.

fanout optimization the circuits were placed using GORDIAN [55] and routed using TimberWolf global router [63] and YACR2 detailed router [87].

The first two columns give results generated by SISFANOUT. SISFANOUT tries a number of fanout optimization algorithms (LT_trees, Two_Level, Balanced, etc) at each gate and picks the best fanout solution. The columns denoted by REQ corresponds to the ALPFANOUT results with REQUIRED_ORDER. Last two columns corresponds to the PLACE_ORDER option of ALPFANOUT. For PLACE_ORDER the circuits were placed using GORDIAN and sink orders were derived from this placement.

As can be seen, ALPFANOUT does better then SISFANOUT in area for all cases. The best performance results are obtained with the REQUIRED_ORDER, and the best routing (smallest chip area) is obtained with the PLACE_ORDER. Overall the PLACE_ORDER saves 14% chip area as compared to SISFANOUT without a significant performance degradation. Fanout trees generated by REQUIRED_ORDER are about 4% faster than PLACE_ORDER but at the cost of 6% increase in chip area. However, I would like to mention that some of the improvement in chip area is also due to improvement in the active gate area. Since alphabetic fanout optimization routine introduces a buffer only when it is absolutely necessary, typically ALPFANOUT achieves similar delay with a much reduced numbed of inverters/buffers.

ALPFANOUT runtimes are quite comparable with those of the SISFANOUT. On a Sun Sparc Station 2, for $C1355$, $C1908$, $C2670$, $C3540$, $C432$, $C6288$ and $C7552$, ALPFANOUT took 171.0, 169.4, 796.5, 595.7, 69.2, 1826.5, and 2634.0 seconds respectively, versus the SISFANOUT time which were 181.5, 215.8, 807.3, 639.1, 81.4, 2099.2 and 1931.4 seconds respectively.

Chapter 5

An Analysis of Placement Based Approaches

A major problem with placement based approaches is that the companion placement information becomes increasingly less reliable at earlier stages of logic synthesis. This is due to unreliability of the gate positions as they are derived from placement solution for a structure that will change drastically during the subsequent logic transformation steps. Moreover, fewer nodes of the original circuit during the earlier stages of logic synthesis remain in the final circuit making the companion placement information even less reliable.

To assess the reliability of the companion placement locations, an analysis is performed to derive correlation between the companion placement locations and final placement locations at different stages of logic synthesis. The concept of *conformity* is introduced to capture the reliability of the companion placement.

Let $\{X_i^c, Y_i^c\}$ represent the X and the Y coordinates of gate i in the companion placement of a network N. Let the corresponding final placement position be $\{X_i^f, Y_i^f\}$. A pair of nodes i and j are defined to be *X-conforming* if $X_i^c \geq X_j^c$ and $X_i^f \geq X_j^f$, or if $X_i^c \leq X_j^c$ and $X_i^f \leq X_j^f$. In other words, nodes i and j are X-conforming if the relative final positions of nodes i and j conform to companion placement position obtained at an earlier stage of logic synthesis.

Based on this definition, the *X-conformity* of a node i is defined as the percentage of the number of gates in the circuit that are X-conforming to gate i. The *X-conformity* of the network is then defined as the average *X-conformity* of all nodes in the network. *Y-conformity* of a node and of the network are defined similarly.

Conformity is a strict measure of the reliability of the companion placement. If the companion placement is a random placement, the value of conformity is 50%.

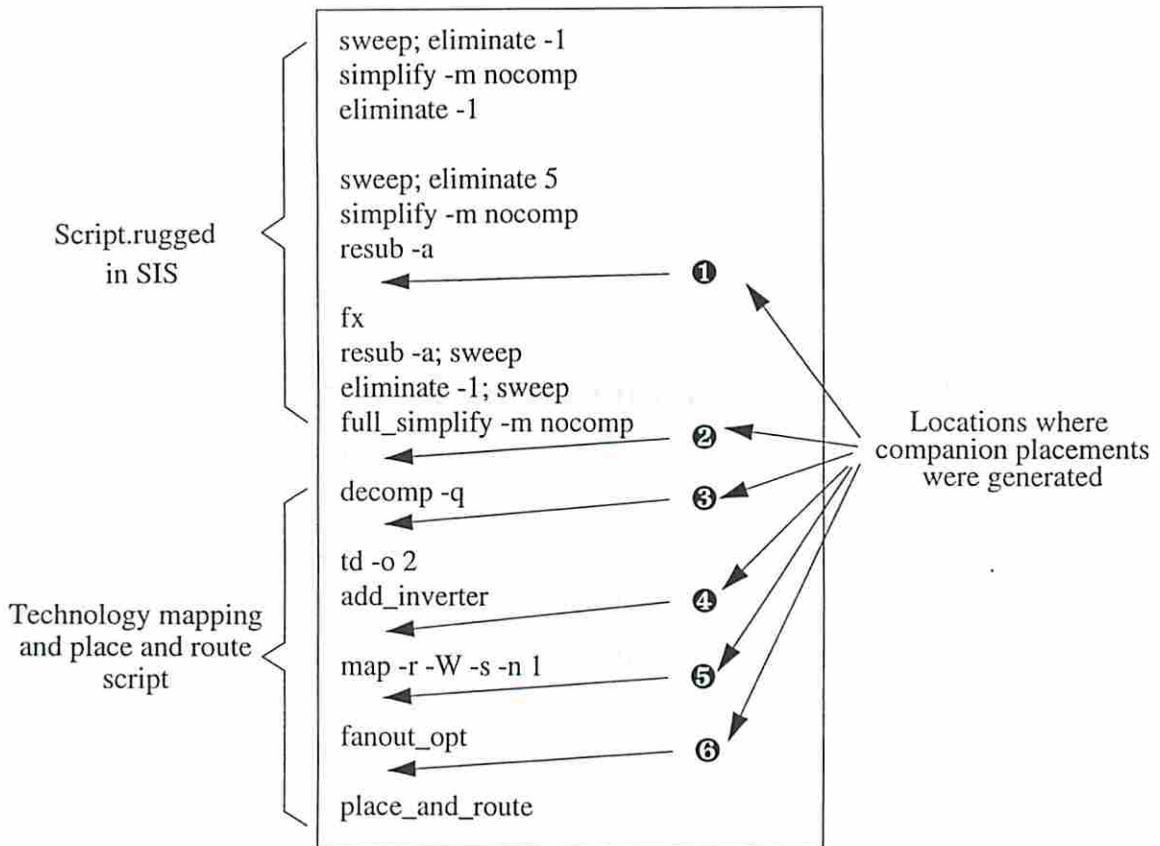


Figure 5.1: The script used for conformity analysis.

Any bias towards 100% (or 0%) indicates improved reliability of the companion placement. However, as the conformity gets closer to 100%, the accuracy of the companion placement increases super-linearly. If only 1% of nodes do not conform to any other nodes, the conformity of the circuit is 98%, while if 25% of the nodes do not conform to any other nodes, the conformity drops to 56.25%.

To analyze the reliability of companion placement during different stages of multi-level logic synthesis, the following experiment was carried out on the benchmark circuit *duke2*. A companion placement was generated at different stages of logic synthesis using the SIS area optimizing script *script.rugged* [92]. That is, (1) immediately before the extraction; (2) after extraction; (3) after node decomposition; (4) after technology decomposition; (5) after technology decomposition; and (6) after fanout optimization. These stages are indicated in Figure 5.1. The corresponding conformities are shown in Table 5.1. The fourth row in the table corresponds to the number of nodes in the circuit that survived the subsequent optimizations and show

	Before fx	After fx	After decomp	After td	After map	After fanopt
X-Conformity	72.73	49.39	57.44	67.71	68.45	100
Y-Conformity	64.88	85.24	78.06	83.83	83.45	100
Average Conformity	68.81	67.31	67.75	75.77	75.95	100
Surviving nodes	22	48	60	115	202	324
Total nodes	29	82	115	509	309	324
% of nodes survived	76	58	52	23	65	100

Table 5.1: X and Y conformity for benchmark circuit *duke2*.

up after fanout optimization. The fifth row corresponds to the total number of internal nodes in the network before the corresponding step. The last row reports to the percentage of total internal nodes that survived through the subsequent synthesis operations.

A number of observations can be made from these tables.

1. As shown in the table, the X-conformity of the initial network (which is a two-level network) is quite high. This is due to the small number of (large) nodes present in the circuit initially. A placement of these nodes essentially captures the interconnection structure of these nodes and the pads. However, the Y-conformity of the initial network is low. This is merely a side effect of standard cell type of placement where Y-location reflect the row number of the cell. Since there are very few nodes in the initial circuit (29 as opposed to 324 in the final circuit), the row structure (e.g., number of rows) is likely to be quite different resulting in low Y-conformity. However, for this experiment it was assumed that placement is not incrementally updated during synthesis. If the placement was updated incrementally by assigning locations during logic synthesis, it is likely that the placement based mechanism can be made more reliable. Specifically, since the conformity is high for the initial circuit, if an appropriate incremental placement mechanism is available, then a placement based mechanism for logic synthesis should be applied beginning at this stage. However, if the incremental placement mechanism not good, an incremental placement of 324 cells derived based on placement of 22 original cells is likely

to be much worse than an independent placement of these 324 cells at the end of synthesis phase. My experiences indicate that, in general, it is difficult to beat existing mathematical programming based placement tools or simulated annealing based placement tools using an incremental placement approach.

2. The X-conformity is significantly lower after extraction. Also, the number of placed and total nodes is still much lower than the final number of placed and total nodes, respectively. This implies that the placement locations at this phase are unreliable and that any incremental placement mechanism based on these location values is likely to mislead the synthesis operations.

3. The average conformity increases after decomposition and increases even further after technology decomposition. At the same time, the number of placed nodes that survive through the subsequent synthesis operations increases. Thus, relatively, it is more appropriate to apply a placement based mechanism after decomposition.

4. The percentage of placed internal nodes that survive through subsequent synthesis operation is highest before extraction. This value decreases monotonically until after technology decomposition and increases thereafter. Typically, lower percentage of surviving placed nodes results in lower values of X-conformity. This indicates that at such intermediate stages the circuit may have a lot of nodes that are going to disappear either due to Boolean optimization (e.g., node simplification) or due to algebraic restructuring (e.g., extraction, decomposition, etc..). When a companion placement is generated based on the network structure at these intermediate stages, it will reflect the corresponding intermediate structure that is going to change significantly during subsequent operations. Thus, relying on a companion solution generated at these stages is likely to misleads the synthesis operations. However, one exception to this is the conformity after technology decomposition operation. Here, in spite of having a lot of nodes that will disappear after technology mapping, the conformity of the surviving nodes is good. The reason for this is that technology decomposition inherently introduces a lot of gates that are eventually disappear as most of them will be covered by corresponding library gates

during technology mapping. However, after technology decomposition, there is no modification of underlying network structure (unlike Boolean operations that may change the immediate supports and hence the transitive fanin/fanout cones of gates). Thus, a companion placement solution generated after technology decomposition can produce quite accurate relative placement of the gates.

This experiment shows that in general a companion placement solution is not very representative of the final placement of gates during intermediate stages of logic synthesis. However, the companion placement solution is reasonably reliable during early stages of logic synthesis, i.e., when the circuit has a small number of large modules. The main reason for this is that when the network has small number of very large nodes, not only there are fewer alternative placement solutions, the relative placements of these modules can be predicted with greater reliability[1]. Unfortunately, in absence of a good incremental placement mechanism, a placement based approach may not produce consistent results even when applied from an earlier phase of logic synthesis. Hence, in next section, I propose some graph structure based measures that attempt to identify inherent routing complexity of the underlying Boolean network in terms of the network structure without relying on a placement solution of the circuit.

[1] In fact, this conclusion can be further extended to application of such placement based approaches to early floor planning strategies where module placements can be predicted (indeed, forced) during early phases of synthesis.

Chapter 6

Structure Based Measures of Post-Layout Costs

Structure based measures are those measures that are derived only from the structure of the underlying Boolean network of a circuit. Since these measures are derived from the network structure, it is intuitive that these measures will not be overly sensitive to the type of placement/routing tools used.

Many graph/structural parameters have a direct impact on the routing cost. These parameters can be classified in two categories: parameters that are dependent on the local structure of the Boolean network; and those that are dependent on the global structure of the Boolean network. Local parameters characterize the routing cost of a gate or a net in the circuit based on the local connections and the local subnetwork structure whereas global parameters characterize the routing cost of the Boolean network as a whole. For example, number of pins on a net constitute a local parameter affecting routing cost. Clearly, a multi-pin net requires larger area to route compared to a two-pin net. It also complicates the subsequent place/route procedures as the net list will contain more global nets. A measure derived from the interconnections of a gate is also a local measure. Another important local parameter is the fanout range of a signal that is defined as the span of the fanouts of a gate in terms of some placement based (e.g., placement positions of the fanouts) or structure based (e.g., logical depths of the fanouts) measure. If fanouts of a gate are within a limited range, the routing length of the output net will be less than that of a net with widely distributed fanouts. The collection of fanout ranges for all the nets in the network constitutes a global parameter affecting the routing. A balanced distribution of these fanout ranges across the network tends to reduce the routing congestion in the network. Global parameters affecting routing also include

topological properties of the underlying Boolean graph. Topological properties like planar thickness or crossing number provide an abstract measure of routability of the corresponding circuit. Circuits with less planar thickness or less crossing number are likely to require less routing area. In fact, a theoretical relation between the layout area and the crossing number is derived in [64]. In this dissertation, routing measures characterizing these parameters, namely, pin count, fanout range, fanout range distribution, planar thickness, crossing number, neighborhood population etc., will be discussed.

As mentioned in Chapter 2, I first identify and analyze the parameters affecting the post-layout area and then derive a reasonable post-layout cost measure using these parameters. In the following chapter, these cost measures are applied to the technology independent phase of logic extraction.

6.1 Post-Layout Area Measures based on Pin Count

To determine the effect of pin-count on the routing length of a net, a simple simple experiment was performed. Three benchmark examples were area-optimized using SIS, placed using GORDIAN [55], and routed using TimberWolf global router [63] and YACR2 [87] channel router. The graph depicting the average netlength as a function of the number of pins on the net is shown in Figure 6.1. The figure confirms that pin-count of a net is directly related to the netlength. However, to effectively minimize the total netlength of a circuit during logic synthesis using pin-counts, the netlengths must be characterized in terms of the pin-counts. I propose such a characterization. It should be noted that instead of estimating absolute netlengths, the emphasis is on estimating relative netlengths to allow a comparative selection during early stages of logic synthesis.

Definition 6.1.1 Equi-perimeter Netlength ratio *of a net with n pins (denoted by $\rho(n)$) is obtained by dividing the netlength of a net with n pins by the netlength of a net with 2 pins given that the two nets establish the same bounding-box.*

For example, in the worst case, the optimal values of $\rho(2) = \rho(3) = 1$ whereas $\rho(4) = 1.5$ as shown in Figure 6.2.

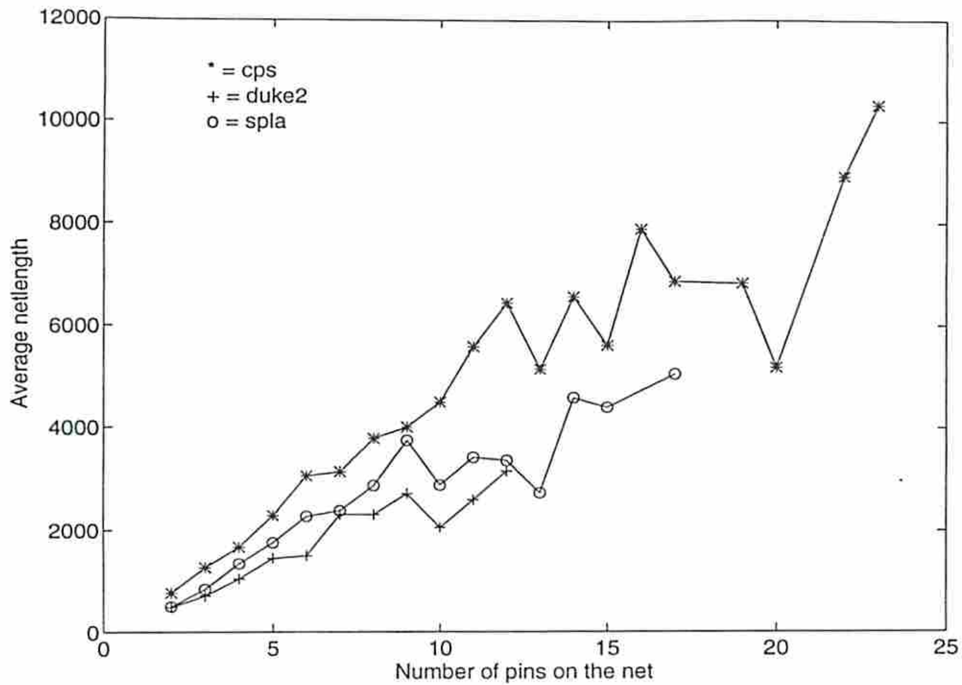


Figure 6.1: Average netlength for different pin counts for three benchmark circuits.

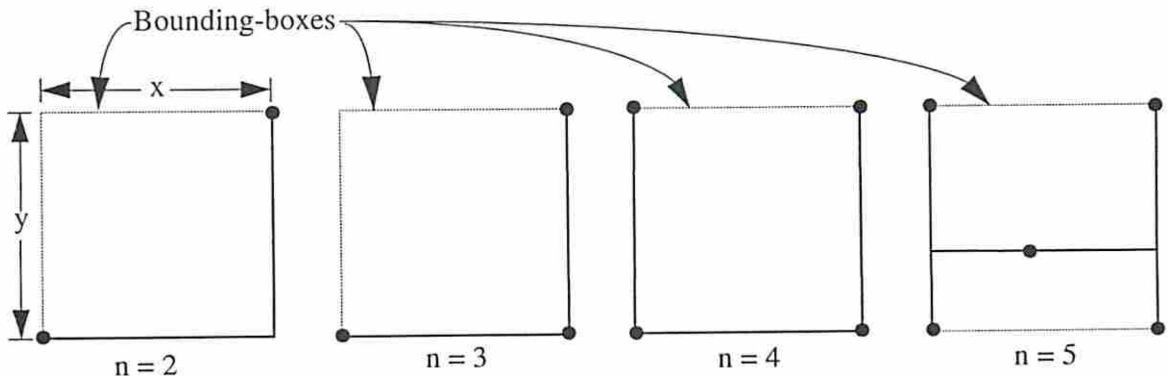


Figure 6.2: An illustration of equi-perimeter netlength ratios.

Definition 6.1.2 Perimeter ratio *of a net with n pins (denoted by $\nu(n)$) is obtained by dividing the perimeter of a net with n pins by the perimeter of a net with 2 pins.*

Based on the above, if $perimeter(2)$ denotes the perimeter of a 2-pin net, the netlength $L(n)$ of an n-pin net is given by

$$L(n) = \frac{perimeter(2)}{2} \cdot \rho(n) \cdot \nu(n)$$

Since $\frac{perimeter(2)}{2}$ is a constant, the relative routing contribution of two nets with pin-count n and n' can be obtained by comparing $\rho(n) \cdot \nu(n)$ with $\rho(n') \cdot \nu(n')$. This measure of relative routing contribution is referred to as normalized netlength of net n.

Definition 6.1.3 Normalized Netlength *of a net with n pins (denoted by $\eta(n)$) is defined as*

$$\eta(n) = \rho(n) \cdot \nu(n) \tag{6.1}$$

In this work, the expected values of $\eta(n)$ for different pin-counts n are used to drive a routing-driven extraction procedure. To calculate expected value of $\eta(n)$, expected values of $\rho(n)$ and $\nu(n)$ must be calculated.

The placement model adopted in this analysis assumes that the chip has width W and Height H in terms of horizontal and vertical pin-pitches, i.e., $W = actual_chip_width/pin_pitch_x$ and $H = actual_chip_height/pin_pitch_y$ where pin_pitch_x is the minimum possible center-to-center distance between two adjacent pins in the X-dimension and pin_pitch_y is the minimum possible center-to-center distance between two adjacent pins in the Y-dimension. This results in a chip layout that has grid-lines determined by pin_pitch_x and pin_pitch_y. Further, it is assumed that each pin has to conform to these grid boundaries, i.e., each pin has to be completely confined within these grid boundaries. It is also assumed that pins are randomly distributed on the chip, independent of other pins. This implies that more than one pins are allowed to share one pin location. Since the bounding-box width and bounding-box height are being estimated separately, this assumption is

n	2	3	4	5	6	7	8	9	10
$\rho(n)$	1	1	3/2	3/2	5/3	7/4	11/6	2	2

Table 6.1: Multiplicative factors to estimate wire lengths given the semi-perimeter of the bounding box.

necessary because x-projections of more than one pins may coincide even tough they have distinct locations due to different y locations.

6.1.1 Estimating Equi-perimeter Netlength Ratios

When the net is routed using a rectilinear steiner tree, Chung and Hwang [15] have exhaustively calculated worst case optimal value of $\rho(n)$, i.e., maximum value of optimal rectilinear steiner tree netlength under all possible pin configurations within a unit size bounding-box, for $n \leq 10$. These values of $\rho(n)$ are reproduced in Table 6.1. They have also shown that

$$lim_{n \to \infty} \rho(n) = \frac{\sqrt{n}+1}{2} \tag{6.2}$$

The above implies that when the nets are routed using optimal rectilinear steiner trees, the equi-perimeter netlength ratio is $O(\sqrt{n})$. Next, the choice of this estimation of $\rho(n)$ which gives the worst case equi-perimeter netlength ratio when nets are routed using optimal rectilinear steiner trees is justified.

Ideally, equi-perimeter netlength ratios should be estimated by enumerating all pin configuration in a given bounding-box and averaging the netlengths obtained by routing each such pin configuration with an optimal rectilinear steiner tree. Apart from being computationally expensive, average netlengths calculated by such an analysis will depend on the bounding-box size. The consequence of assuming the worst case pin configuration to calculate $\rho(n)$ is pessimistic estimates of netlengths. However, the pessimistic values of $\rho(n)$ may be negated to some extent due to the assumption of *optimal steiner* routing trees for each net which gives optimistic estimates since existing routers only approximate optimal steiner trees.

6.1.2 Estimating Perimeter Ratios

To estimate perimeter ratios, the following question must be addressed. How does the net bounding box increase with an increase in pin-count?

Consider a net with n pins. Let $P_{i,i+x}(n)$ denote the probability of all n pins falling within a X-span of $[i, i+x]$. Let $P_{|i,i+x|}(n)$ denote the probability of n pins having an exact X-span of $[i, i+x]$. Given the chip width W, the probability that the net bounding-box has width x, denoted by $X_x(n)$, is

$$X_x(n) = \sum_{i=1}^{W-x} P_{|i,i+x|}(n)$$

Based on this, the expected value of the width $W(n)$ of net bounding-box is given by

$$W(n) = \sum_{x=0}^{x=W-1} x X_x(n) = \sum_{x=0}^{x=W-1} x \sum_{i=1}^{W-x} P_{|i,i+x|}(n) \tag{6.3}$$

Next, $P_{|i,i+x|}(n)$ is calculated. This is given by probability that all n pins fall within $[i, i+x]$ minus the probability that all pins fall within$[i, i+x-1]$ or within $[i+1, i+x]$, i.e.,

$$P_{|i,i+x|}(n) \quad = \quad P_{i,i+x}(n) - P_{i,i+x-1}(n) - P_{i+1,i+x}(n) + P_{i+1,i+x-1}(n)$$

The last term in the above equation is added because both $[i, i+x-1]$ and $[i+1, i+x]$ contain $[i+1, i+x-1]$. Thus, by subtracting $P_{i,i+x-1}(n)$ and $P_{i+1,i+x}(n)$, $P_{i+1,i+x-1}(n)$ has been subtracted twice.

When the pins are uniformly distributed, $P_{i,i+x}(n) = P_{j,j+x}(n)$ and $P_{|i,i+x|}(n) = P_{|j,j+x|}(n)\ \forall\, i, j;\ 1 \le i, j \le W - x$. In this case, the dependence on i can be dropped from $P_{i,i+x}(n)$ and $P_{|i,i+x|}(n)$, representing them as $P_x(n)$ and $P_{|x|}(n)$, respectively. This reduces equation (6.3) to

$$\begin{aligned} W(n) &= \sum_{x=0}^{x=W-1} x \sum_{i=1}^{W-x} P_{|x|}(n) \\ &= \sum_{x=0}^{x=W-1} x(W-x) P_{|x|}(n) \end{aligned} \tag{6.4}$$

Also, when the pins are uniformly distributed, the probability that a pin falls within $[i, i+x]$, given that the maximum possible span is $[1, W]$, is given by $\frac{x+1}{W}$. Since the pin locations are independent of each other, the probability that n pins fall within span $[i, i+x]$, given that the maximum possible span is $[1, W]$, is given by $(\frac{x+1}{W})^n$. Thus, equation (6.4) can be further simplified to

$$\begin{aligned} P_{|x|}(n) &= (\frac{x+1}{W})^n - 2(\frac{x}{W})^n + (\frac{x-1}{W})^n \\ &= \frac{(x+1)^n - 2x^n + (x-1)^n}{W^n} \end{aligned}$$

Substituting this in equation (6.4) gives the expected width of the bounding-box as

$$W(n) = \sum_{x=1}^{x=W-1} x(W-x)\frac{(x+1)^n - 2x^n + (x-1)^n}{W^n} \quad (6.5)$$

This is a closed form expression estimating the width of the bounding box for a net with n pins under a uniform and independent pin distribution. Based on equation (6.5), the following lemma is presented that gives the expected bounding-box width of 2-pin nets. The lemma can be proved by simple manipulations of equation (6.5) after setting $n = 2$.

Lemma 6.1.1

$$W(2) = \frac{W}{3} \quad (6.6)$$

Dividing equation (6.5) by equation (6.6) gives the normalized width, denoted N_w, of bounding-box for each pin-count as given below.

$$N_w(n) = \frac{3}{W}\sum_{x=1}^{x=W-1} x(W-x)\frac{(x+1)^n - 2x^n + (x-1)^n}{W^n}$$

The solution to this equation for different values of W is presented in Table 6.2.

As equation (6.5) indicates, expected width of the bounding-box is dependent on the final chip width W, implying that it is not useful during logic synthesis where the final chip dimensions are not known a-priori. However, as can be seen from Table 6.2, value of $N_w(n)$ is relatively independent of W. Indeed, as shown next, for

n	3	4	5	6	7	8	9	10
$W = 10$	1.5	1.798	1.99	2.13	2.237	2.317	2.38	2.429
$W = 10^2$	1.5	1.8	2	2.14	2.25	2.333	2.4	2.454
$W = 10^3$	1.5	1.8	2	2.14	2.25	2.333	2.4	2.454
$W = 10^4$	1.5	1.8	2	2.14	2.25	2.333	2.4	2.454

Table 6.2: Expected normalized width of the bounding-box.

a given range of W and n, the error can be bound by making (6.5) independent of W by setting $W = \infty$.

Theorem 6.1.2

$$lim_{W\to\infty}\frac{3}{W}\sum_{x=1}^{x=W-1} x(W-x)\frac{(x+1)^n - 2x^n + (x-1)^n}{W^n} = \frac{3(n-1)}{n+1}$$

Proof Using a binomial expansion of $(x+1)^n$ and $(x-1)^n$, the LHS of equation (6.5) is rewritten as

$$\frac{3}{W^{n+1}}\sum_{x=1}^{x=W-1}(xW - x^2)(x^n + C_1{}^n x^{n-1} + C_2{}^n x^{n-2} + \cdots + 1$$
$$-2x^n + x^n - C_1{}^n x^{n-1} + C_2{}^n x^{n-2} + \cdots - 1)$$

After cancelling the corresponding terms, this reduces to

$$\frac{3}{W^{n+1}}\sum_{x=1}^{x=W-1}(xW - x^2)(2C_2{}^n x^{n-2} + C_4{}^n x^{n-4} + \cdots) \tag{6.7}$$

For the time being, focus on the first term, i.e.,

$$\frac{3}{W^{n+1}}\sum_{x=1}^{x=W-1}(xW - x^2)2C_2{}^n x^{n-2}$$

The above can be simplified to

$$\frac{6C_2{}^n}{W^{n+1}}(W\sum_{x=1}^{x=W-1}x^{n-1} - \sum_{x=1}^{x=W-1}x^n)$$

n	3	4	5	6	7	8	9	10
$W = 10$	1.5	1.798	1.99	2.13	2.237	2.317	2.38	2.429
$W = \infty$	1.5	1.8	2	2.14	2.25	2.333	2.4	2.454
$\%Error$	0	0.1	0.2	0.3	0.5	0.7	0.8	1.0

Table 6.3: Expected normalized width as $W \to \infty$ and the percentage error.

As $W \to \infty$, $\sum_{x=1}^{x=W-1} x^{n-1} \to \frac{W^n}{n}$ and $\sum_{x=1}^{x=W-1} x^n \to \frac{W^{n+1}}{n+1}$. Substituting this in the above, we get

$$\frac{6C_2{}^n}{W^{n+1}}(W\frac{W^n}{n} - \frac{W^{n+1}}{n+1}) = \frac{3(n-1)}{n+1}$$

Similarly, for the second term in equation (6.7) we get

$$\frac{6C_4{}^n}{W^{n+1}}(W\frac{W^{n-2}}{n-2} - \frac{W^{n-1}}{n-1}) = \frac{n(n-3)}{12W^2}$$

Thus, the second term and all subsequent terms in equation (6.7) have some positive power of W in denominator, implying that the value of subsequent terms is 0 as $W \to \infty$. ∎

Thus, when W is assumed to be ∞, $\nu(n)$ is given by

$$\nu(n) = \frac{3(n-1)}{n+1} \tag{6.8}$$

Corollary 6.1.3 *For $W \geq 10$, by using $W = \infty$ (i.e., equation (6.7)), the error in estimation of $\nu(n)$ is bounded by 1% when $n \leq 10$, and by 4.7% when $n \leq 40$.*

The above corollary is a direct results of equation (6.7) and the values presented in Table 6.3. As can be seen in Table 6.3, for $W \geq 10$, error in the value of $W(n)$ is bounded by 1% for $n \leq 10$ if $W = \infty$ is used to estimate $W(n)$. Also, for $n \leq 40$ the error in $W(n)$ is bounded by 4.7% as seen in Figure 6.3. Indeed, the error bound can be reduced to negligible amount (i.e., less than 0.01% for $n \leq 10$ and less then 0.05% when $n \leq 40$) if it is guaranteed that $W \geq 100$. However, here $W \geq 10$ is chosen to allow the same numbers to remain applicable for H also since in many standard cell designs H corresponds to the number of rows which may be as little as 10.

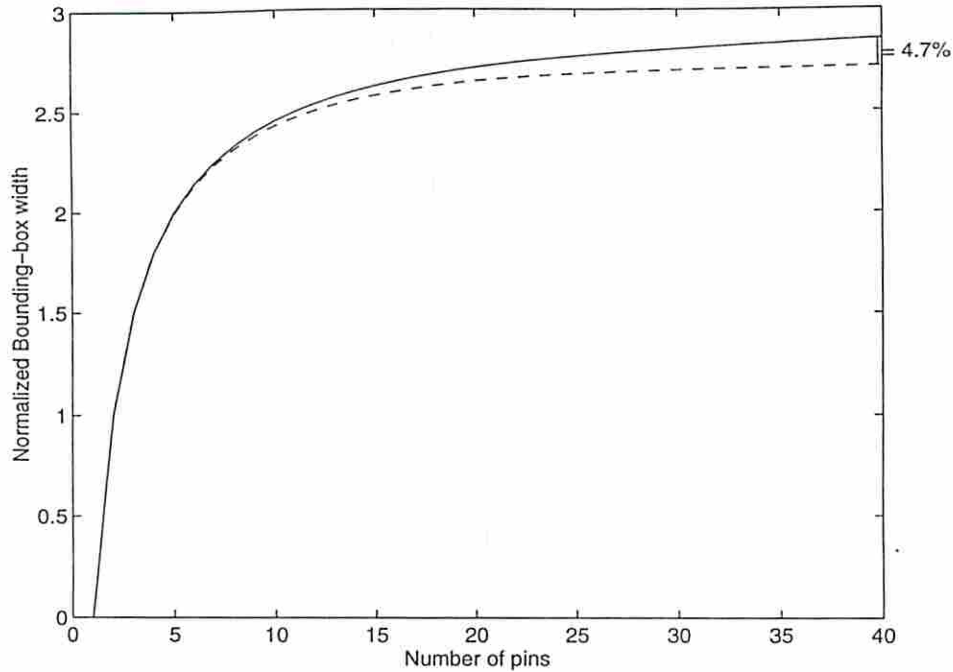


Figure 6.3: Effect of W on $\nu(n)$, $N_w(n)$ and $N_h(n)$. Solid line corresponds to $W = \infty$ whereas dotted line corresponds to $W = 10$.

The analysis required to obtain the expected height of the bounding-box is identical and leads to the same results as in Table 6.2. Thus, expected values of normalized height $N_h(n)$ and perimeter ratio $\nu(n)$ are also identical to the values of $N_w(n)$ presented in Table 6.2.

6.1.3 Estimating Normalized Netlengths

Given the equi-perimeter netlength ratios and perimeter ratios, the normalized netlengths can be calculated using equation (6.1), The following table characterizes the increase in netlength with respect to an increase in the pin-count of the net. This table is generated by multiplying the equi-perimeter netlength ratio for each pin-count (characterized by Table 6.1 and equation (6.2)) with the corresponding perimeter ratio (characterized by equation (6.8)). To generate Table 6.4, $W = \infty$ was chosen which guarantees an error bound of at most 1% when $n \leq 10$ and an error bound of at most 4.7% when $n \leq 40$ for $W \geq 10$. As the experimental results verify, in general, more than 95% of the total netlength is due to nets with pin-count

n	3	4	5	6	7	8	9	10
$\eta(n)$	1.5	2.7	3	3.57	3.94	4.28	4.8	4.91

Table 6.4: Expected normalized netlength as a function of n.

of 10 or less. Thus, an inaccuracy greater than 1% for nets with pin-count greater than 10 does not affect the overall netlength estimation.

The table gives the value of $\eta(n)$ for $n \leq 10$. For $n > 10$, value of $\eta(n)$ can be calculated using the following equation.

$$\eta(n) = \frac{3}{2} \cdot (\sqrt{n} + 1) \cdot \frac{n-1}{n+1} \tag{6.9}$$

Table 6.4 and equation (6.9) characterize the relative netlengths of nets with different pin-counts. This function only depends on n and hence, can be used during logic synthesis. In particular, the following objective function can be minimized

$$R(N) = \sum_{i=2}^{n_{max}} |N_i| \eta(i) \tag{6.10}$$

where N_i represents the set of nets with i pins and n_{max} is the maximum pin-count.

However, before adopting this objective function for optimization during logic synthesis, in the next subsection, the accuracy of η is experimentally verified.

6.1.4 Evaluation of the Netlength Cost Function

To verify the accuracy of the normalized netlength cost function the expected relative netlength were compared with the actual average netlength obtained on the benchmark circuits generated as explained at the beginning of this section. Since only the relative netlength contribution are captured and not the absolute netlength, the curve produced by Table 6.4 must be fitted to the actual curve by multiplying the values in Table 6.4 by some normalizing constant. The constant used here is the total netlength of all nets divided by the total normalized netlength of these nets. This is presented in Figure 6.4.

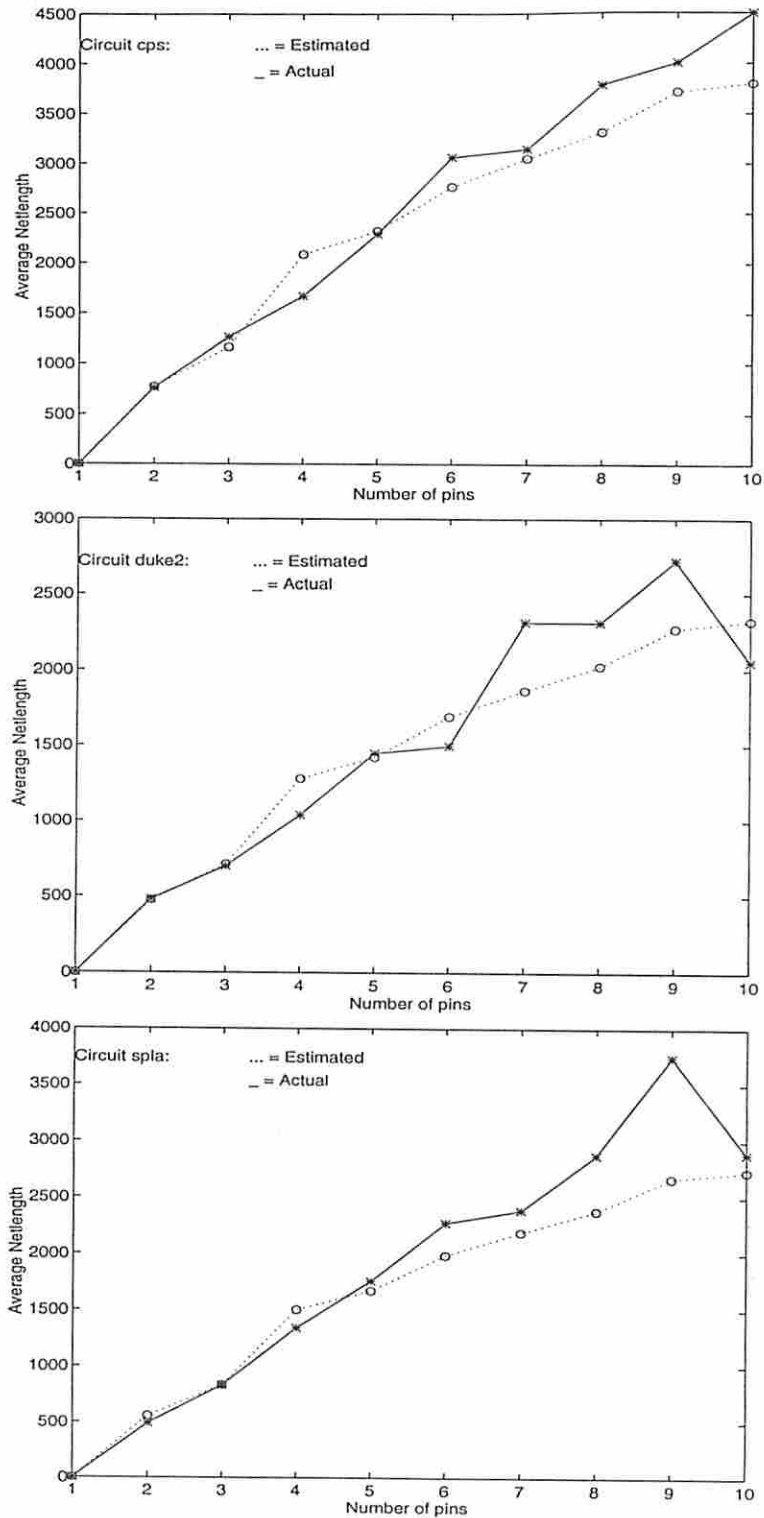


Figure 6.4: Estimated Vs. Actual average netlengths.

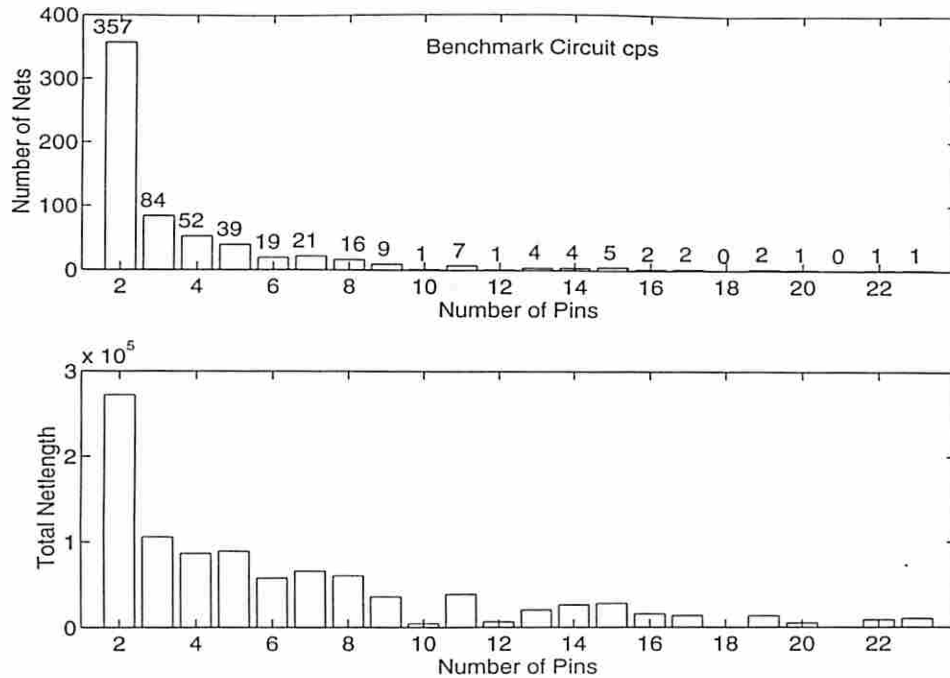


Figure 6.5: Number of nets with different pin count and sum of their netlengths for circuit *cps*.

As can be seen from Figure 6.4, estimated netlengths track the actual netlengths quite well. For higher value of n, the actual netlength deviations increase with respect to the expected netlength. This can be attributed to the following. The assumption in calculation of $\rho(n)$ that an optimal rectilinear steiner tree is used to route a net does not hold for larger nets, with larger nets being routed with increasingly sub-optimal routing trees by the existing routing tools. However, As can be seen in Figure 6.5, almost 90-95% of the nets in a circuit have a pin-count of 10 or less. Also, these nets contribute more than 80% of the total netlength of the circuit. Thus, an increased deviation of expected netlengths from the actual netlengths for nets with higher pin-count has does not affect the total netlength significantly.

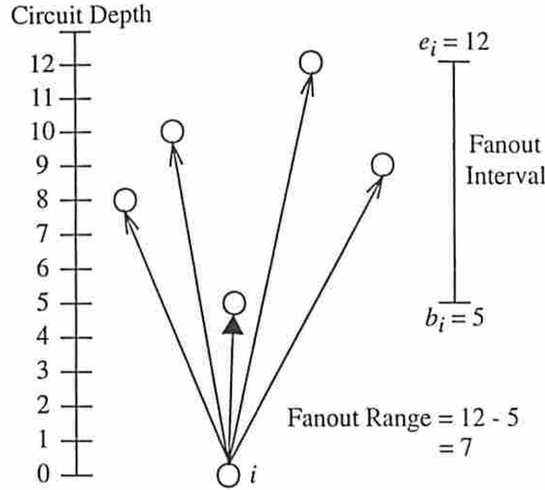


Figure 6.6: Illustration of fanout ranges and fanout intervals.

6.2 Post-Layout Area Measures based on Fanout Ranges

An ideal scenario for minimizing the routing overhead is when the routing length of each net is minimized and the routing is equally distributed across the chip. Minimization of the routing lengths can be achieved by localizing the fanout ranges during logic synthesis. Localization implies, bringing all the gates belonging to a net close to each other in terms of some geometrical or topological distance metric. During logic synthesis the network is still being modified, making it difficult to arrive at a reliable mechanism for obtaining an estimation of post-layout placement distances between gates. Hence, during technology independent phase, structural distances are used as a representative of the post-layout placement distances.

Let O_i denote the set of fanouts of a gate i and d_i denote the depth of gate i ($d_i = 0$ if i is a primary input).

Definition 6.2.1 *The* **fanout interval** *of a gate i, is given by* $[b_i, e_i] = [min_{j \in O_i}\{d_j\}, max_{j \in O_i}\{d_j\}]$ *where b_i and e_i correspond to the beginning and the end of the fanout interval, respectively. The* **fanout range** *R_i of gate i is calculated as* $R_i = e_i - b_i$

These definitions are illustrated in Figure 6.6.

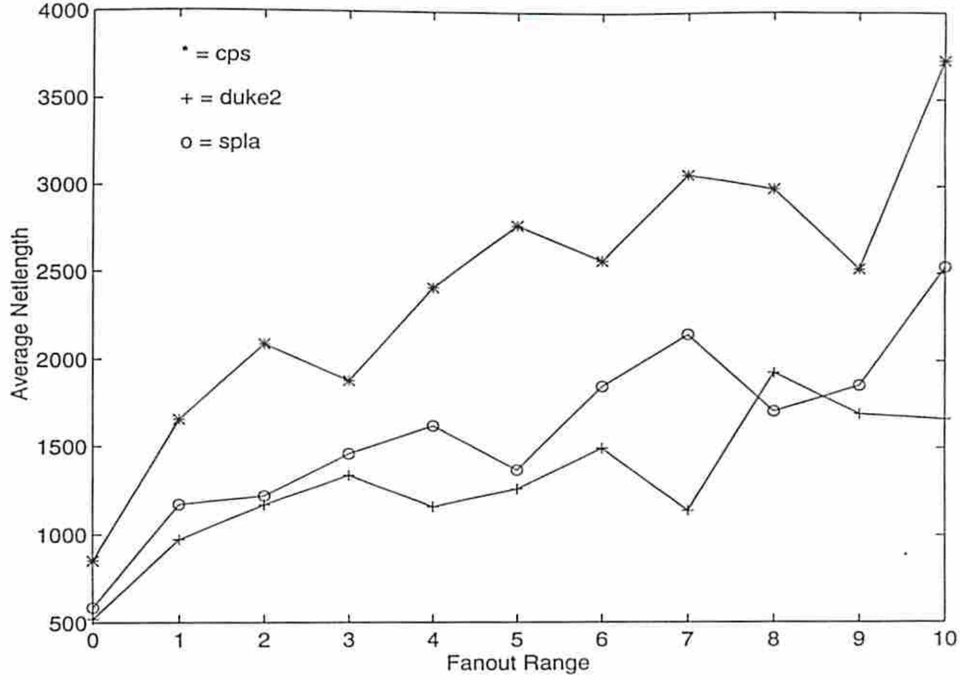


Figure 6.7: Average netlengths for different fanout ranges.

Alternately, a fanout range function that emphasizes more on fanout distributions as opposed to exact upper and lower bounds can be devised based on average fanout depth and standard derivation of the fanout depths.

Average fanout depth of gate i, denoted by D_i is given by,

$$D_i = \frac{\sum_{j \in O_i}\{d_j\}}{|O_i|}.$$

Standard deviation of fanout depths for gate i, denoted by σ_i is given by

$$\sqrt{\frac{\sum_{j \in O_i}(d_j - D_i)^2}{|O_i| - 1}}.$$

Then, the corresponding fanout interval is given by $[D_i - \sigma_i, D_i + \sigma_i]$.

6.2.1 Evaluation of Fanout Ranges

To confirm the correlation between fanout ranges and actual wiring cost, we calculated the wire lengths of nets with different fanout ranges after placement and

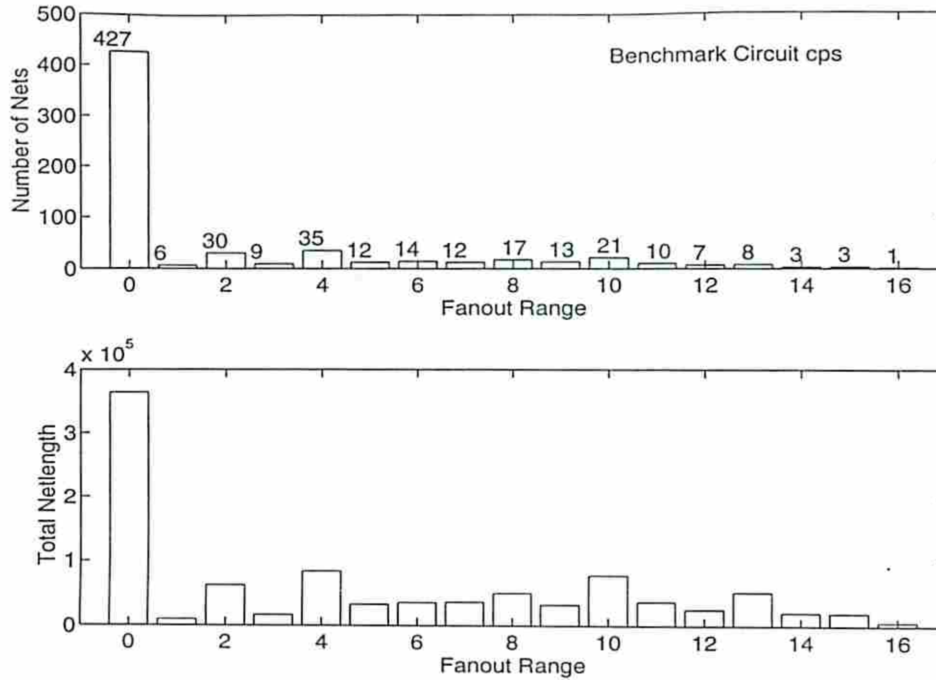


Figure 6.8: Number of nets with different fanout ranges and sum of their netlengths in circuit *cps*.

routing. These circuits were area-optimized using SIS, placed using GORDIAN [55], and routed using TimberWolf global router [63] and YACR2 [87] channel router. The results for three benchmark circuits are presented in Figure 6.7. As shown in the graph in Figure 6.7, netlength increases with an increase in the fanout range. The correlation is especially good for nets with low fanout range. For higher values of fanout ranges, netlengths do not increase monotonically with an increase in fanout ranges. This indicates that for larger nets, placement randomizes the effect of fanout ranges on the netlength. However, as shown in Figure 6.8, even for large circuits, about 95% of nets have fanout range of 10 or less. Moreover, these nets contribute more than 90% of the total netlength. Hence, it is appropriate to use fanout ranges to capture the relative routing costs during logic synthesis.

6.2.2 Signal Localization using Fanout Ranges

Let $R(N)$ denote the routing cost of a Boolean network N. The concept of fanout ranges can be used to estimate this routing cost as

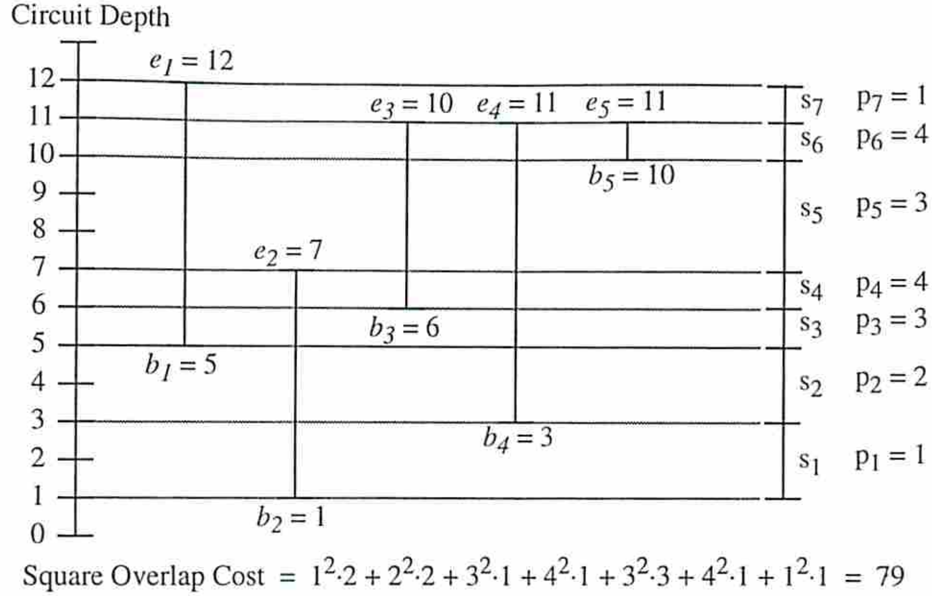


Figure 6.9: Illustration of the square overlap function.

$$R(N) = \sum_{i \in N} \{max_{j \in O_i}\{d_j\} - min_{j \in O_i}\{d_j\}\} \tag{6.11}$$

In this case, the fanout ranges of all the gates in the network are added to determine the routing cost of the network.

However, this objective function does not attempt to distribute the fanout ranges across the circuit. Uniform distribution of the signals across the chip tends to reduce congestion in the circuit. Apart from creating routing difficulties, congestion results in dead area in other parts of the chip, increasing the total chip area significantly. I propose a modified objective function that achieves minimization as well as distribution of fanout ranges simultaneously.

6.2.3 Signal Distribution using Fanout Ranges

Suppose the fanout interval $[b_i, e_i]$ for each gate $i \in N$ is known. Let M_N denote the sorted set union of all b_i's and e_i's, and S_N denote the set of segments formed between two adjacent elements in M_N. Furthermore, let p_r denote the number of fanout intervals that overlap with segment $s_r \in S_N$, and l_r denote the length of

segment s_r (see Figure 6.9). Based on this, an abstract measure of routing cost $R(N)$ of the set of gates N is defined. I experimented with two cost functions.

The Fanout Interval Overlap Function: Since reducing the fanout interval overlap tends to uniformly distribute the signals across the network, the following cost function is proposed:

$$R(N) = \sum_{s_r \in S_N} F(p_r, l_r) = \sum_{s_r \in S_N} {p_r}^2 l_r \qquad (6.12)$$

This function reduces the fanout range overlap by minimizing the sum of the square of overlaps. Note that $F(p_r, l_r)$ can be any other function that discourages interval overlaps[1].

The Interval Density Function: Minimizing the maximum number of intervals that overlap in a segment reduces the density of that segment. The corresponding cost function is given by

$$R(N) = \max_{s_r \in S_N} p_r \qquad (6.13)$$

Assuming that all the other fanout intervals remain the same, reducing the fanout range of one gate will result in a monotonic decrease in the value of the above functions represented by equations (6.11), (6.12) and (6.13). However, keeping all other fanout intervals fixed, if the fanout interval of one gate is gradually shifted such that the fanout range overlap is reduced (i.e., signals are more distributed) the value of only equations (6.12) and (6.13) decreases monotonically whereas value of (6.11) remains unchanged. Hence, minimization of cost function based on equation (6.11) will be a result of improved signal localization whereas a minimization of equation (6.12) and (6.13) will be as a result of either improved signal distribution or improved signal localization or a combination of the two.

[1]In fact, defining $F(p_r, l_r) = p_r l_r$ gives the same cost function as given by equation 6.11.

6.3 Post-Layout Area Measures based on Other Graph Properties

Topological properties of a Boolean graph, specifically, planarity properties are likely to have an impact on the routing complexity of the corresponding circuit. However, planarity properties of graphs may not be directly applicable to CAD applications. For example, from a pure graph theoretical point of view, a planar graph is embeddable in a plane without any edge crossing whereas a bad embedding (i.e, a bad placement) along with a manhattan style multi-layer routing of the same graph may introduce crossings. However, in general, it is reasonable to assume that choosing a mechanism that either does not add to the non-planarity of the circuit, or reduces the non-planarity of the circuit will improve routing complexity.

Measurement of non-planarity of a graph is one of the most difficult problems. Most of the results about non-planarity measure of graphs are in terms of upper bounds and lower bounds and a set of conjectures. Some of these results are reproduced here from graph theory literature. Relevant measures of non-planarity are defined below [124].

Definition 6.3.1 Planar thickness *of a graph G, denoted $t(G)$, is the minimum number of planar graphs, union of which is G.*

Definition 6.3.2 Crossing number *of a graph G, denoted $\nu(G)$, is the minimum number of crossings in a drawing of G on a plane.*

Apart from these measures, measures like genus of a graph, coarseness of a graph, or page number of a graph are also used to measure non-planarity of a graph [124].

Planar Thickness: From Euler's identity, if a graph $G(V, E)$ is planar, it has at most $3n - 6$ edges, where $n = |V|$. Hence, planar thickness of a graph is upper bounded by $e/3n - 6$, where $e = |E|$. It is difficult to arrive at a tighter bound for general graphs. However, for complete graphs the planar thickness is more precisely calculated. For example, the planar thickness of a complete graph with n vertices, denoted K_n, is given by $t(K_n) \geq \lfloor \frac{n+7}{6} \rfloor$, with the equality not being satisfied only for $n = 9$ and $n = 10$. Planar thickness of a complete

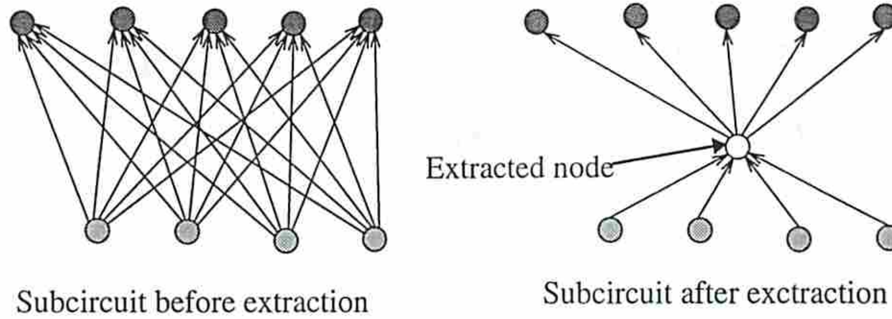


Figure 6.10: Illustration of an extraction.

bipartite graph with sets of m and n vertices, denoted by $K_{m,n}$ is given by $t(K_{m,n}) \geq \lceil mn/(2m+2n-4) \rceil$, with equality verified up to $K_{19,29}$. This is particularly applicable for the operation of logic extraction procedure since, as shown in Figure 6.10, every extraction converts a bipartite complete graph into a planar star graph.

Crossing Number: Crossing number of a graph is a more powerful measure of non-planarity as it grows relatively faster with increase in graph size. Also, it is a more realistic routing measure as every crossing point can potentially introduce a via in the final routing. In fact, a theoretical relation between the layout area and the crossing number is derived in [64]. It has been conjectured by Erdős [22] that $C_1 e^3/n^2 \leq \nu(G) \leq C_2 e^3/n^2$ where C_1 and C_2 are constants. This provides a very good abstract measure as it implies that for reducing crossing number, it is desirable to reduce the ratio e^3/n^2. For a complete graph and for complete bipartite graph, sets of tight upper and lower bounds are determined. Guy [38] proved that for a complete graph $\lceil \frac{n(n-1)(n-2)(n-3)}{80} \rceil \leq \nu(K_n) \leq \frac{1}{4} \lfloor \frac{n}{2} \rfloor \lfloor \frac{n-1}{2} \rfloor \lfloor \frac{n-2}{2} \rfloor \lfloor \frac{n-3}{2} \rfloor$, with the upper bound being optimal for $n \leq 10$. For a complete bipartite graph, Zarankiewicz' conjecture that $\nu(K_{m,n}) = \lfloor \frac{n}{2} \rfloor \lfloor \frac{n-1}{2} \rfloor \lfloor \frac{m}{2} \rfloor \lfloor \frac{m-1}{2} \rfloor$ has been proven for $min(m,n) \leq 6$ by Kleitman [56]. Guy [39] obtained an upper bound concluding that $\lceil \frac{n(n-1)m(m-1)}{20} \rceil \leq \nu(K_{m,n}) \leq \lfloor \frac{n}{2} \rfloor \lfloor \frac{n-1}{2} \rfloor \lfloor \frac{m}{2} \rfloor \lfloor \frac{m-1}{2} \rfloor$. Similar work has been done for rectilinear crossing number, i.e. crossing number of a graph when edges are required to be of manhattan type.

Properties relating to the connectivity of the underlying Boolean network is also an important parameter affecting routing. Connectivity of a Boolean graph can be captured using the neighborhood population as described next.

Definition 6.3.3 **d-Neighborhood population** *of a gate g is the set of gates that are reachable via at most d nets from gate g.*

Thus, 1-neighborhood population (also referred to as neighborhood population) of a gate g corresponds to the set of gates that share at least one net with gate g. Note that the direction of the net is irrelevant. Also, note that neighborhood population of a net can be defined similarly on the dual graph of the Boolean circuit graph. Intuitively, the nets with high neighborhood population are likely to be more difficult to route and hence are likely to consume more routing resources. Likewise, a graph that has greater average neighborhood population is likely to have a greater routing contribution than the graph with lower value of average neighborhood population.

Though the planarity based and connectivity based graph parameters are as promising as the normalized netlength and fanout range based parameters, in this thesis, I have focused only on normalized netlength and fanout range based parameters. In the next chapter, these routing cost measures are optimized during the operation of logic extraction. The effectiveness of planarity based parameters and connectivity based parameters will be explored further in future.

Chapter 7

Application of Structure Based Measures to Logic Synthesis

Before presenting my approaches for routing driven extraction, a brief introduction to the operation of extraction is presented. *Logic extraction* is the process of identifying subexpressions common to two or more functions, which are then extracted as intermediate nodes in a multi-level circuit. The goal of extraction is to minimize the chip area by sharing logic across the network. Literal count has been traditionally used as the objective function during extraction as it correlates well with the gate area.

Initial work on extraction was reported by Roth and Karp [89] and by Lawler [61]. Both these approaches apply Boolean techniques for extraction. More recently, Boolean decomposition/extraction techniques based on OBBDs have been proposed [60]. However, the computational complexity of these techniques render them impractical for large circuits. Hence, recent approaches use the following procedure along with algebraic division methods to derive common divisors [7, 5, 86].

```
Algorithm EXTRACT(𝒩)
𝒩 is a given Boolean network
01 begin
02   Generate candidate divisors G
03   do
04     BestLiteralSavings = 0
05     BestDivisor = NULL
06     ForEachDivisor D ∈ G
07       if LiteralSavings(D) ≥ BestLiteralSavings
08         BestDivisor = D
09         BestLiteralSavings = LiteralSavings(D)
10     Divide the Boolean equations by BestDivisor
11     Update the weight of remaining candidate divisors
12   While A candidate divisor of sufficient merit exist
13 end
```

Initially, all candidate divisors associated with nodes in the network are generated and common divisors are detected. A divisor may consist of a single cube or multiple number of cubes. A multiple cube divisor that can not be further divided by a single cube, i.e., cube-free multiple cube divisors, are called kernels. The quotient of the division process is also referred to as a *cokernel*. A cost value is then assigned to each divisor. This cost value has traditionally been the literal savings of the divisor. A divisor that has the best cost is then selected and introduced in the network. After dividing the fanout nodes of a divisor by the selected divisor, the cost of the remaining candidate divisors might change. Hence, it is necessary to update the weight of remaining candidate divisors after each extraction.

For example, consider $f = abeg+aceg+deg+h$. This function can be divided by $d_1 = b+c$, $d_2 = ab+ac+d$, and $d_3 = ab+ac$ to yield $f = ad_1eg+def+h$, $f = d_2eg+h$, and $f = d_3+deg+h$, respectively. In the above example d_1 and d_3 are double divisors while d_3 is a three cube divisor. Also, d_3 is not a kernel as it can be further divided by the single cube a. Cokernels of kernel d_1 and d_2 above are aeg and eg respectively. It should be noted that a cokernel is, in fact, a single cube divisor of the function.

A kernel is of level 0 if it contains no kernels except itself. A kernel is of level n if it contains at least one kernel of level $n-1$ and no kernels of level n of higher. Since d_2 above can be represented as $ad_1 + d$, d_2 is of level 1. However, d_1 is a kernel of level 0.

Kernel based decomposition and a mechanism to derive and extract common kernels (based on rectangle covering problem) was first proposed by Brayton et al. [7, 5]. Rajski and Vasudevamurthy [86] simplified the problem by considering only cube-free double divisors, i.e., kernels with two cubes and single divisors with two literals[1]. A double divisor is generated by intersecting two cubes with at least one literal in common and then taking the sum of product of the remaining literals in each cube. Literals belonging to the intersection form the cokernel of the corresponding occurrence of the divisor. The restriction to double divisors reduces the possible number of divisors from exponential (in case of a general kernelization procedure) to polynomial simply by reducing the granularity of each divisor. In case of single divisors, all possible two literal single cube divisors are generated by performing a pairwise intersection of the set of cubes in which the literals appear. This approach (which is also referred to as the "fast-extract" approach) improves the run time while not sacrificing the quality of resultant circuits. In my approach, the greedy scheme of algorithm EXTRACT was used with all divisors restricted to be either double divisors, or single divisors with two literals as in [86].

Fact 7.0.1 *Each divisor identified by a fast-extract based mechanism is a level-0 kernel.*

This implies that in a fast-extract based mechanism, no further kernel extraction from an extracted node is possible. In fact, this is true for any extraction algorithm that extracts only level-0 kernels.

From the list of candidate divisors, conventional approaches choose a divisor that provided the best literal savings whereas I select a "good" divisor that optimizes the routing cost. A "good" divisor is selected from the list of candidate divisors based on their literal savings potential to ensure that the reduction of routing area is not achieved at the cost of a substantial increase in the active area.

[1]The quotient corresponding to a double divisor is referred as the *base* in their work. I follow their convention and refer to all *cokernels* as *base*.

There are two basic strategies to select "good" divisors from the list of candidate divisors: if maximum literal savings from any divisor is M, (1) select all divisors with literal savings within $p\%$ of M; (2) select the k best literal saving divisors. With the percentage based selection, the emphasis can be shifted from literal driven to routing driven by changing the percentage value. Of course, increased value of p results in increased run time whereas selecting from a constant number of divisors maintains the time complexity of the algorithm. In these experiments, the smaller of the top 20 divisors and the divisors with top 10% literal savings were used.

```
Algorithm ROUTINGDRIVEN_EXTRACT(N, P)
N is a given Boolean network
P is % value to identify good divisors
01 begin
02   Generate candidate divisors G
03   do
04     BestLiteralSavings = 0
05     BestRoutingCost = −∞
06     BestDivisor = NULL
07     ForEachDivisor D ∈ G
08       if LiteralSavings(D) ≥ 1 − P/100*BestLiteralSavings
09         if LiteralSavings(D) ≥ BestLiteralSavings
10           BestLiteralSavings = LiteralSavings(D)
11         if RoutingCost(D) ≤ BestRoutingCost
12           BestDivisor = D
13           BestRoutingCost = RoutingCost(D)
14     Divide the Boolean equations by BestDivisor
15     Update the cost of remaining candidate divisors
16   While a candidate of sufficient merit exists
17 end
```

A generic mechanism to perform the routing driven extraction based on the routing cost function proposed in Chapter 6 is given in algorithm ROUTINGDRIVEN_EXTRACT. Line 11 of the algorithm was implemented to report the corresponding routing cost. It should be noted that if $|G|$ = constant and if complexity of $RoutingCost(D)$ is $O(1)$, algorithm ROUTINGDRIVEN_EXTRACT has the same complexity as algorithm EXTRACT. Since the time complexity of each extraction in my approach is $|G|TimeComp(RoutingCost)$ as compared to $O(1)$ for "fast extract", the overall time complexity of ROUTINGDRIVEN_EXTRACT can be given by *TimeComp(*EXTRACT*)*$*|G|*$*TimeComp(RoutingCost)*

7.1 Pin Count Based Extraction

To perform pin count based extraction, simple modifications were made to the cost function used during extraction as follows. Assume that a double divisor D consists of set of literals $L_D = \{l_1, l_2, \ldots, l_m\}$, each of which appears $a_1, a_2, \ldots, a_m$ times in the circuit before the extraction of D. If the new divisor D appears a_D times in the circuit after the extraction, then appearance of each $l_i \in L_D$ is reduced from a_i to $a_i - a_D + 1$, which will reduce the routing cost of these literals. However, a new node D with $a_D + 1$ pins is introduced. Apart from this, if B_k denotes the set of literals in the base of the kth appearance of divisor D, where $1 \leq k \leq a_D$, then for every literal $l_j{}^k \in B_k, 1 \leq k \leq a_D$, the original appearance of the literal (denoted $a_j{}^k$) is reduced by one (since D is a double divisor).

7.1.1 Minimizing Normalized Netlengths during Extraction

The netlength cost of a divisor is calculated as the change in the total netlength of the circuit due to the extraction. Suppose as a result of extraction, a node D with a_D outputs is introduced in the original circuit N. Let the resultant circuit be denoted by N'. The circuit routing cost before and after extraction is calculated using equation (6.10). Then, $\Delta(D)$ needs to be calculated where $\Delta(D) = R(N') - R(N)$. Apart from the additional routing cost of a new net corresponding to the output of D, pin-counts of already existing nets might also change. Hence, $\Delta(D) \neq$

$\eta(a_D)$. However, by identifying all the nets whose pin-count might change, $\Delta(D)$ is calculated efficiently as explained next.

For every candidate divisor, the routing cost before extraction is calculated as

$$\sum_{l_i \in L_D} \eta(a_i + 1) + \sum_{l_j{}^k \in B_k, 1 \le k \le a_D} \eta(a_j{}^k + 1)$$

After the extraction of a double divisor, routing cost is

$$\sum_{l_i \in L_D} \eta(a_i - a_D + 2) + \sum_{l_j{}^k \in B_k, 1 \le k \le a_D} \eta(a_j{}^k) + \eta(a_D + 1)$$

Hence, the change in routing cost due to the extraction, namely, $\Delta(D)$, is given by

$$\begin{aligned} \Delta(D) \quad = \quad & \sum_{l_i \in L_D} \{\eta(a_i + 1) - \eta(a_i - a_D + 2)\} + \\ & \sum_{l_j{}^k \in B_k, 1 \le k \le a_D} \{\eta(a_j{}^k + 1) - \eta(a_j{}^k)\} - \eta(a_D + 1) \end{aligned} \qquad (7.1)$$

Or in case of a single divisor

$$\Delta(D) = \sum_{l_i \in L_D} \{\eta(a_i + 1) - \eta(a_i - a_D + 1)\} - \eta(a_D + 1) \qquad (7.2)$$

This extraction procedure based on the pin count measure attempts to minimize the routing cost of the final circuit by choosing a divisor with maximum route savings during each iteration of the greedy heuristic. The experimental results are presented in the next section.

It should be noted that the traditional area cost, namely, the number of literals saved due to extraction, can be obtained from divisor costs given by equations (7.1) and (7.2) simply by removing the η parameter (along with 1 pin corresponding to the net source introduced for calculating η). For example, equation (7.1) reduces to

$$\begin{aligned} & \sum_{l_i \in L_D} \{a_i - a_i + a_D - 1\} + \sum_{l_j{}^k \in B_k, 1 \le k \le a_D} \{a_j{}^k - a_j{}^k + 1\} - a_D \\ & \quad = \sum_{l_i \in L_D} \{a_D - 1\} + \sum_{l_j{}^k \in B_k, 1 \le k \le a_D} \{1\} - a_D \end{aligned}$$

$$= |L_D| * (a_D - 1) + \sum_{k, 1 \leq k \leq a_D} \{|B_k|\} - a_D$$

which is exactly the literal savings of a double divisor.

Likewise, equation (7.2) reduces to

$$\sum_{l_i \in L_D} \{a_i - a_i + a_D\} - a_D \quad = \quad |L_D| * a_D - a_D = a_D * (|L_D| - 1)$$

Thus, the only difference in calculating equations (7.1) and (7.2) with respect to the literal savings objective function is that, the values of variables $a_i, \forall l_i \in L_D$ and $a_j{}^k, \forall l_j{}^k \in B_k, 1 \leq k \leq a_D$ must be known. In other words, the original number of occurrences in the circuit for the literals appearing in the divisor as well as for literals appearing in the base of each occurrence of the candidate divisor must be known. This information can be pre-calculated before each extraction iteration in $O(E)$ time along with the circuit update in Line 13 and/or 14 and stored at each node in the circuit. This implies that time complexity of $RoutingCost(D)$ is same as the time complexity of $LiteralSavings(D)$. Thus, the overall time complexity of algorithm ROUTINGDRIVEN_EXTRACT minimizing a normalized netlength based routing cost is identical to the time complexity of algorithm EXTRACT.

7.1.2 Experimental Results

Before presenting experimental results, an issue that demanded attention during the implementation needs to be addressed. When a node/divisor has multiple occurrences in the same node, the question arises whether this should be counted as one pin or as multiple pins to calculate pin-count based routing cost? I believe that it is more appropriate to consider multiple occurrences in the same node as a single pin on the corresponding net. This amounts to using the number of fanouts of a node to calculate $\eta(n)$ instead of identifying and accounting for each occurrence of the node in each of its fanout. The main justification for this is that after the circuit is mapped onto the library gates, even for multiple occurrence in the gate, connection to only a single pin is required. Also, with this approach, an extra incentive is given to the extraction procedure to extract a divisor that reduces the immediate support of the fanouts by completely extracting out some variables. The experimental results

Circuit	Routing Area	Gate Area	Chip Area	Delay
b12	165	844	249	9.63
cps	10787	1299	12086	56.86
duke2	1797	474	2271	23.77
ex1010	40066	3112	43178	205.78
ex4	1357	518	1876	13.83
misex3c	2064	550	2614	46.67
pdc	1951	504	2455	20.95
rd84	275	148	423	14.28
spla	3313	727	4040	26.12
Z5xp1	332	140	472	36.77
Z9sym	691	221	912	15.04
alu4	2968	628	3597	32.59
apex2	1289	382	1671	18.88
apex3	18217	1755	19972	78.10
clip	342	146	488	16.45
misex3	3881	813	4694	33.47
seq	17071	1917	18989	61.02

Table 7.1: Results using *script.rugged* for comparison with normalized netlength based extraction.

verify that using the number of fanouts is more effective than using the number of occurrences. An additional advantage is that now there is no need to calculate $\eta(n)$ for the literals in the base of a divisor because the number of fanouts for the literals in the base remain unchanged subsequent to an extraction.

The *PIn-count based eXtraction* (PIX) procedure was implemented within the SIS synthesis environment. Before presenting results with PIX, the results obtained by optimizing circuits in SIS using the "script.rugged" (which uses "fast_extract" to perform extraction)[2] are produced in Table 7.1.

[2]Results produced in this dissertation using "script.rugged" for one set of results may not be identical to the results produced by using "script.rugged" for another set of results. These differences are caused due to differences in gate libraries used to map the circuit, differences in placement and routing tools used, different placement of pads, randomness of TimberWolf etc..

Circuit	Routing Area	Gate Area	Chip Area	Delay
b12	1.01	1.10	1.04	1.07
cps	1.03	1.11	1.04	0.76
duke2	0.90	0.88	0.90	1.23
ex1010	0.93	0.94	0.93	1.02
ex4	0.96	1.00	0.97	0.98
misex3c	0.92	1.02	0.94	0.95
pdc	0.80	0.89	0.82	1.05
rd84	0.97	0.91	0.95	1.11
spla	0.90	0.93	0.91	1.03
Z5xp1	0.79	0.96	0.84	1.00
Z9sym	0.70	0.80	0.73	0.95
alu4	0.91	1.01	0.93	0.93
apex2	0.86	0.95	0.88	1.10
apex3	0.95	0.98	0.95	1.07
clip	0.84	0.98	0.88	0.96
misex3	0.94	0.96	0.94	1.14
seq	0.92	1.05	0.93	0.85
AVERAGE	0.90	0.97	0.92	1.01

Table 7.2: Results using *script.rugged.pix* (normalized by corresponding values in Table 7.1).

To generate the results, the circuits were mapped using the SIS mapper with LIB2 gate library in delay mode, placed using GORDIAN [55] and routed using TimberWolf global router [63] and YACR2 detailed router [87]. Results presented in Tables 7.1 are used as a basis for comparison with results obtained with PIX.

In the experiments with PIX, traditional extraction operation in *script.rugged*, namely, "fast_extract", was replaced by PIX. This modified script is referred to as "script.rugged.pix". Other operations in *script.rugged* were kept intact. The results of this approach are given in Table 7.2. Each entry in the table is normalized with respect to the corresponding entry in Table 7.1. As can be seen, on average, 10% improvement in routing area and 8% improvement in chip area was obtained for about the same delay.

7.2 Fanout Range Based Extraction

For each routing cost described by equations (6.11)-(6.13), a corresponding routing-driven extraction procedure was implemented. However, synthesis based on minimizing the overlap density as in equation (6.13) is more appropriate when there is a direct correlation between the maximum density and maximum number of vertical tracks available, i.e., for a routing-driven version of FPGA decomposition. Since an extraction procedure based on fast extraction is not appropriate for FPGA decomposition, only the results obtained with implementations corresponding to equations (6.11) and (6.12) are reported here.

7.2.1 Updating the Fanout Ranges during Extraction

Calculating the routing cost based on equations (6.11) and (6.12) requires a recalculation of fanout ranges of the nodes in the network for each candidate divisor.

Lemma 7.2.1 *If the depth of a node changes after extraction, it will increase by one.*

Proof Follows from the monotone speed-up property of a unit delay model and the fact that a depth increase of one for a fanin of a node can lead to a depth increase of at most one for the node. ■

Extracting a divisor d may only change the depth of the nodes in its transitive fanout cone TFO_d. Hence, fanout range of a node may change only if it fans out to a node in this TFO_d. The transitive fanout cone of the extracted node is recursively traversed until a node whose depth does not change due to the extraction is encountered. After identifying the nodes whose depth changes, i.e., increases by one, the immediate fanins of these nodes are checked to update the ranges of these nodes. In the worst case, this might require a complete depth update of the network and a traversal of each fanin connection resulting in the time complexity of $O(E)$ for a Boolean graph $G(V, E)$. This is illustrated in Figure 7.1. However, this range update can be done efficiently by considering only the *depth determining edges* (DD edges) as explained next.

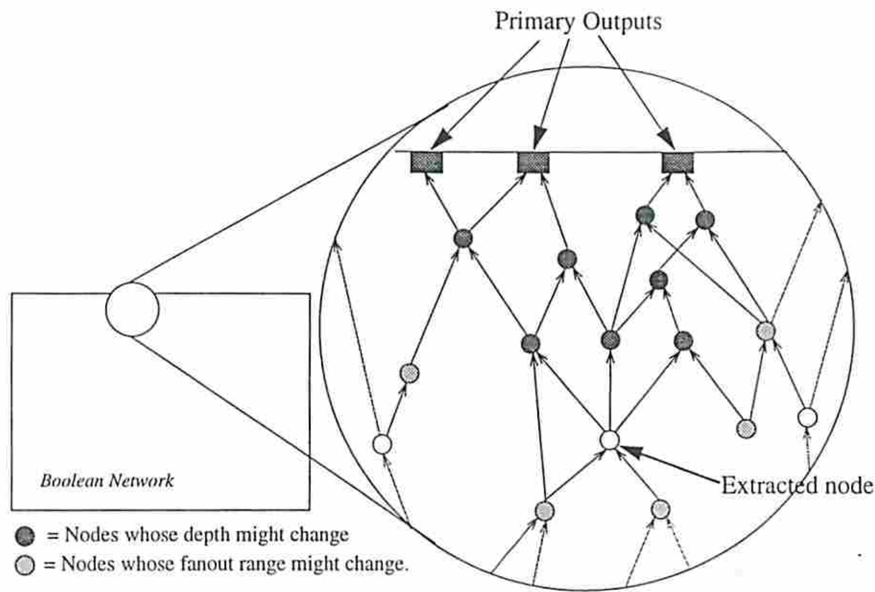


Figure 7.1: Identifying the nodes whose fanout ranges may change due to extraction.

7.2.2 Improving the Efficiency of Fanout Range Update

Before the extraction, all the depth determining edge, i.e., all edges ij such that $depth(j) = depth(i) + 1$, are identified. The concept of DD edges is shown in Figure 7.2. Each such edge indicates a depth dependency, i.e., if a DD edge connects output of node i to the input of node j, increasing the depth of i or extracting a divisor from j containing i will increase the depth of node j by one. Such an extraction is referred to as an extraction *along a DD edge*. For example, in Figure 7.2, the divisor $A + B$ is extracted along DD edges Ae, Af, Be and Bf.

Theorem 7.2.2 *Fanout range of a node in the network may change only if a candidate divisor is extracted along a depth determining edge.*

Proof If a divisor is not extracted along a DD edge, depth of none of the fanout nodes of the divisor will change. Hence, fanout range of all other nodes in the network will remain unchanged. ■

Hence, if a divisor is not extracted along a DD edge, the routing costs based on equations (6.11) and (6.12) remain unchanged except for the new fanout range of the extracted node. In this case, equation (6.11) and equation (6.12) can be computed

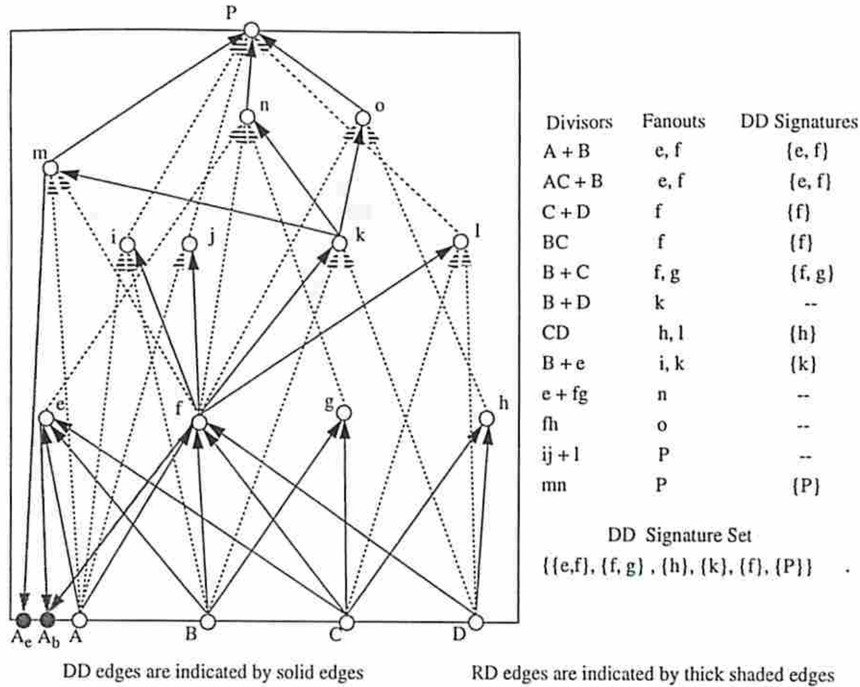


Figure 7.2: Illustration of the DD edges, DD Signature Set and RD Edges.

in $O(1)$ and $O(maxDepth)$, respectively where $maxDepth$ is depth of the resultant circuit.

Apart from allowing a more efficient network traversal for depth update, the concept of DD edge can be used to reduce the number of updates that need to be done for each set of "good" divisor as explained next. I define as *depth dependent signature* (DD signature) the set of fanouts along DD edges of the divisor. For example, in Figure 7.2 divisor CD fans out to nodes h and l. However, since depth of only h will change due to the extraction of CD, the DD signature of CD is $\{h\}$. For each such divisor, the nodes whose depths changed due to the extraction are identified by identifying all the nodes that are reachable on DD edges from the each node belonging to the DD signature of that divisor.

From the set of candidate divisors, *all* distinct DD signatures are identified as *depth dependent signature set* (DD signature set). The size of the DD signature set corresponds to the number of depth updates and routing cost calculations needed to identify the best divisor minimizing the routing cost. It should be noted that further minimization of the DD signature set can be obtained by reducing a DD signature containing all depth dependent fanouts of a node i into node i itself. For

example, in Figure 7.2 signature $\{P\}$ can be reduced to $\{m\}$ or $\{n\}$. However, in this case it does not lead to further minimization of the DD signature set. In any case, the size of DD signature set is upper bounded by the number of candidate divisors under consideration. Thus, the concept of DD signature set is used to reduce $|G|$. $|G|$ can be further reduced by restricting the number of "good" divisor by a constant. Thus, time complexity of each extraction is reduced to $constant * TimeComp(RoutingCost) = O(TimeComp(RoutingCost))$.

In spite of the exact depth update using DD signature set and DD edges, the worst case complexity for *RoutingCost* is still $O(E)$ for each element of the DD signature set. However, by constructing a *Range Determining network* (RD Network) before each extraction, the *RoutingCost* routine can be further sped up as described next.

In the RD network, for each node i, two additional nodes i_b and i_e are introduced. If the fanout range of node i is $[b_i, e_i]$, then a *Range Determining* (RD) edge is introduced from fanout j of node i to i_b if $b_i = d_j$ and/or to i_e if $e_i = d_j$. Thus, in the RD network, sources of each incoming edge to i_e (i_b) correspond to the set of nodes whose depths should change in order to change the fanout range of i to $[b_i, e_i + 1]$ ($[b_i + 1, e_i]$).

Thus, the RD network and DD edges exactly identify nodes whose range has changed due to the extraction of a candidate divisor. During the depth update, the RD edges are also traversed and pointers to each affected i_b and i_e is put in a list of nodes with the corresponding RD edge marked "visited". After the depth update, each node in this list is examined to verify if all RD edges are marked "visited". If so, the corresponding b_i or e_i is incremented by one. The RD edges corresponding to node A are shown in Figure 7.2. RD edges for other nodes are not shown in the figure to simplify the figure.

7.2.3 Effect of Low Weight Divisors and Single Divisors

During the experiments, it was observed that extraction of low literal savings divisors, though improved the literal count of the circuit, usually led to worse routing and chip area. Since the literal savings per occurrence of a single divisor is limited by the number of literals constituting the single divisor, this is also true for single

divisors. Specifically, when only two literal single divisors are used as is done in any "fx" based mechanism, each single divisor saves only one literal per occurrence except for the first and second occurrences for which the overall literal savings is -1 and 0, respectively. Such low weight divisors typically introduce large global nets, thereby increasing the connectivity (and hence routing area) at minimal gain in literal count of the circuit.

By not extracting these divisors, for larger circuits, although the literal count and the active cell area increased, the chip area decreased substantially. This is due to the fact that the existing placement/routing algorithms can handle clusters of nodes that are weakly connected much better than those that are strongly connected. For small circuits, however, if the additional literal savings due to extraction of these low weight divisors is significant, then the savings in gate area compensates for the increase in routing area. If the additional literal savings due to extraction of such low weight divisors is small compared to overall literal savings, increased routing overhead dominates the potential reduction in gate area and hence, such divisors should not be extracted.

Hence, in these experiments, single divisors and any divisor with less than two literal savings were not extracted for circuits with 2000 literals or more. These circuits are *ex1010, apex3 and apex4*. For the remaining circuits, low weight divisors and single divisors were extracted as long as they accounted for 10% or more of the total literal savings.

7.2.4 Experimental Results

Before presenting results with my extraction mechanisms, I produce in Table 7.3, the results obtained by optimizing circuits in SIS using the rugged script (which uses "fx" to perform extraction). All circuits including the recommended set of two level examples were mapped using the SIS mapper with LIB2 gate library in area mode, placed using GORDIAN [55] and routed using TimberWolf global router [63] and YACR2 detailed router [87]. Since the fanout range based cost functions are likely to be more effective when the layout of the circuit is directional, all the input pads were placed along the bottom edge of the chip and all the output pads were placed

Circuit	Routing Area	Gate Area	Chip Area	Delay
b12	160	96	256	14.41
cordic	159	108	267	17.17
cps	6447	1316	7762	88.12
duke2	1496	481	1977	38.79
ex1010	24067	2622	26689	523.40
ex4	1188	516	1704	17.09
misex2	222	125	347	13.41
misex3c	1471	508	1979	75.92
pdc	2018	643	2661	37.12
rd84	218	152	370	23.91
spla	2362	700	3062	47.07
9sym	502	234	736	26.02
alu4	1936	606	2542	54.43
apex2	874	362	1236	27.23
apex3	14019	1722	15741	123.84
apex4	22252	2525	24777	599.54
rd73	137	88	225	23.46
table3	4855	993	5848	202.90

Table 7.3: Results using *script.rugged* for comparison with fanout range based extraction.

along the top edge of the chip. These results are used as a basis for comparison with my experiments.

In all of my experiments, "fx" in the rugged script was substituted by the corresponding extraction procedure. The rest of the rugged script was kept intact.

I implemented extraction mechanisms based on the measures given in equations (6.11) and (6.12). Equation (6.11) corresponds to the sum of ranges for nodes in the network, and hence, it is referred to as *Range based extraction.* Likewise, extraction based on equation (6.12) is referred to as *overlap based extraction.*

Circuit	Routing Area	Gate Area	Chip Area	Delay
b12	0.88	0.99	0.92	0.98
cordic	0.72	0.73	0.72	0.84
cps	0.96	1.00	0.97	0.89
duke2	0.84	0.99	0.88	0.94
ex1010	0.43	1.32	0.51	1.01
ex4	1.02	0.99	1.01	1.03
misex2	0.97	1.00	0.98	1.07
misex3c	0.86	0.97	0.89	1.07
pdc	0.98	1.03	0.99	1.01
rd84	1.05	0.98	1.02	1.11
spla	1.20	1.04	1.16	1.00
9sym	0.76	0.79	0.77	1.13
alu4	0.96	0.99	0.96	1.01
apex2	0.98	1.01	0.99	1.14
apex3	0.60	1.19	0.67	1.33
apex4	0.42	1.17	0.49	0.79
rd73	0.92	1.05	0.97	0.76
table3	0.88	1.00	0.90	1.06
AVERAGE	0.86	1.01	0.88	1.01

Table 7.4: Results of extraction minimizing sum of fanout ranges (normalized by corresponding values in Table 7.3).

Range based Extraction:

Using equation (6.11), a divisor that results in minimum increase (or maximum decrease) in the sum of fanout ranges of the nodes in the network is selected. In the greedy procedure, this implies that at every step, the increase in the sum of fanout ranges is being minimized.

The results of this approach are given in Table 7.4. Each entry in the table is normalized with respect to the corresponding entry in Table 7.3. On average, an improvement of 14% in routing and 12% in chip area was obtained. The active gate

area of the circuits and the circuit delay remained the same. The improvement on the recommended set of circuits is about 10% in route area and 9% in chip area with no change in the delay and gate area.

It should be noted that circuits with literal count of 2000 or more, (*ex1010, apex1, apex2*, and *apex4*) the routing area improved by 39% with a chip area improvement of 33% in spite of an increase of 16% in active area. The delay degraded by 5% for these circuits.

Overlap Based Extraction:

This minimizes the fanout range overlap based on the routing cost function given by equation 6.12.

The results are given in Table 7.5. Each entry in the table is normalized with respect to the corresponding entry in Table 7.3. On average, an improvement of 13% in routing and 11% in chip area was obtained. The active gate area of the circuits remained the same while the circuit delay degraded by about 5%. The improvement on the recommended set of circuits is about 9% in route area and 8% in chip area for no change in the delay and gate area.

Once again, for circuits with literal count of 2000 or more, the routing area has improved 40% with chip area improvement of 33% in spite of an increase of 24% in active area. Delay increased by 18%.

Thus, both the routing measures result in similar improvements in the chip area and the routing area. However, the range based mechanism performs better in terms of circuit delay. On further analysis, it was observed that the range based mechanism also resulted in circuits with reduced network depth. This explains the improved delay with respect to the overlap based extraction. It is quite reasonable that minimizing equation (6.11) results in reduced network depth as every increase in circuit depth results in a significant increase in the global sum of fanout ranges. A mechanism minimizing this sum of fanout ranges is hence likely to generate circuits with less circuit depth. However, an overlap based mechanism attempts to minimize fanout ranges as well as distribute signals across the network. Hence, a procedure minimizing equation (6.12) is likely to improve the chip area better than a procedure minimizing equation (6.11).

Circuit	Routing Area	Gate Area	Chip Area	Delay
b12	0.93	0.99	0.95	0.98
cordic	0.72	0.73	0.72	0.84
cps	1.06	1.00	1.05	0.93
duke2	0.94	0.99	0.95	0.97
ex1010	0.43	1.32	0.52	1.01
ex4	1.00	0.99	1.00	1.02
misex2	0.97	1.00	0.98	1.07
misex3c	0.88	0.97	0.90	1.07
pdc	0.98	1.04	1.00	0.99
rd84	0.98	0.94	0.96	1.08
spla	1.18	1.04	1.14	0.99
9sym	0.79	0.79	0.79	1.07
alu4	0.95	0.99	0.96	1.01
apex2	0.96	1.01	0.97	1.13
apex3	0.63	1.19	0.69	1.32
apex4	0.42	1.17	0.49	0.79
rd73	0.91	0.98	0.94	0.91
table3	0.99	1.00	0.99	1.10
AVERAGE	0.87	1.01	0.89	1.02

Table 7.5: Results of extraction minimizing the square of the fanout range overlaps (normalized by corresponding values in Table 7.3).

As expected from the increased time complexity, the runtimes of both these routing-driven extraction were about 2-3 times slower than the "fast extract".

7.3 Fanin Range Based Extraction

A modification of structural parameter fanout ranges can be proposed to achieve delay and power optimization during technology independent logic synthesis[119, 117]. In this section, a novel approach for performance optimization during logic extraction is presented. The main idea is to use ranges of arrival times at fanins (i.e., fanin ranges instead of fanout ranges) of gates in the circuit to perform extraction such that path lengths from primary inputs to each gate and to each primary output are approximately equal. I show that this fanin range based extraction mechanism can be performed as efficiently as traditional extraction mechanisms. The experimental results on recommended benchmark circuits indicate improvements when compared with the results generated using the standard area and delay optimization scripts.

7.3.1 Prior Work in Performance Optimization

As mentioned earlier in this section, *Logic extraction* is the process of identifying subexpressions common to two or more Boolean functions, which are then extracted as intermediate nodes in a multi level circuit. The goal of extraction is to minimize the chip area by sharing logic across the network. Literal count has been traditionally used as the objective function during extraction as it correlates well with the gate area. However, no approach has been proposed to perform extraction to minimize the circuit delay. I summarize below what the two main reasons for that and describe how the proposed approach addresses them.

1. Accurate delay models for use during technology independent phase of logic synthesis are lacking.

   The algorithm proposed here does not attempt to directly minimize delay of the circuit, instead it tries to modify the global structure of resultant circuit so as to balance the path lengths in the circuit. Thus, this approach reduces the dependence on accuracy of delay estimation to achieve delay optimization.

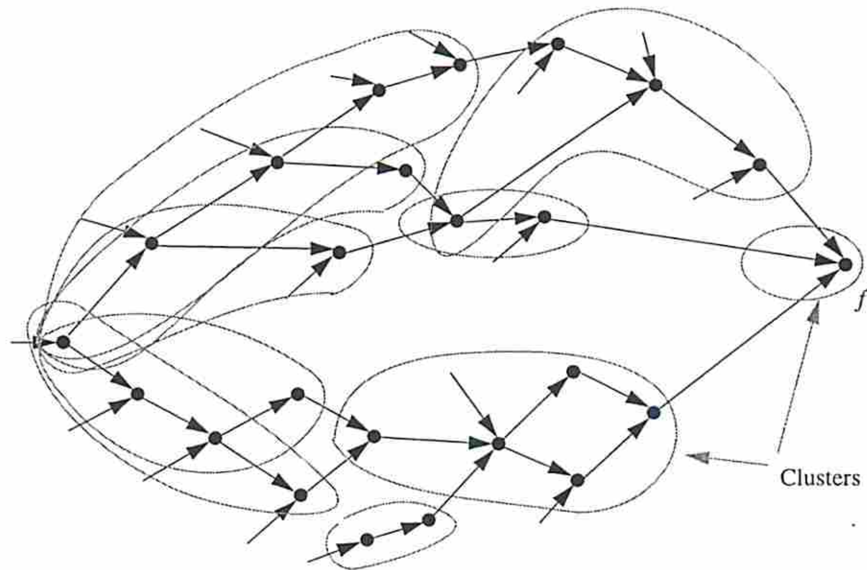
However, the use of a good delay estimation technique is not precluded as the proposed approach is general enough to accommodate any delay model.

2. Standard delay optimization scripts (e.g., *script.delay* of SIS) have shifted the onus of delay optimization from extraction procedure to other preprocessing steps (e.g., performance-driven clustering), leaving extraction with little freedom to perform further delay optimization.

   With respect to such delay optimization scripts, our approach is still effective by producing circuits with lower area and power dissipation, but the same delay. It is also shown that with a slight modification to the standard delay script that increases the flexibility of the extraction operation, my approach further improves results.

The main idea behind my approach is to synthesize circuits where path delays from primary inputs to each internal node and to each primary output are approximately equal. A direct effect of a path balanced circuit structure is that potential for further delay optimization by trading-off delay on non-critical paths is reduced as the number of such non-critical paths (and their non-criticality) are reduced; implying that the resultant circuit is already delay-optimal. Indeed, it has been shown in [125] through formal arguments that a circuit is delay-optimal when all the paths are equally critical. The simplifying assumptions in their work were:(1) area could be traded-off for delay (and vice versa) in a continuous, rather than a quantized fashion; (2) the operations to achieve these trade-offs are such that the network structure remains unchanged, e.g., gate/transistor sizing. Their proof therefore does not extend to operations in logic synthesis that change the network structure. However, the idea of balancing paths to improve delay is intuitively promising and hence should be attempted for other operations of logic synthesis. Furthermore, if path balancing is attempted in a constructive manner (e.g., during extraction) as opposed to an incremental manner (e.g., post-synthesis buffer insertion), then circuits with lower delay can be obtained at little or no area penalty.

It is interesting to note that almost all of the delay optimization procedures proposed for logic synthesis indirectly lead to a path balanced circuit. One of the earliest works on delay optimization was proposed by Lawler et al. [62] who proposed a clustering algorithm, which when given a size constraint on each cluster, produces



Before Clustering: Earliest arrival time at $f = 2$; Latest arrival time at $f = 9$
After Clustering: Earliest arrival time at $f = 2$; Latest arrival time at $f = 3$

Figure 7.3: Illustration of Lawler's algorithm for optimal depth clustering with size constraint 5.

delay-optimal clustering by using a node labeling scheme. As can be seen in the Figure 7.3 (which is reproduced from [62]), Lawler's algorithm tends to balance paths in terms of the clustered node depth.

A circuit speed-up procedure was proposed by Wang [122]. In his approach, circuit speed-up was obtained by selective collapsing and delay-optimal re-decomposition of nodes on critical paths. This approach essentially applied Huffman's algorithm to decompose nodes to two input NAND gate trees. As can be seen in Figure 7.4, once again, this indirectly tends to balance paths from the leaf nodes of the decomposed tree to the root of the decomposed tree.

Likewise, it can be illustrated that fanout optimization procedures [104, 111], performance driven technology mapping procedures [104, 13] and other delay optimization procedures [73, 99] also indirectly tend to balance paths.

This section presents a mechanism to achieve path balancing during extraction. This is achieved by minimizing *fanin ranges* of the nodes in the circuit. Fanin range of a node is defined as the difference between the earliest arrival time and the latest arrival time at the inputs of the node.

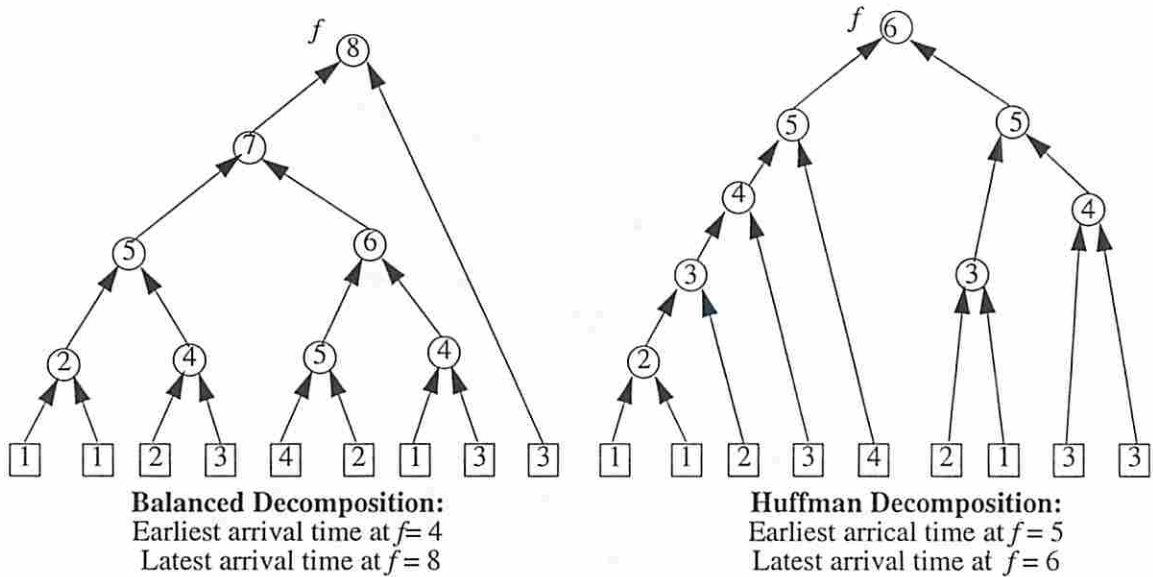


Figure 7.4: 2-input AND decomposition: Balanced Vs. Huffman.

The rest of the section is organized as follows. In Section 7.3.2, I present the formal definition of fanin range and present the theory behind fanin range based extraction. Section 7.3.3 presents the experimental results and discussion.

7.3.2 Performance Optimization during Extraction

Ideally, to obtain a delay optimal circuit, maximum path delay should be optimized directly during extraction. However, optimizing the delay directly is not a practical approach as explained below.

First of all, in order to minimize the maximum path delay during extraction, a good delay estimation tool for technology independent phase should be available. Otherwise, the quality of delay optimization will be limited by the accuracy of the delay optimization tool. Also, as described in at the beginning of this chapter, the overall extraction procedure is a greedy mechanism. Optimizing the circuit delay at each iteration of this greedy mechanism does not guarantee delay optimality of the resultant circuit. Finally, such an approach would be very time consuming as it will require a delay update of the entire network for each candidate divisor during each iteration of the greedy procedure.

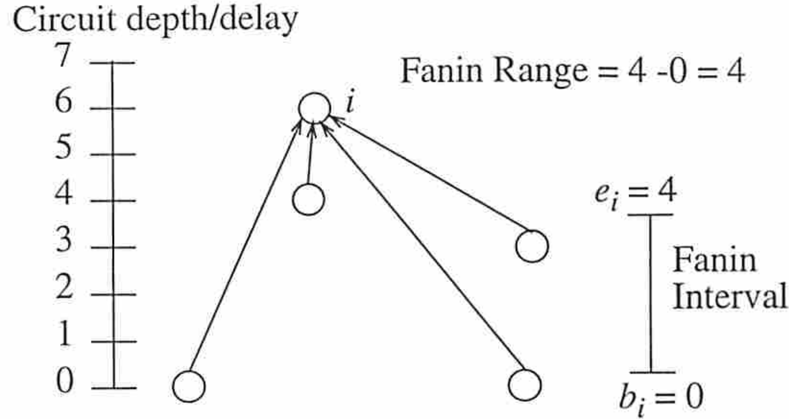


Figure 7.5: Illustration of fanin ranges and fanin intervals.

Hence, I attempt to achieve performance optimization indirectly by minimizing the fanin ranges of the nodes extracted. I believe that by abstracting out the details about delay estimation and focusing on network structure, not only is the delay optimization more robust (i.e., persists through the following technology dependent phase), but also remains valid irrespective of the delay model.

Before describing my approach for performance-driven extraction based on fanin ranges, I give the definition of fanin range. Let I_i denote the set of immediate fanins of a node i and d_i denote the depth of node i ($d_i = 0$ if i is a primary input).

Definition 7.3.1 *The* **fanin interval** *of a node i is defined as $[b_i, e_i]$, where $b_i = min_{j \in I_i}\{d_j\}$ and $e_i = max_{j \in I_i}\{d_j\}$ denote the beginning and end of the fanin interval, respectively. The* **fanin range** *R_i of node i is then calculated as $R_i = e_i - b_i$.*

These definitions are illustrated in Figure 7.5.

The Objective Function:

In order to achieve path balancing at a primary output, all paths from primary inputs to that primary output need to be as balanced as possible. Unfortunately, enumeration of all such paths and application of transformations to achieve path balancing is very inefficient and difficult. In the proposed approach, global path balancing is achieved using only local information, i.e., fanin range of a each node. The following theorem shows a relationship between minimization of fanin ranges and

delay optimality. It is assumed for the sake of following theorem that delay/depth[3] of a non-critical path can be unconditionally traded-off for a delay improvement on critical paths. There are two mechanisms to achieve this: (1) the circuit can be restructured such that some logic on critical path is transfered to non-critical paths; (2) area trade-offs can be made between the critical and non-critical path without any circuit restructuring, e.g., by performing gate/transistor sizing. However, in this section, since the extraction operation is being targeted, it is assumed that each node is delay-optimal with respect to operations like gate/transistor sizing, thereby, focusing my attention to structural modifications. However, instead of achieving this in an incremental manner, e.g., by transferring logic from critical paths to non-critical paths, I attempt to achieve path balancing in a constructive manner during logic extraction.

Theorem 7.3.1 *There exists a depth-optimal network where fanin range of each node is minimum.*

Proof Assume that there is a node i which does not have minimum fanin range. Partition the set of inputs of node i in two subsets: $\bar{E}_i = \{j | j \in I_i, d_j \neq e_i\}$ and $E_i = \{j | j \in I_i, d_j = e_i\}$. By assumption, value e_i can be reduced by transferring logic from the transitive fanin cone of nodes in E_i to the transitive fanin cones of nodes in $\bar{E}_i$. It should be noted that this process also minimizes the fanin range of node i. This process can be repeated until it is not possible to reduce e_i anymore, i.e., until the fanin range of node i is minimized. This, however, can not lead to an increase in the depth of node i (this is also true for any delay function that satisfies the monotone speed-up property). Therefore, it is always possible to transform a network that does not have minimum fanin range at each node to one that has minimum fanin range at each node and has equal (or less) depth. ∎

The assumption that the circuits can be restructured unconditionally holds for tree decompositions of simple functions (e.g., AND/OR function) where any set of

[3]In this work, the unit delay model is used. Hence, from now on, the term "depth" of a node is used in place of "delay" at the node. However, the theorems presented here are applicable to any load independent delay model. This approach can also be modified to accommodate other delay models that satisfy the monotone speed-up property [67].

inputs can be extracted without any restriction and without affecting decomposability of other inputs. However, the assumption does not remain valid for general decomposition or for extraction. Still, Theorem 7.3.1 suggests the following objective function for performance optimization .

$$P_N = \sum_{i \in N} R_i \tag{7.3}$$

P_N is a monotone cost function, i.e., $\forall N' \subseteq N, P_{N'} \leq P_N$. In general, it is not necessary to minimize fanin ranges of nodes that belong only in the transitive fanin cones of non-critical primary outputs. When $P_N = 0$, all input signals at each node arrive at exactly the same time, thus, resulting in a perfectly balanced and hence, depth-optimal circuit. However, other circuits with $P_N > 0$ may also be depth-optimal.

Performance-Driven Extraction:

An extraction procedure (referred to as DELAY_EXTRACT) based on equation (7.3) was implemented based on algorithm EXTRACT. The greedy scheme of algorithm EXTRACT was used with all divisors restricted to be either double divisors, or single divisors with two literals as in [86]. However, instead of selecting the divisor based on the best literal savings, in algorithm DELAY_EXTRACT the divisor with minimum delay cost calculated based on equation (7.3) was selected. A routine *DelayCost* was implemented to calculate this delay cost.

Since the time complexity of each extraction in this approach is $TimeComp(DelayCost)$ as compared to $O(1)$ for "fast extract", the overall time complexity of *Delay_Extract* is given by $TimeComp(Extract) * TimeComp(DelayCost)$

Delay Cost of a Divisor:

Calculating the delay cost based on equations (7.3) requires a recalculation of depths (arrival times) of the nodes in the network for each candidate divisor, requiring $O(E)$ in the worst case for each divisor. This is because subsequent to an extraction, depths

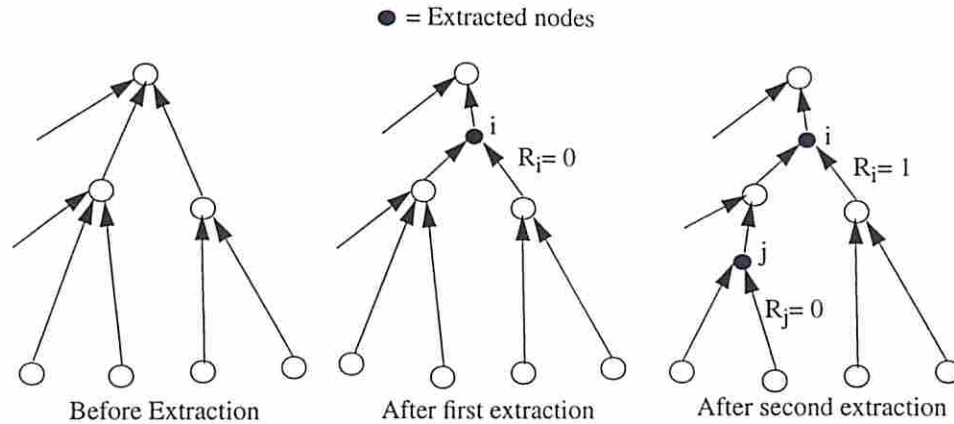


Figure 7.6: Effect of a subsequent extraction on fanin range of a node.

and fanin ranges of some nodes in the transitive fanout cone of the extracted node may change. This is illustrated in Figure 7.6.

On further analysis of the extraction procedure, I realized that if the original circuit is a two-level circuit, then a modified form of equation (7.3) can be minimized simply by minimizing the fanin range of the divisor being extracted. This implies that the delay cost of each divisor depends only on local information, thereby *DelayCost* can be performed in $O(1)$. This is described in detail next.

Two-Level Circuits:

Theorem 7.3.2 *If the original circuit is a two-level circuit, then the fanin range of an extracted node remains unchanged during subsequent kernel extractions.*

Proof Fact 7.0.1 implies that no further extraction from a newly extracted node is possible. When starting from a two-level circuit, this implies that depth of an extracted node and all nodes in its transitive fanin will remain unchanged through subsequent extractions. ∎

Corollary 7.3.3 *For two-level circuits, selecting a divisor with minimum fanin range in algorithm* DELAY_EXTRACT *minimizes the following objective function.*

$$P_{N-PO} = \sum_{i \in N-PO} R_i \tag{7.4}$$

Proof Since the original circuit was two-level, the set of internal nodes (i.e., $N - PO$) corresponds to all the nodes extracted by algorithm DELAY_EXTRACT. Since algorithm DELAY_EXTRACT introduces these divisor such that there is a minimal increase in P_n, and since, as stated in Theorem 7.3.2, once extracted, the fanin range of an extracted node does not change through subsequent extractions, algorithm DELAY_EXTRACT minimizes equation (7.4). ∎

As can be observed from equation (7.4), fanin ranges of the primary outputs are ignored. In fact, improvement in the runtime is achieved as a direct result of ignoring the primary outputs. Furthermore, it should be noted that although this method can be applied to other delay models, the accuracy of delay calculation is still limited by the "unknown load problem" of load dependent delay models.

Multi-Level Circuits:

For a circuit that is originally multi-level, the fanin ranges of extracted nodes may change due to subsequent extractions as shown in Figure 7.6. One can deal with the problem using either of the following two approaches:

1. Divisors with minimum fanin ranges in algorithm DELAY_EXTRACT can be extracted ignoring the implications of changes in fanin ranges of already existing nodes. The correct fanin ranges may be recalculated after every k extractions where k is a user specified parameter.

2. Algorithm DELAY_EXTRACT can be modified to guarantee that all divisors extractable from a node i are extracted before a divisor containing node i is extracted. This essentially levelizes the extraction procedure. In this case, the extraction mechanism might select a divisor with non-minimal fanin range to confirm to the level constraint.

In the present implementation, the first approach is adopted.

7.3.3 Experimental Results and Discussion

A *Fanin Range based eXtraction* (FRX) procedure was implemented within the SIS synthesis environment. During my experiments with FRX, I realized that it is not

necessary to apply a purely performance-driven extraction algorithm to produce a circuit with minimum post-route delay. In fact, a purely performance driven extraction mechanism might produce a depth optimized circuit at the cost of increased literal count and hence an increased gate area. Increased gate area might lead to increased chip area, subsequently degrading the circuit performance due to increased interconnect delay. Hence, it was decided that it would be better to merge algorithms EXTRACT and DELAY_EXTRACT to produce a hybrid algorithm similar to ROUTINGDRIVEN_EXTRACT that allows a trade-off between area and delay costs.

```
Algorithm AREADELAY_EXTRACT (N, P)
N is a given Boolean network
P is % value to identify good divisors
01 begin
02   Generate candidate divisors G
03   do
04     BestAreaCost = 0
05     BestFaninRange = ∞
06     BestDivisor = NULL
07     ForEachDivisor D ∈ G
08       if AreaCost(D) ≥ 1 − P/100*BestAreaCost
09         if AreaCost(D) ≥ BestAreaCost
10           BestAreaCost = AreaCost(D)
11         if FaninRange(D) ≤ BestFaninRange
12           BestDivisor = D
13           BestFaninRange = FaninRange(D)
14     Divide the Boolean equations by BestDivisor
15     Update the cost of remaining candidate divisors
16   While a candidate of sufficient merit exists
17 end
```

In this hybrid extraction algorithm, instead of choosing the "best" divisor with minimum fanin range, a divisor with minimum fanin range from a set of "good"

divisors is selected. These "good" divisors are identified from the list of candidate divisors based on their literal savings potential to ensure that the improvement in performance is not achieved at the cost of a substantial increase in the active area. The basic strategy to select "good" divisors from the list of candidate divisors is as follows: if maximum literal savings from any divisor is M, select all divisors with literal savings within $\mathcal{P}\%$ of M. As in algorithm ROUTINGDRIVEN_EXTRACT, with this percentage based selection, the emphasis can be shifted from area/literal-driven extraction to performance-driven extraction by increasing the percentage value. The experiments were conducted while varying the percentage value from 100% (i.e., completely performance-driven) to 0% (i.e., completely area-driven as in "fast-extract"). In all the results reported here, top 25% of literal savings were identified as "good" divisors, i.e., $\mathcal{P} = 25$.

Before presenting results with FRX, the results obtained by optimizing circuits in SIS using the "script.rugged" (which uses "fx" to perform extraction) are produced in Table 7.6. Table 7.7 presents the results obtained by optimizing circuits using the "script.delay" (which uses "fx -l" to perform extraction). To generate the results, the circuits were mapped using the SIS mapper with LIB2 gate library, placed using GORDIAN [55] and routed using TimberWolf global router [63] and YACR2 detailed router [87].

In my experiments with FRX, traditional extraction operations in *script.rugged* and *script.delay*, namely, "fx" and "fx -l", respectively, were replaced by FRX. These modified scripts are referred to as "script.rugged.frx" and "script.delay.frx", respectively. Other operations in *script.rugged* and *script.delay* were kept intact.

I report *post-route* area, delay and power dissipation. I would like to mention that the only reason for reporting post-route costs is to confirm the improvements even after layout phase. The results follow the same trend after technology mapping. For all circuits, power dissipation was calculated under a real delay model, using the tagged probabilistic simulation approach [106], except for circuit *C6288*. For this circuit, the power dissipation could not be computed due to memory limitations.

Results presented in Tables 7.6 and 7.7 are used as a basis for comparison with my experiments. The first set of results in each table corresponds to the **recommended** two-level benchmark circuits, whereas the second set of results correspond to the recommended multi-level benchmark circuits.

Circuit	Chip Area	Delay	Power
Two-level Benchmarks			
b12	186798	13.12	1423.00
cordic	149974	13.29	1249.94
cps	8823648	72.91	21166.17
duke2	1670544	34.28	6341.49
ex1010	2008192	166.19	29899.54
ex4	1467390	17.03	10170.23
misex2	280924	12.66	1511.06
misex3c	1711358	66.32	11986.85
pdc	1412376	24.33	6720.21
rd84	341820	18.36	2405.23
spla	2913820	36.47	11068.50
Multi-level Benchmarks			
C1355	1409814	32.65	9318.21
C1908	1695806	49.35	10756.75
C2670	3035436	51.51	21706.63
C3540	6125550	79.05	43418.80
C432	589778	53.21	4212.97
C6288	16011560	176.89	257292.96
C7552	12553632	141.93	89485.52
b9	325428	15.11	2195.94
dalu	4002290	84.40	16488.66
des	22539834	150.46	53111.01
k2	6688776	56.83	14240.58
rot	2645460	34.98	14738.80
t481	1751310	33.29	6138.30

Table 7.6: Results using *script.rugged* for comparison with fanin range based extraction.

Circuit	Chip Area	Delay	Power
b12	228956	5.82	1756.55
cordic	255060	9.40	2235.95
cps	17872024	48.33	59544.13
duke2	3522250	14.23	15615.45
ex1010	3582312	15.35	45307.00
ex4	1929068	11.20	12093.50
misex2	447252	7.21	2593.15
misex3c	4246122	16.64	18460.27
pdc	2169860	15.23	11111.17
rd84	503378	11.27	3619.76
spla	7581942	19.73	27304.08
C1355	3015826	26.27	22120.63
C1908	4473872	30.66	27264.74
C2670	4922758	30.47	34913.31
C3540	9158650	42.25	58018.53
C432	828162	26.40	5867.40
C6288	18881114	101.55	304130.06
C7552	28519706	42.86	166241
b9	467860	7.37	2804.09
dalu	7452282	25.71	35991.78
des	54439343	16.69	88145.96
k2	17072374	28.88	26896.99
rot	4345128	18.97	22325.09
t481	7590596	23.85	19492.72

Table 7.7: Results using *script.delay* for comparison with fanin range based extraction.

Circuit	Chip Area	Delay	Power
b12	1.05	0.98	0.98
cordic	1.04	0.97	1.01
cps	0.99	0.82	1.04
duke2	0.97	0.95	0.96
ex1010	1.00	0.98	1.00
ex4	0.99	1.00	1.00
misex2	1.08	1.02	1.01
misex3c	1.00	0.82	0.93
pdc	1.10	1.08	0.99
rd84	0.77	0.90	0.71
spla	0.95	1.05	0.98
C1355	0.99	1.00	1.00
C1908	1.08	0.99	1.06
C2670	0.94	0.96	0.98
C3540	0.99	0.90	0.96
C432	0.97	1.00	1.00
C6288	1.00	1.00	1.00
C7552	0.92	0.97	0.97
b9	1.06	0.86	0.98
dalu	0.83	0.98	0.93
des	1.00	1.00	0.98
k2	1.03	0.75	0.98
rot	0.96	0.95	0.98
t481	0.91	0.90	1.03
AVERAGE	0.98	0.95	0.98

Table 7.8: Results using *script.rugged.frx* (normalized by corresponding values in Table 7.6).

FRX in the context of script.rugged:

As can be seen in Table 7.8, using FRX in place of "fx" has improved the circuit delay by as much as 25%. On average, an improvement of 5% in circuit delay and about 2% in circuit power dissipation and circuit area was obtained.

FRX in the context of script.delay:

I also conducted some experiments to check the effectiveness of my extraction algorithm in the context of *script.delay*. For sake of reference, the *script.delay* is reproduced here.

```
sweep
decomp -q
tech_decomp -o 2
resub -a -d
sweep
reduce_depth -b -r
red_removal
eliminate -l 100 -1
simplify -l
full_simplify -l
sweep
decomp -q
fx -l
tech_decomp -o 2
rlib lib2.genlib
rlib -a lib2_latch.genlib
map -s -n 1 -AFG -p
```

The results are presented in Table 7.9. As can be seen, using FRX in place of "fx -l" has improved the chip area and power dissipation by about 6% each, but the circuit delay has degraded by about 3%. However, this is due to the following reasons.

1. Path balancing has already been achieved using another depth optimizing operation, namely, "reduce_depth" (which is based on Lawler's clustering algorithm). Unlike delay optimization procedures that modify network structure locally, e.g., technology decomposition, technology mapping etc., FRX and *reduce_depth* are both global operations modifying the underlying structure of the network. As mentioned in Section 7.3.1, both these operations are targeting the same goal. Thus, an optimization by one is likely to reduce optimization potential for the other.

2. As can be seen from *script.delay*, significant delay optimization has already been performed before the extraction phase. The very fact that a quick decomposition (i.e., "decomp -q" which decomposes each node into smaller node reducing the extraction potential of "fx -l" and FRX) is performed just before extraction implies that this script does not expect much performance optimization from the extraction operation as long as it does not increase the depth of the resultant circuit ("-l" option in "fx -l" ensures that the circuit has same depth after extraction). Thus, at this stage, the circuit is already delay optimized and the potential and flexibility available to extraction operation to improve the delay of the circuit are severely limited. As the experimental results verify, for the same circuit, the number of divisors extracted and their literal savings in *script.delay* are much less than that in *script.rugged*.

To increase the potential of extraction operation, a partial collapse of the circuit was performed just before extraction. Specifically, all nodes from the circuit whose collapse increase the literal count of the circuit by at most 100 (i.e., by performing an "eliminate 100") were collapsed. The results are presented in Table 7.10.

As can be observed, the delay could not be improved as a result of this experiment. This implies that *reduce_depth* has indeed generated robust circuits with respect to delay. However, the power dissipation has improved by about 17% and area has improved by about 11%. Also, it was observed that the power improvement did not come only from minimization of hazard activity; power dissipation under zero delay model was also minimized.

This improvement in area and power dissipation can be explained as follows. Delay optimization in *reduce_depth* is obtained by duplicating the logic, which might

Circuit	Chip Area	Delay	Power
b12	1.03	1.00	1.00
cordic	0.97	1.00	1.00
cps	0.95	0.95	0.95
duke2	1.05	1.06	0.99
ex1010	0.98	1.02	1.01
ex4	0.99	1.00	1.00
misex2	0.98	1.06	0.93
misex3c	0.99	1.03	1.02
pdc	1.01	0.99	1.01
rd84	1.00	1.03	0.95
spla	0.89	0.97	0.92
C1355	0.72	0.96	0.72
C1908	0.76	1.15	0.74
C2670	0.79	0.98	0.73
C3540	0.98	1.01	0.95
C432	0.95	1.02	0.97
C6288	1.08	1.10	—
C7552	0.94	0.97	0.96
b9	0.91	1.05	0.93
dalu	1.03	1.14	1.03
des	0.98	1.05	0.97
k2	0.93	1.00	0.84
rot	0.97	1.02	0.99
t481	0.96	1.06	0.96
AVERAGE	0.95	1.03	0.94

Table 7.9: Results using *script.delay.frx* (normalized by corresponding values in Table 7.7).

Circuit	Chip Area	Delay	Power
b12	1.21	1.10	0.98
cordic	1.23	1.26	1.00
cps	0.73	0.64	0.55
duke2	0.80	0.97	0.70
ex1010	0.92	1.05	0.58
ex4	1.12	0.94	1.13
misex2	0.90	1.05	0.77
misex3c	0.98	1.11	0.98
pdc	0.95	1.04	0.87
rd84	0.98	1.06	0.99
spla	0.76	0.91	0.66
C1355	0.67	0.92	0.65
C1908	0.68	1.07	0.69
C2670	0.78	0.95	0.71
C3540	1.20	1.10	0.97
C432	1.03	1.10	1.11
C6288	0.93	1.09	—
C7552	0.97	1.02	0.90
b9	0.93	1.06	0.92
dalu	0.86	1.10	0.76
des	0.80	1.06	0.69
k2	0.75	0.99	0.98
rot	0.93	1.10	0.89
t481	0.39	0.86	0.53
AVERAGE	0.89	1.02	0.83

Table 7.10: Results using *script.delay.frx* with "eliminate 100" (normalized by corresponding values in Table 7.7).

increase the gate area and and power dissipation. By performing a partial collapse, extraction mechanism was given greater flexibility to identify and reduce such duplicated logic, thereby resulting in area and power savings. Use of FRX to perform extraction ensured that this is achieved at no cost in delay. This also shows that if both *extraction* and *reduce_depth* can independently produce circuits with approximately same delay characteristics, it is better to use *extraction*, because by its very nature, *reduce_depth* introduces logic duplication, whereas *extraction* reduces duplication.

I believe that extraction mechanism is an ideal candidate for performing path balancing in a constructive manner. Extraction has significant impact on the global structure of the circuit as it identifies common-subexpressions, i.e., it introduces nodes with multiple fanouts. In fact, for a circuit that was originally a two-level circuit, extraction identifies and introduces all internal nodes (including all multiple fanout nodes). Even in a multi-level circuit, extraction operation introduces most of the internal nodes with multiple fanouts. Since most of the operations performed after extraction are local operations working within the boundaries defined by multi-fanout nodes (e.g., the existing technology mapping algorithms rarely produce a circuit in which a multiple fanout node is hidden within a gate), most of the nodes introduced by extraction remain in the final circuit. Indeed, experiments have verified that almost all multiple fanout nodes generated by extraction are present in the final circuit. Thus, being the last operation working on the global structure of the network, extraction is mostly responsible for the structure and the behavior of the final circuit. Furthermore, since extraction constructs the final circuit incrementally, i.e., one divisor at a time, it provides a finer control on the growth of circuit structure, thus allowing path balancing in a constructive manner.

In the results, it was also noticed that circuits optimized using FRX have less power dissipation under the real delay model. This is due to minimization of the fanin range of the node, which causes the signals to arrive at a node within a smaller time interval, thereby reducing the hazard activity of the circuit.

Chapter 8

Conclusion: Layout-driven Logic Synthesis

The CAD industry/research is on the verge of a total transformation with the anticipated move towards deep-submicron technology. Existing synthesis tools fail to meet the challenge in the following respects: 1) they ignore the interconnect contribution to circuit area, delay and power dissipation; 2) they cannot handle circuits of even moderate size and are clearly incapable of handling multi-million transistor deep-submicron circuits; 3) they use simple circuit modeling (specially for delay and power dissipation) which becomes increasingly inaccurate for deep-submicron devices. The main thesis of this dissertation is to propose synthesis mechanism to address these challenges posed by deep-submicron technology, specifically, the ever-increasing interconnect contribution to all circuit costs. Even in the submicron circuits, routing affects the circuit significantly. Currently, routing accounts for about 60-80% of total chip area, 40-60% of the circuit delay and a significant amount of total power dissipation. A combined effect of ever-increasing interconnect contribution and synthesis tools that ignore the interconnect contribution is that the area, delay and/or power dissipation constraint violations are increased substantially after the interconnect contribution is taken into account. This results in a dramatic increase in the number of synthesis-layout iterations to meet the area, delay or power dissipation constraints, increasing the design time significantly. These interconnect dominated, multi-million transistor deep-submicron circuits pose a major challenge to the CAD community, the only solution to which is stronger links between layout and synthesis tools. The work presented in this dissertation is a step in that direction by providing mechanisms to optimize post-layout area during logic synthesis, namely, layout-driven logic synthesis.

8.1 Main Results and Contributions

Two basic approaches for layout-driven logic synthesis were presented and analyzed in this dissertation, namely, placement based approaches and structure based approaches. The distinction between these two approaches is based on the underlying mechanism used for capturing the post-layout costs.

8.1.1 Placement Based Approaches

Departing from the traditional norm for the placement based approach, I presented an abstract mechanism for layout-driven logic synthesis based on alphabetic trees that is less dependent on the exact locations and hence likely to be more consistent in producing good quality circuits. Some theory on alphabetic trees was also developed in the course of this work [112]. The generic theory on alphabetic tree optimization was then applied to the layout-driven versions of technology decomposition and fanout optimization operations during logic synthesis. Specifically, binary alphabetic tree optimization procedure was applied to technology decomposition operation to generate delay-optimal alphabetic technology decomposition trees [84] whereas nonbinary alphabetic tree optimization procedure was applied to generate delay-optimal alphabetic fanout trees [111, 109]. In both applications, it was established that routing contribution of the final circuit can be reduced effectively during technology dependent phase of logic synthesis.

8.1.2 Structure Based Approaches

Apart from proposing a placement based approach based on alphabetic trees, this dissertation also presented a new approach for layout-driven logic synthesis based on the structural properties of the network. Since the interconnect contribution of a network is completely determined by the structure of the underlying graph, use of structural parameters to capture post-layout interconnect cost is quite intuitive. A number of structural parameters that correlate well with the actual routing contribution were proposed and analyzed. Based on these parameters, some useful measures to capture post-layout area were presented. These measures are more abstract than the placement based measures and hence are applicable to earlier stages

of logic synthesis. Specifically, a measure of relative interconnect costs of nets based on the number of pins on the nets was proposed and optimized [116, 120]. Furthermore, a novel circuit parameter based on the depth/delay ranges at the fanin or fanout of a node in the network was proposed in this dissertation. It was shown that effective measures of post-layout routing cost can be derived based on the fanout ranges of the nodes that, when minimized, leads to improved chip area [118]. The concept of fanin ranges was used to minimize the circuit delay [119, 117]. It was also shown that when delay optimization is obtained from a operation that reduces duplication/redundancy (e.g. extraction) instead of an operation that introduces redundancy (e.g., clustering), the improvements in area and power dissipation can be substantial.

8.2 Discussion and Future Directions

Apart from the contributions mentioned above, the work done during the course of this work also leads to a better understanding of the problem of interaction between layout and synthesis tools. Next, I summarize the lessons learned from my experiments with layout-driven logic synthesis.

8.2.1 Placement Based Approaches

The main impediments to the success of a placement based approach are: (1) the behavior of state-of-the-art layout tools is difficult to predict; and (2) the quality of layouts produced by the existing layout tools is very high reflecting the maturity of research in layout tools. Fortunately, the problem of unpredictability of placement tools can be circumvented by forcing the final companion placement solutions as final placement solutions. However, this has to be achieved at no loss in quality of the resultant layout. Indeed, in cases where forcing a companion placement solution still generates a good quality layout (e.g., for early floor-planning), placement based approaches have been quite successful. My experience indicates that reliable companion placement solutions are achieved when the total circuit size is small compared to the granularity (i.e., typical size of individual element) of the circuit; a characteristic satisfied by circuits during early floor-planning. During logic

synthesis, compared to the size of each element, the circuit size is quite large leading to unpredictability of layout tools. This requires that to maintain control and to produce consistent results, the companion placements should be forced as final placements without going through a stand-alone placement tool. Thus, the success of placement based approaches for layout-driven logic synthesis depends on the capability of incremental placement mechanism to produce circuits of quality similar to or better than stand-alone layout tools. However, due to a large number of movable objects, mathematical programming based or simulated annealing based placement tools typically do a better job than some ad-hoc incremental placement mechanism applied during a placement based logic synthesis methodology.

In the absence of a good incremental placement mechanism, as mentioned in Chapter 5, the placement based approach is more effective during the technology dependent logic synthesis as the network structure is finalized during technology dependent logic synthesis resulting in a companion placement that reflects the final placement more accurately.

Even when a good incremental placement mechanism is available, it is desirable to apply the placement based approach as early in the synthesis as possible based on the observation made in Chapter 5 regarding high X-conformity of the nodes earlier in the synthesis operations. The high conformity during earlier phases of logic synthesis can be explained by the small number of large gates existing in the circuit at that earlier phases of logic synthesis. This scenario is similar to the early floor-planning scenario which, as mentioned above, is ideal for a placement based approach[1]. This implies that for a placement based approach to be successful, it should be started as early as possible to capture the global connectivity of the circuit and to avoid reliance on intermediate circuit structures. Furthermore, it is quite important that assumptions made about placement during a placement based version of one synthesis operation remain relevant during subsequent synthesis operations to guarantee that improvements obtained due to the placement based operation do not disappear during subsequent operations. This implies that all subsequent operations should target the same or a compatible objective function and should be based on

[1]Unfortunately, unlike floor-planning where the circuit structure does not change at the same hierarchy, during logic synthesis the structure changes substantially, requiring an incremental placement mechanism.

the same or compatible set of placement assumptions. Ideally, the overall flow in a placement based logic synthesis system should not only maintain the relevance of previous decisions, but should also reinforce them through subsequent synthesis operations.

8.2.2 Structure Based Approaches

The structure based approach is very promising for layout-driven logic synthesis based on the intuition that the interconnection structure of the underlying network eventually determines routing area of the circuit. However, predicting the routing contribution is a very difficult task as it depends on many structural parameters and interactions between those parameters. This can be best exemplified by the outcome of an experiment conducted as follows.

For this experiment, for twenty benchmark circuits, two different implementations of the same circuit were identified such that the gate area was about the same but the routing area was different by 20% or more. I refer to the implementation with less routing area as the set of "good" implementations and the implementations with more routing area as "poor" implementations. The value of a number of routing measures, including those proposed in Chapter 6, was calculated for these circuits in an attempt to identify a common routing measure which identified the good implementation for each circuit. The experimental results showed that though many routing measures appear to be quite successful in identifying the good implementation, none of them were convincingly consistent. Also for each circuit, at least one of the measures was successful in identifying the good implementation. This implies that good implementations of these circuits need not have a common factor contributing to their goodness. This makes it difficult to identify structural parameters/measures using such a reverse engineering approach.

The above experiment implies that given a good and a poor implementation of a circuit, a routing measure may not be able to detect the good implementation in some cases as the quality of these implementations may be determined by parameters that are not included in the formulation of the given routing measure. However, by minimizing a given routing measure, the routing area of the circuit can still be improved when compared with a circuit generated without minimizing that routing

cost measure. This was the approach followed in this dissertation as described in Chapter 6 and Chapter 7.

Apart from the application of structure based approach for minimizing the interconnect contribution of the circuit, it can also be applied to minimize other costs associated with the circuit. This synthesis approach, which I refer to as *structural synthesis*, is indeed very promising outside the context of layout-driven synthesis as illustrated by application of fanin ranges to minimize delay during logic extraction. In the context of deep-submicron circuits, the circuit structure has a greater impact on the circuit costs (i.e., area, delay and power dissipation) as compared to traditional objective functions which optimize only the "gate contribution" to these circuit costs. Furthermore, calculation of a structure based measure typically takes time that is linear in the size of the circuit. Hence, it is likely that such structure based measures can be minimized using faster algorithms, making them ideal candidates for circuits with high gate count. Finally, as the experiments indicate, by appropriately abstracting out intricate details while still capturing the essence of underlying circuit structure, circuits can be optimized independent of low-level circuit details. Thus, I believe that, the "structural synthesis" approach is capable of answering challenges posed by deep-submicron technology.

8.2.3 Post-Layout Logic Resynthesis

During the course of my experiments with layout-driven logic synthesis, I realized that significant potential for improvement exists in post-layout logic resynthesis. In placement based layout-driven logic synthesis, physical design information was incorporated into the synthesis phase by carrying a companion placement during logic synthesis. In contrast, in post-layout logic synthesis, the synthesis operations are performed during/after physical design phase. Post-layout synthesis has two distinct advantages over layout-driven logic synthesis: (1) the layout information is more accurate, and (2) much larger circuits can be handled efficiently by optimizing only small regions of the placed/routed circuit. Both these advantages make post-layout optimization very attractive for emerging deep-submicron technologies where the interconnect contribution to area, delay and power will be significantly higher and the

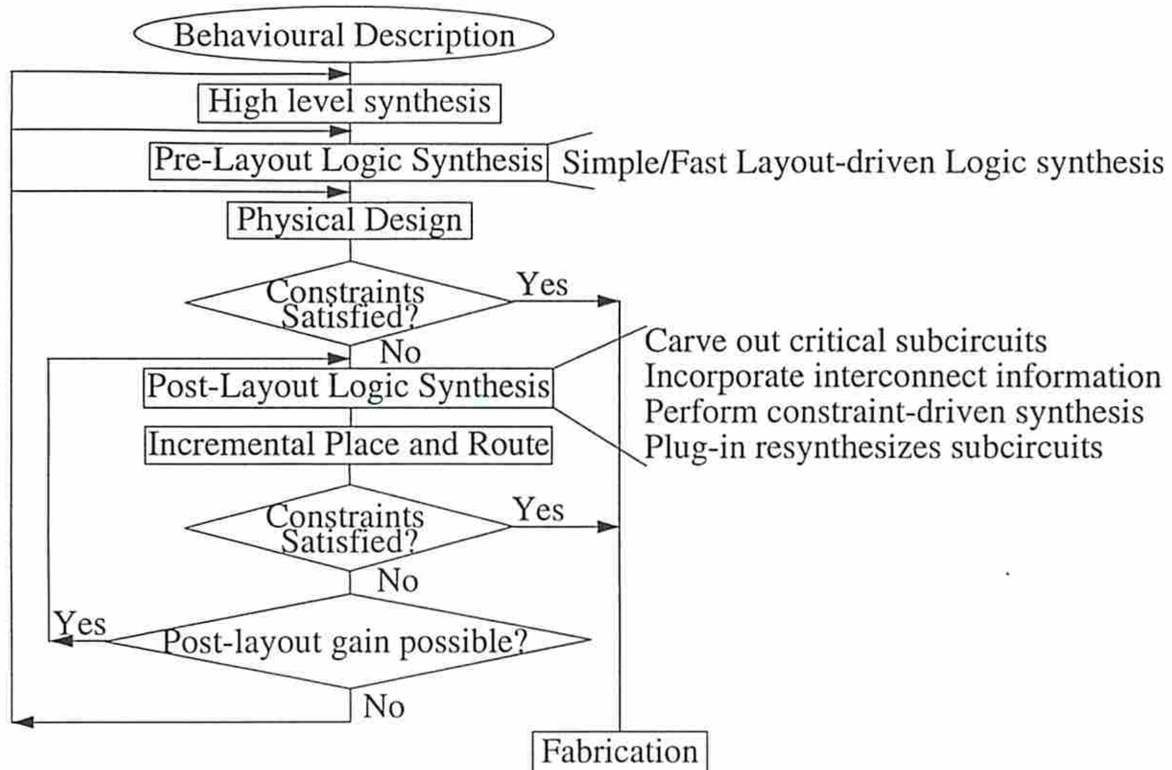


Figure 8.1: Alternate/future design flow.

circuits will have a much larger number of gates. Indeed, I believe that as interconnect contribution increases in dominance, most of the synthesis will be performed in a post-layout resynthesis phase where a greater control over interconnect contribution is possible. This is illustrated in Figure 8.1. This post-layout resynthesis will require new synthesis techniques that perform in place optimization with little perturbation to the rest of the circuit. Also, since the circuit size during the pre-layout synthesis will be large, only simple/fast synthesis operations (e.g., structural or algebraic manipulations) will be performed during pre-layout synthesis, moving expensive operations (e.g., Boolean manipulations) to post-layout resynthesis phase.

I have presented an effective approach for post-layout logic resynthesis [113]. Generic nature of the approach makes it applicable for any post-layout synthesis operation. However, for sake of brevity, I am presenting here only a brief discussion of the post-layout resynthesis approach that proposes a methodology and a set of techniques for performing post-layout logic synthesis in order to correct certain timing violations in large-scale designs.

The proposed methodology and techniques promise significant improvements (in terms of the circuit quality and design time) compared to the traditional approach that performs synthesis and layout one after the other. Two general problems related to post-layout resynthesis (i.e., region selection and constraint-based logic synthesis) are described and possible first-order solutions are outlined.

Many things can go wrong in a design, including timing violations, area/wire-ability problems and excessive power dissipation. Currently, I focus on post-layout timing violations. The proposed approach is however general and can handle other types of constraint violations.

The general flow of the proposed approach is as follow. Given a placed design where some or all of the nets have been routed, we, 1) Identify the "candidate regions" (hereafter referred as C-regions) for resynthesis; 2) Annotate these regions with the desired behavior necessary to satisfy the timing constraints; 3) Resynthesize these regions and put them back into the circuit.

In the simplest case, each region consists of a single gate. Thus, the only transformations applicable to each region are re-powering (also known as gate sizing or drive strength selection) and buffering (more generally, fanout optimization). The region can however consist of several gates whereby more powerful logic operations (e.g., critical signal isolation, collapse in two-level logic form followed by logic restructuring, logic minimization using appropriate don't care information, re-mapping) can be applied. These transformations have greater potential to meet the constraints at little or no cost, however, if they are not applied carefully and judiciously, the resultant circuit may be very different from the original one, thus necessitating the repeat of placement and routing steps in order to obtain the correct area/delay values, and we are back in the synthesis-layout iterations that we were trying to avoid in the first place.

My answer to this problem is two-fold: 1) Select the target resynthesis regions carefully, that is, these regions should have a good potential for local timing improvement, timing improvement in these regions should translate to timing improvement for the whole circuit, there is some slack to increase their area without causing a major perturbation to the existing circuit layout; 2) Modify the logic synthesis tools to satisfy timing constraints while honoring various topological (e.g., fixed positions for

the inputs and outputs of these regions) and size constraints (e.g., area and aspect ratio).

It is also important to note that the C-regions are derived in a layout environment while the resynthesis is performed in a synthesis environment. I therefore separate the aspect of analyzing the design to identify C-regions from that of actually re-synthesizing these C-regions. The first, namely *problem analysis*, is rightly addressed in the physical design phase where large designs (in the order of 100,000 movable objects or more) can be analyzed using a rich set of accurate analysis tools. The second, namely *problem resolution*, is rightly addressed in the synthesis phase, where small pieces of logic (in the order of a few thousand gates) can be handled using a rich set of powerful re-structuring techniques.

In the following, I summarize the proposed approach.

Region Identification The following factors must be considered when identifying the C-regions:

- **Timing Correction:** The regions should have good potential for improving the circuit timing with little increase in circuit area.
- **Layout Fit:** The regions should have some area slack so that if their area increases as a result of resynthesis, the re-synthesized region can be incrementally placed without perturbing the existing circuit.

These regions are formed by a greedy cluster-growth algorithm. First, a set of seed gates are selected based on their timing criticality and their potential for growing into a region with good timing improvement potential. Next, these seeds are grown into regions while taking into account the estimated speed-up potential and the corresponding area penalty.

Seed Generation An important problem in post-placement resynthesis is that of finding a minimum number of regions to be re-synthesized. This is desirable to minimize the perturbation to the existing layout. The following vertex separator based mechanism [65] is used for generating the seeds[2].

[2]A vertex separator is a set of vertices whose removal causes source vertices (e.g., circuit inputs) to be disconnected from sink vertices (e.g., circuit outputs).

- Based on the slack information, a critical DAG of the underlying Boolean network is created. Each vertex in this DAG corresponds to a gate with a negative slack pin while each edge corresponds to a connection with negative slack.
- A minimum cardinality vertex separator of this critical DAG is constructed[3].

It is desirable to have a vertex separator such that no vertex in the set is in the transitive fanin cone of any other vertex in the set. This is a sufficient condition for timing independence of these vertices. Such a vertex separator is referred to as a *timing-independent* vertex separator. Now then, if each vertex i of this vertex separator is sped up by $-slack_i$ where $slack_i$ is the slack at vertex i and if min_slack is the maximum negative slack among all vertices on the vertex separator, then a total circuit speed up of min_slack is assured. However, it is likely that the vertex separator has timing-dependent vertices. In this case, by speeding up each vertex in the set by Δ_i, a total circuit speed up of min_slack is still assured. However, some vertices may have been sped up more than necessary due to time dependency between two vertices possibly resulting in an increased area penalty. Alternately, instead of creating a 0-critical DAG (i.e., a critical DAG where each vertex and edge has a slack of -0 or less), a smaller k-critical DAG could be created. In this case, the best global slack after resynthesis is limited to $-k$. However, the value of k can be incrementally decreased and resynthesis operation can be applied iteratively to achieve a 0 slack circuit. Details of this procedure are reported in [113].

Cluster Growth After identifying the seeds, each seed is grown into a region based on the following factors:

[3]The vertices may be weighted to reflect their potential for local and/or global timing improvement, the amount of available area slack on them, etc. In this case, a minimum weighted vertex separator must be constructed.

- **Topological Connections:** Logic re-synthesis works better on larger clusters as more logic transformations are applicable. Each of these clusters should be topologically connected.
- **Timing Correction:** Regions should have good potential for local timing improvement. For example, a region consisting of equally critical gates may not have any potential for timing improvement and hence should be avoided. Note that because of the seed selection, it is guaranteed that local timing improvement in these regions will lead to global timing improvement in the circuit.
- **Timing Independence:** To avoid logic duplication, the regions should be disjoint from each other. Furthermore, to avoid unnecessary area penalty, the set of regions should also remain timing-independent, that is, changing the arrival time at the output of one region does not change the arrival time of any other region.
- **Geographical Proximity:** Gates that are geographically near a C-region (but are not topologically connected to it) can be re-synthesized (based on the amount of positive slack on them) to consume less area (at the cost of their slowing down), thus freeing some area that can then be used during the resynthesis of the associated C-region.
- **Aspect Ratio:** It is easier to replace a region which was originally square with another square region without perturbing the existing layout.
- **Empty Area:** Empty (dead) layout area in the bounding box enclosing gates in the C-region can be used to advantage during the resynthesis by relaxing the area increase constraint on the connected subgraph containing the seed.

The cluster growth is thus performed as follows. Given a set of seed gates, each seed is grown recursively by adding all of its immediate fanin nodes to it. Next, all gates in the bounding box enclosing the seed and its fanin gates are included in the seed. A check is performed to keep only those gates that assure timing independence from other seeds/regions. The

current growth is then accepted only if the region has a good resynthesis value (this is a function of timing correct-ability and area penalty). The process continues until no region can be expanded any further; At that point all regions are re-synthesized one at a time, re-mapped, individually placed and glued back to the existing layout.

It can be assumed that the regions are timing-independent as long as they are disjoint. The only effect of ignoring timing dependency between the regions is excessive speeding up of a region because another region in the input cone or output cone may already have been sped up[4]. In the worst case, this could lead to excess area overhead to achieve a given speed up. However, this can be resolved exactly by identifying exact dependencies between the regions. Alternatively, the cells can be re-synthesized sequentially or an area recovery operation can be performed on the regions after resynthesis. Once again, additional details are provided in [113].

Constraint-based Logic Resynthesis Once the C-regions are selected, the timing, topological and size constraints for each region are generated based on the current circuit timing and layout of the region. The synthesis tool then returns a collection of alternative implementations for each region. The physical dimensions, input/output connections and the external nets passing through the regions are already known. Based on these, timing and wire-ability analysis can be performed and the most suitable implementation can be selected.

The remaining question is how to resynthesize the region such that the timing constraints are satisfied with a minimal increase in area. Most of the existing logic synthesis operations address either a pure area minimization or a pure delay minimization. Timing-constrained area minimization is applied only to a few logic synthesis operations like fanout optimization. Also, apart from being able to perform timing constrained optimization, the synthesis operations should be able to explore area-delay trade offs and present a set of alternative implementations for re-placement. The availability of a synthesis tool that presents a set of alternative implementations is essential to such post-layout

[4]This is true only if the dynamic false path problem is ignored.

resynthesis framework as described next. Let the initial speed up required for C-region i be Δ_i to achieve a global speed up of $MAX_i\{\Delta_i\}$. It is likely that for some C-region i, the best possible speed up is still δ_i short of the required speed up Δ_i. In this case, to avoid extra area penalty, it is necessary to speed up all other C-regions j only by $\Delta_j - \delta_i$. This situation can be handled simply by picking an alternative implementation when sufficient alternative solutions are available for each C-region, thereby avoiding an expensive resynthesis iteration with modified timing constraints.

Thus, generating alternate solutions with different area-delay trade-offs is an important requirement for post-layout synthesis tools. The *Area-Delay Mapper (ADMap)* [13] is the only technology mapping algorithm which can provide efficient and accurate enumeration of all different area-delay trade-off points for the region. Though timing constrained optimization is inherent in the problem definition of fanout optimization, the problem solution needs to be modified to allow trade offs between delay and area and to be able to present a number of good fanout tree implementations with different area penalties. Likewise, techniques allowing exploration of alternate implementations need to be developed/modified for delay optimization at the technology independent logic level, bounded-depth logic decomposition and extraction, two-level logic minimization for given delay constraint, use of network don't cares to speed-up various C-regions, use of logic equivalence and symmetry properties of certain inputs of a gate to improve its timing, transistor sizing, etc.

Thus, post-layout synthesis techniques promise significant timing improvements as well as faster turn-around times compared to the traditional approach that performs synthesis and layout sequentially. As shown in Figure 8.1, layout-driven approaches for pre-layout logic synthesis in combination with post-layout logic resynthesis will provide the answer to the challenge posed by deep-submicron technology.

Part II

Power-Driven Physical Design

Chapter 9

Delay Optimal Circuit Partitioning for Low Power

In this chapter, a delay optimal clustering/partitioning algorithm for minimizing the power dissipation of a circuit is proposed. Traditional approaches for delay optimal partitioning are based on Lawler's clustering algorithm [62] that, given a constraint on size of partitions, minimizes the circuit depth under a unit delay model, i.e., it minimizes the maximum number of partitions along any path from circuit inputs to circuit outputs. Lawler's algorithm guarantees a depth-optimal partitioning but it makes no attempt to explore alternative partitioning solutions that may have the same depth but better power implementation. The algorithm proposed here provides a formal mechanism which implicitly enumerates alternative partitioning solutions and selects a partitioning solution that has the same depth but less power dissipation. Indeed, for tree circuits, the proposed algorithm produces delay and power optimal partitioning whereas for non-tree circuits it produces delay optimal partitioning with significantly improved power dissipation. The experimental results indicate that on average, this approach obtains 35% improvement in power dissipation at no loss in delay.

9.1 Introduction

Circuit partitioning is a technique to divide large circuits into smaller physical or logical sub-components. Partitioning is necessary during synthesis and/or layout due to either of the following reasons: physical limitations on the number of transistors a

chip or module can accommodate (e.g., to implement a large design on multiple chip modules or on fixed size PLA/FPGAs), for performing circuit restructuring during synthesis, for reducing complexity of synthesis or layout procedures by reducing the problem size, etc.. In most of these applications, a net that is external to a part drives a significantly larger load than an internal net. This implies that such inter-part nets contribute significantly to the circuit delay. An improper assignment of gates to parts can thus seriously degrade the performance of the circuit. Specifically, if tightly connected cells are put on different parts, paths may cross a partition boundary many times resulting in significant degradation of performance. Thus, it is important to partition a circuit using a performance-driven partitioning technique.

Initial work on performance-driven circuit partitioning was presented by Lawler et al. in [62]. His approach was a bottom-up approach which is usually referred to as "clustering" in contrast to a top-down approach which is referred to as "partitioning". Given a constraint M on the size of parts, Lawler's algorithm produces a delay optimal partitioning of the circuit by labeling gates based on the capacity of partial clusters at the input of a gate during a postorder traversal of the circuit. The partitioning produced by Lawler's algorithm is delay optimal under the assumption that internal delays within a cluster are zero and the external delay from one cluster to the other is one.

The main disadvantage of clustering algorithms based on Lawler's algorithm is that given a cluster size M, it arbitrarily selects one delay optimal clustering without providing the flexibility of exploring alternative delay optimal clustering solutions. Indeed, a large number of clustering solutions may exist with the same delay but different amount of power dissipation. For example, consider the benchmark circuit *con1* in Figure 9.1 where the values at the output of each gate corresponds to the switching activity as calculated by the symbolic simulation mechanism [30]. Square boxes in the figure correspond to the circuit inputs and outputs. Each circuit input is assumed to have a switching activity of 0.5. Figure 9.1(a) shows the delay optimal clustering solution generated by the Lawler's algorithm. Figure 9.1(b) corresponds to the power optimal clustering solution generated by my algorithm that also has optimal depth. By selecting an alternative solution, the visible power dissipation of *con1* is reduced by 36% whereas the total power dissipation is reduced by about 5%.

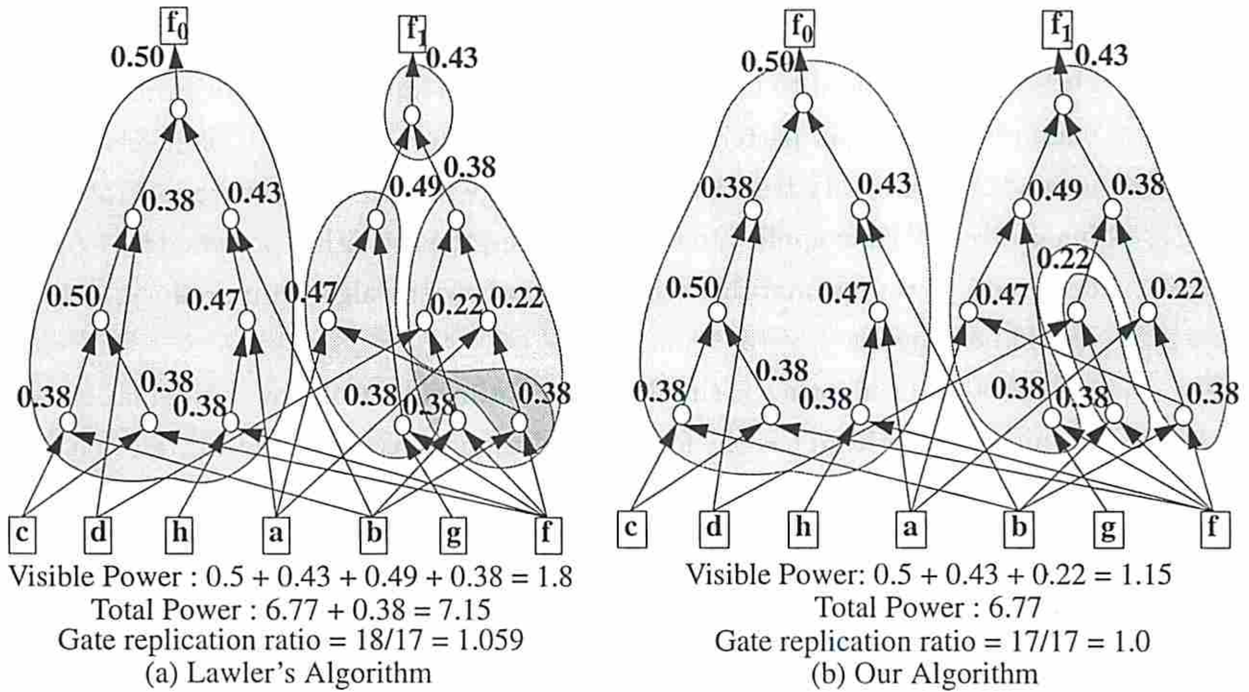


Figure 9.1: Two delay optimal clusterings under size constraint 8 for circuit *con1*.

Hence, if the objective is to minimize the power dissipation during circuit clustering, it is necessary to explore alternative clustering solutions.

I propose a systematic approach for delay optimal clustering that implicitly enumerates all clustering solutions for a tree and selects a power optimal clustering from all delay optimal clustering solutions. The main idea is to maintain only the power optimal clustering solution for different delay values at every gate. This is achieved by enumerating all cluster patterns of size M at every gate starting from the primary inputs in a postorder fashion and selecting the power optimal cluster pattern for each delay value[1]. It is shown here that the problem of enumeration of all cluster patterns of size M at a gate i is related to the problem of enumerating all alphabetic trees of size m rooted at i where $m = \lceil \frac{M}{w_{min}} \rceil$ and w_{min} is the size of the smallest gate. This approach is similar in concept to the approach proposed

[1]An approach to enumerate different cluster patterns in the context of FPGA mapping was proposed by Murgai et al. [74]. However, they are enumerating clusters under a constraint on the number of inputs to a cluster whereas I enumerate clusters under a constraint on the size of the cluster.

for technology mapping in [13] with two significant differences: 1) technology mapping works within the context of available library patterns whereas in this case, all cluster patterns need to be enumerated thereby requiring an algorithm to efficiently enumerate all cluster patterns; 2) In this case, cluster enumeration needs to be performed on non-binary trees whereas technology mapping procedures operate only on binary trees. When applied to tree structures, the algorithm proposed here produces delay and power optimal clustering whereas Lawler's algorithm produces only delay optimal clustering.

Apart from allowing a simultaneous optimization of power dissipation of the circuit, my algorithm has the following advantages as compared to traditional delay optimal partitioning algorithms.

- It produces a delay optimal clustering under a generalized load independent delay model.

- The proposed algorithm can be used to minimize objective functions other than power dissipation.

- The proposed algorithm produces a set of non-inferior power-delay solutions, allowing for an informed trade-off between delay and power dissipation of the circuit.

- If required, minimum cluster size constraints can be easily incorporated in the proposed algorithm, thereby allowing a partitioning solution with approximately equal size partitions.

- The method can be easily adapted to perform partitioning in a hierarchical manner thereby allowing larger partition sizes at little gate replication penalty.

The rest of the chapter is organized as follows. In the next section, I present the power model used for power estimation/minimization in this chapter and the next chapter. Section 9.3 presents a brief description of Lawler's algorithm and describes other work in performance-driven clustering. In Section 9.4, my approach for power optimal clustering for tree structures is presented. Generalization of this approach for DAGs and the consequences are described in Section 9.5. Section 9.7 presents experimental results and some empirical studies that characterize the impact of

maximum cluster size M on the algorithm. Concluding remarks are presented in Section 9.8.

9.2 The Power Model

Power consumption in CMOS circuits is caused by three sources: the charging and discharging of capacitive loads during output switchings, the short circuit current which flows during output transitions and leakage current. The last two sources can be made small with proper device and circuit design techniques. CAD tools have thus concentrated on the dynamic power consumption.

The average dynamic power consumed by a CMOS logic gate i (if the gate is a part of a synchronous digital system controlled by a global clock) is given by

$$P_i^{average} = 0.5 \frac{V_{dd}^2}{T_{cycle}} C_i^{load} N_i \tag{9.1}$$

where C_i^{load} is the load capacitance, V_{dd} is the supply voltage, T_{cycle} is the global clock period, and N_i is the number of gate output transitions per clock cycle (i.e., *switching activity* of gate i).

The fanout load C^{load} in equation (9.1) consists of two components: the gate load C^{gate} which accounts for the input capacitances of fanout gates and the drain to ground capacitance of the driver, and the wire load C^{wire} which accounts for the load due to the interconnection tree formed between the driver and its fanout gates. Logic synthesis for low-power attempts to minimize $\sum_i C_i^{gate} N_i$, whereas physical design for low-power should minimize $\sum_i C_i^{wire} N_i$.

Estimating Switching Activities

The number of transitions N a gate makes during a clock cycle is a complex function of its global logic function, delays in the circuit, and the input sequences applied. Given a set of input vector sequences, exact number of transitions can be computed by time-consuming simulation. In most cases, however, exhaustive simulation is not a realistic option. Apart from this, when the input sequences are not known, simulation can only produce an estimation of switching activity.

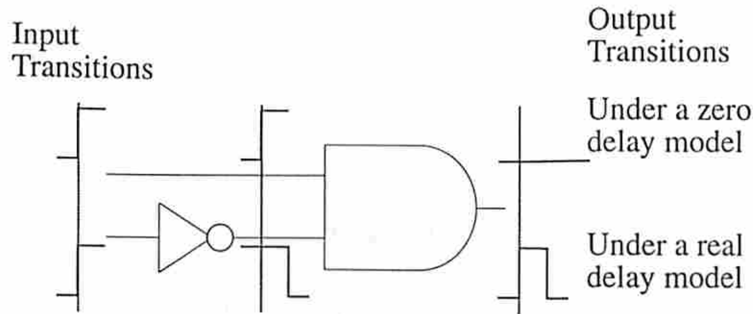


Figure 9.2: An illustration of effect of delay model on switching activity.

Probabilistic and symbolic simulation methods for estimating the number of gate output transitions have provided an attractive alternative to simulation. Burch et al. [8] introduce the concept of probability waveforms. Given such waveforms at the primary inputs and with some convenient partitioning of the circuit, they examine every sub-circuit and derive the corresponding waveforms at the internal circuit nodes. Najm [75] describes an efficient technique to propagate transition densities at the circuit primary inputs up in the circuit and thus estimate the total power consumption (ignoring signal correlations due to re-convergent fanout). Ghosh et al. [30] address the problem of estimating the average power consumption in VLSI combinational and sequential circuits using a symbolic simulation technique. Under a pure binary delay model, Tsui et al. [106] present a power analysis technique for CMOS circuits which estimates the signal and transition probabilities at internal nodes of a circuit accounting for re-convergent fanouts. This approach is more efficient (in terms of memory and run time) than previous work as it approximates the dynamic correlations among intermediate signals via their steady-state correlations, yet the error introduced is negligible.

Effect of Delay Model on Switching Activity Estimation

At each gate, the switching can be classified into two types: steady state switching and switching due to hazards [25]. The steady state transitions are caused due to a change in the value of the global function of the gate (i.e., function of the gate in terms of primary input of the circuit). Hazardous transition occur due to different arrival times at the immediate inputs of the gate. This is illustrated in Figure 9.2.

As shown in the figure, if all gates are assumed to have zero delays, input transitions are propagated instantaneously through the circuit. Consequently, the output of a gate may make a transition at most once during any cycle. Hence, power dissipation calculated under a *zero delay model* thus ignores glitches and captures only the steady state or functional switching of the gates [30]. In reality, however, gates have a real finite delay resulting in additional power dissipated due to glitches at some gates. If the power dissipation due to glitches is a significant portion of total power dissipation of the circuit, power should be estimated under a *real delay model.* It has been shown in [3] that glitching activity typically accounts for about 15-20% of total power dissipation of the circuit.

A Simplified Power Model for Circuit Partitioning

The fanout load ${C_i}^{load}$ in equation (9.1) consists of two components: the basic net load ${C_i}^{basic}$ which accounts for the load capacitances seen by gates in absence of any partitioning, and the extra load ${C_i}^{extra}$ which accounts for the additional load capacitance due to the external connections of the net. Note that for a net with no external connections ${C_i}^{extra} = 0$.

Then, the total power dissipation of the circuit $\mathcal{G}$ is given by

$$P_{\mathcal{G}} = 0.5\frac{{V_{dd}}^2}{T_{cycle}}\sum_{i\in\mathcal{G}}({C_i}^{basic} + {C_i}^{extra})N_i \tag{9.2}$$

When a circuit partitioning corresponds to a physical partitioning, the additional load driven by gate i driving an external net, namely, ${C_i}^{extra}$, is one to three orders of magnitude larger than that of an internal net (i.e., ${C_i}^{basic} << {C_i}^{extra}$). In this case, the power model given in (9.2) can be simplified further as follows. First, it can be assumed that the power dissipation contribution that can be attributed to variations of ${C_i}^{basic}$ under different partitioning solutions is negligible, i.e., ${C_i}^{basic} = 0 \;\; \forall i \in \mathcal{G}$. Furthermore, considering that the fixed overhead capacitance for an external net is dominant within ${C_i}^{extra}$, it can be assumed that ${C_i}^{extra}$ is identical for each net. Under these assumptions, the objective function to be minimized during partitioning is given by

$$O_{\mathcal{G}} = \sum_{i\in\mathcal{G}_v} N_i \tag{9.3}$$

where $\mathcal{G}_v$ corresponds to the set of gates that are visible, i.e., the set of gates that drive a load external to the partition. As described in Section 9.7, the algorithm proposed here can minimize power dissipation under both the basic power dissipation model given in equation (9.2) and the simplified power dissipation model given in equation (9.3) (i.e., when $C_i^{basic} << C_i^{extra}$).

Note that depending on the application, i.e., whether targeting a physical partitioning or a logical partitioning, the proposed algorithm can be used to optimize either equation (9.3) or equation (9.2).

9.3 Background

Given a directed acyclic combinatorial circuit $\mathcal{G}$, the set of inputs of a gate $i \in \mathcal{G}$ are denoted by $\mathcal{I}_i$. The weight (i.e., area) of gate i is denoted by w_i and the label of the gate is denoted by l_i. Let $W_i(l_i)$ denote the total weight of all the gates with label l_i in the transitive fanin of gate i.

A **cluster** is defined as a connected rooted DAG where the vertices correspond to the gates belonging to the cluster, edges correspond to the connections between the gates and the root of the cluster is the gate reachable by all the gates in the cluster (i.e., all the gates in the cluster are in the transitive fanin of the root of the cluster).

Let the constraint on the size of the cluster (i.e., sum of the weights of gates belonging to that cluster) be M. Then, Lawler's algorithm [62] produces a depth-optimal partitioning as follows:

1. Gates with in-degree 0 (i.e., circuit inputs) are given the label 0.

2. Find any unlabeled gate i such that all gates belonging to $\mathcal{I}_i$ have been labeled. Let k be the largest label applied to any of these gates. If $w_i + W_i(k) \leq M$ then $l_i = k$. Otherwise, $l_i = k + 1$.

3. After the gates are labeled, locate all gates that have a label distinct from all of its outputs. Such gates correspond to the roots of a cluster. The root gate and all the gates in the transitive fanin of the root with the same label constitute a cluster.

Figure 9.1(a) illustrates the partitioning obtained by applying Lawler's algorithm to the benchmark circuit *con1* with a size restriction 8.

As a consequence of the labeling scheme, when a gate has multiple fanouts, it is likely that the gate is replicated in clusters containing each of its fanouts. Thus, the delay optimality is achieved at the cost of an increase in the gate area. This gate area can be recovered by some post-clustering operation during which clusters are merged into larger clusters such that the size constraint is not violated and the gate replication is reduced. Lawler indicates that the problem of merging to minimize the gate replication is similar to that of rectangle covering which is shown to be NP-hard. Lawler also proposes a heuristic to minimize gate replication. Murgai et al. [73] have proposed some heuristics to reduce the number of clusters and to reduce the gate replication. Touati [104] has also developed a gate area recovery mechanism based on a relabeling scheme.

Murgai et al. [73] have modified Lawler's algorithm to allow a generalized load independent delay model, i.e., a delay model in which each gate i has delay of $\delta(i)$ time units and each inter-cluster net has delay of d time units[2]. However, they show that the modified Lawler's algorithm they propose is delay optimal only under certain special conditions. Recently, a clustering algorithm that produces delay optimal clustering under a load independent delay model has been proposed in [85]. Both these algorithms still rely on the basic framework of Lawler's algorithm and hence, have no flexibility to explore alternative clustering solutions.

9.4 Power Optimal Partitioning for Trees

9.4.1 Definitions and Notations

A **Cluster Pattern** at a gate i, denoted P_i, is defined as a cluster of size less than or equal to M with gate i as the root of the cluster. The concept of cluster patterns is illustrated in Figure 9.3. Note that each cluster pattern has a set of leaf nodes

---

[2] To emphasize that in spite of being more general than unit delay model, this model still assumes that gate delays are independent of the load driven by gates, I refer to this delay model as a "generalized load independent delay model" in contrast to the term "generalized delay model" used in [73].

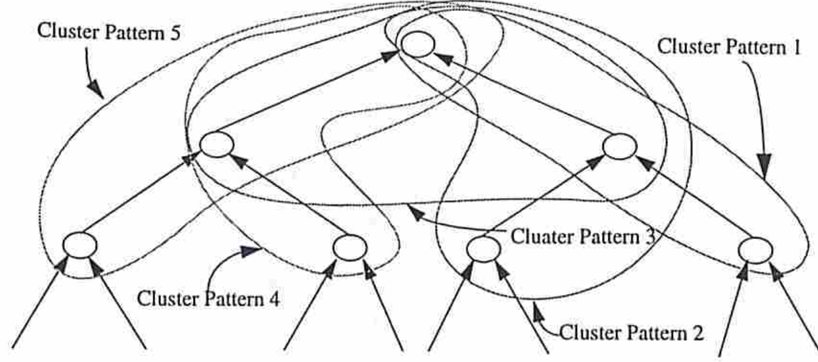


Figure 9.3: An illustration of size 3 cluster patterns.

associated with it. The set of leaf nodes associated with a cluster pattern P_i is referred to as its **leaf set** $\mathcal{L}_i$.

A **clustering solution** at a gate i is characterized by a **PD-point** (power-delay point) which is a 3-tuple $\{a_i, p_i, \mathcal{LP}_i\}$ where a_i gives the delay value (i.e., arrival time) associated with the PD-point, p_i gives the corresponding power cost of the clustered sub-circuit rooted at gate i and $\mathcal{LP}_i$ denotes the corresponding set of PD-points at the leaf nodes of the cluster pattern. It should be noted that a clustering solution at gate i determines the clustering solution of the entire sub-circuit rooted at gate i.

Given $\mathcal{LP}_i$, the power cost p_i and the arrival time a_i are calculated as follows.

$$p_i = N_i + \sum_{l \in \mathcal{LP}_i} p_l \tag{9.4}$$

$$a_i = max_{l \in \mathcal{LP}_i}\{a_l + d_l\} + d \tag{9.5}$$

where d_l corresponds to the delay internal to the cluster from the leaf node l to the root of the tree i whereas d corresponds to the delay of the external net. Note that under a unit delay model, $d_l = 0$ and $d = 1$.

It should be noted that when the internal power dissipation is also important, equation (9.4) can be modified to account for internal power dissipation as shown below.

$$p_i = N_i * C_i^{extra} + \sum_{j \in \mathcal{IN}_i} N_j * C_{j,i}^{basic} + \sum_{l \in \mathcal{LP}_i} p_l \tag{9.6}$$

where $\mathcal{IN}_i$ corresponds to the set of gates that are internal to the cluster at i and ${C_{j,i}}^{basic}$ corresponds to the load driven by gate j that is within the cluster formed at i.

For each gate, at most one PD-point is allowed for each delay value. The set of PD-points at a gate i is referred to as the **PD-set** at gate i (denoted by $\mathcal{S}_i$). Within a PD-set it is ensured that the PD-points are **non-inferior** to each other, i.e, $\forall j,k \in \mathcal{S}_i, {p_i}^j \neq {p_i}^k, {a_i}^j \neq {a_i}^k, {p_i}^j > {p_i}^k \Rightarrow {a_i}^j < {a_i}^k$ and ${p_i}^j < {p_i}^k \Rightarrow {a_i}^j > {a_i}^k$. This set of non-inferior PD-points allows trade off of delay for power and vice versa. A graphical representation of such PD-set is shown in Figure 9.4.

Given a leaf set $\mathcal{L}_i$, the **arrival time set** (denoted $a_{\mathcal{L}_i}$) is defined as a set with the same cardinality as $\mathcal{L}_i$ with each element corresponding to a arrival time at the corresponding leaf node. If the latest time a signal may arrive at any input is given by A_{max} and the earliest time all signals are ready at the inputs is given by A_{min}, i.e., if

$$A_{max} = max_{l \in \mathcal{L}}\{max_{j \in \mathcal{S}_l}\{{a_l}^j\}\} \text{ and } A_{min} = max_{l \in \mathcal{L}}\{min_{j \in \mathcal{S}_l}\{{a_l}^j\}\}$$

then the number of such arrival time sets is restricted by the range $[A_{min}, A_{max}]$. Under a unit delay model, for example, the length of this range will be limited by the clustered depth of the resultant circuit. Under a generalized load independent delay model, the total delay through the circuit can be discretized with respect to the GCD of individual gate delays and net delays, resulting in a linear number of arrival time instances at each gate. This maximum discretized delay is denoted by D. Thus, the maximum size of each PD-set is bounded by D.

If w_{min} corresponds to the size of the smallest gate, then the maximum gate count of a cluster, denoted by m, is calculated as $m = \lceil \frac{M}{w_{min}} \rceil$. Finally, it is assumed that the maximum number of inputs of a gate before clustering is T.

9.4.2 An Enumerative Clustering Algorithm

A power optimal clustering can be obtained by exhaustively enumerating all possible clustering solutions of the circuit. However, such an approach is computationally infeasible. Instead, I propose a dynamic programming approach that generates power

optimal clustering solutions at a gate given the power optimal clustering solution at each gate in its transitive fanin.

In order to apply a dynamic programming approach to explore alternative clustering solutions and to identify the delay and power optimal clustering solution for the circuit, the clustering mechanism should satisfy the following two conditions: 1) It should provide the freedom to explore all cluster patterns at a gate; 2) For each pair of such patterns, it should be able to correctly characterize the relative delay and power cost of each pattern and select the best cluster pattern which guarantees delay and power optimality of the entire circuit.

Assuming that mechanisms that satisfy both of these conditions (i.e., mechanisms that enumerate and characterize cluster patterns at each gate) can be provided, the outline of my algorithm is as follows. POWER_CLUSTER visits each gate in a postorder (i.e., all inputs of a gate are visited before the gate is visited) and generates a power optimal clustering at each gate. In fact, the algorithm maintains a set of non-inferior power-delay solutions at each gate. The inherent benefit of maintaining such non-inferior solutions during a dynamic programming algorithm is that if the cost of the solution to a problem is monotone in the cost of the solutions to its subproblems, then these non-inferior solutions are sufficient to guarantee optimality of the main problem.

```
Algorithm 1 POWER_CLUSTER (𝒢, M_min, M_max)
01  for each gate i ∈ 𝒢 (in postorder) do
02    for size = M_min to M_max
03      ℒ = first_leaf_set(i, children(i), size)
04      do
05        for each set of arrival times a_ℒ for ℒ
06          a = output_arrival_time(ℒ, a_ℒ, i)
07          p = power_dissipation_after_merge(ℒ, a_ℒ)
08          update_PD-set(a, p)
09        ℒ = next_leaf_set(i, children(i), ℒ, size)
10      while (ℒ ≠ NULL)
11  ℒ = set of primary outputs
12  for each set of arrival times a_ℒ for ℒ
13    a_o = max(a_ℒ)
14    p_o = power_dissipation_after_merge(ℒ, a_ℒ)
15    update_PD-set(a_o, p_o)
16  Select_a_non-inferior_solution
```

In the following, I describe the operation of POWER_CLUSTER in the context of tree circuits[3]. Modification of this algorithm to handle general DAGs will be presented in Section 9.5.

The routines *first_leaf_set* and *next_leaf_set* on Line 3 and Line 9, respectively, identify each cluster pattern as will be described in Section 9.4.4. Once the leaf set $\mathcal{L}$ corresponding to a cluster pattern P_i is available, the sets of arrival times at gate i are determined. Since the worst size of $[A_{min}, A_{max}]$ is D, the maximum number of arrival time sets and the maximum number of execution of Lines 5-8 for each cluster pattern is D. Thus, maintaining a PD-set as opposed to maintaining a single delay optimal clustering at each gate increases the time complexity of the algorithm by a multiplicative factor D.

[3]The same analysis is also applicable to leaf-DAGs, i.e., circuits that have multiple fanouts only at the circuit inputs.

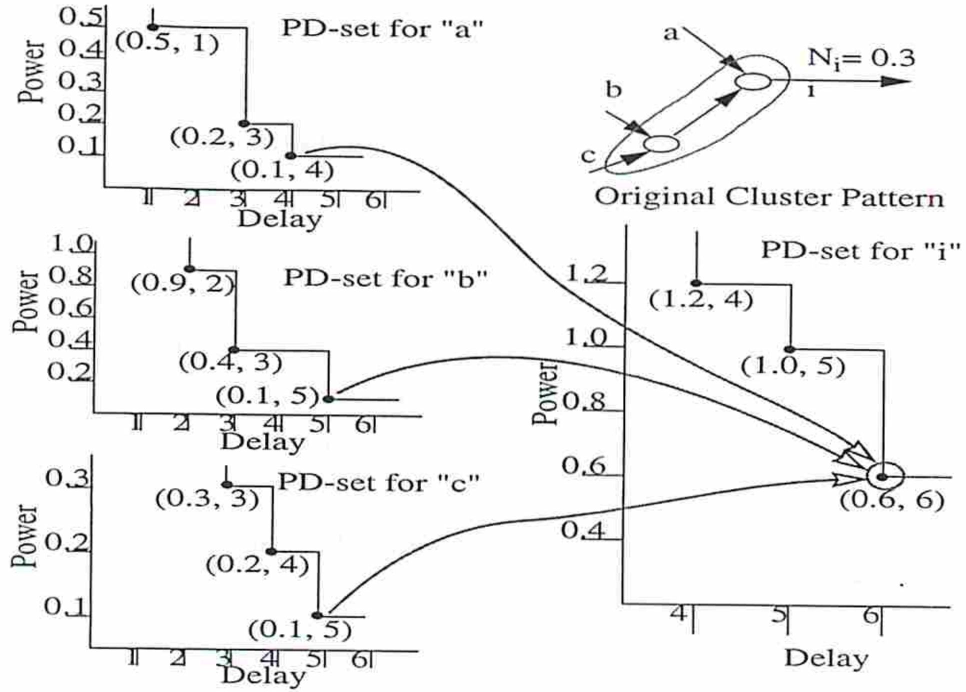


Figure 9.4: An illustration of PD-set.

Given the PD-sets at each element in leaf set $\mathcal{L}$, Line 5 in POWER_CLUSTER enumerates all arrival time set $a_{\mathcal{L}}$ for the leaf set $\mathcal{L}$. For example, if the leaf set $\mathcal{L} = \{a, b, c\}$ and if

$$\begin{aligned}
\mathcal{S}_a &= \{\{1, 0.5, \mathcal{LP}_a{}^1\}, \{3, 0.2, \mathcal{LP}_a{}^2\}, \{4, 0.1, \mathcal{LP}_a{}^3\}\} \\
\mathcal{S}_b &= \{\{2, 0.7, \mathcal{LP}_b{}^1\}, \{3, 0.4, \mathcal{LP}_b{}^2\}, \{5, 0.1, \mathcal{LP}_b{}^3\}\} \\
\mathcal{S}_c &= \{\{3, 0.3, \mathcal{LP}_c{}^1\}, \{4, 0.2, \mathcal{LP}_c{}^2\}, \{5, 0.1, \mathcal{LP}_c{}^3\}\}
\end{aligned}$$

then $A_{min} = 3, A_{max} = 5$ and the arrival time sets are given by $\{3, 3, 3\}, \{4, 3, 4\}$ and $\{4, 5, 5\}$ where the first, second and the third elements correspond to the leaf nodes a, b and c respectively. The output PD-set corresponding to the above example is shown in Figure 9.4.

For generalized load independent delay model, the delay from the leaf node to the root is first added to the value of each PD-point of the leaf before identifying A_{min} and A_{max}. For example, if the delays from leaf nodes a, b and c to the root

of the cluster are $3, 1$ and 2, respectively, then the modified PD-sets (denoted by $\mathcal{S}_a', \mathcal{S}_b'$ and $\mathcal{S}_c'$, respectively) are given by

$$\begin{aligned}
\mathcal{S}_a' &= \{\{4, 0.5, \mathcal{LP}_a{}^1\}, \{6, 0.2, \mathcal{LP}_a{}^2\}, \{7, 0.1, \mathcal{LP}_a{}^3\}\} \\
\mathcal{S}_b' &= \{\{3, 0.7, \mathcal{LP}_b{}^1\}, \{4, 0.4, \mathcal{LP}_b{}^2\}, \{6, 0.1, \mathcal{LP}_b{}^3\}\} \\
\mathcal{S}_c' &= \{\{5, 0.3, \mathcal{LP}_c{}^1\}, \{6, 0.2, \mathcal{LP}_c{}^2\}, \{7, 0.1, \mathcal{LP}_c{}^3\}\}.
\end{aligned}$$

Apart from this, the rest of the algorithm is independent of the delay model used. For each such arrival time set, Line 6 of the algorithm calculates the arrival time at the output of the cluster. This is simply computed by adding d to the maximum value in the arrival time set. Thus, the overall time complexity of Line 6 is $O(1)$ for unit delay model and $O(m)$ for the generalized load independent model.

Line 7 computes the power cost corresponding to this clustering solution for gate i using equation (9.4). For example, for arrival time set $\{3, 3, 3\}$ (under unit delay model) the power cost of the clustering solutions for the transitive cone corresponding to each leaf is given by $0.2 + 0.4 + 0.3 = 0.9$. As illustrated in Figure 9.4, if the switching activity of gate i is 0.3 then p is calculated as $0.9 + 0.3 = 1.2$. Thus, at the end of Line 7, I have identified a PD-point for gate i with $a = 4$ (i.e., $3 + 1$) and $p = 1.2$. The time complexity of *power_dissipation_after_merge* is $O(m)$.

Line 8 of the algorithm adds this PD-point to the PD-set of gate i if it has lower power dissipation with respect to the faster solutions. Whenever a PD-point is introduced to a PD-set, all the slower solutions that have become inferior (i.e., have worse power cost) to the newly introduced PD-point are removed. Since the size of the PD-set is limited by D, the time complexity of *update_PD-set* is $O(D)$.

Lines 11-15 apply a similar technique to the leaf set corresponding to the primary outputs of the circuit. This identifies non-inferior PD-points for the entire circuit in the context of the slowest primary output. In other words, if the earliest arrival time at an output is 4, then it is not necessary to select a clustering solution with arrival time 2 at another output when a clustering solution with lower power dissipation but with arrival time 3 is available at that output. Finally, Line 16 of the algorithm allows one to select the solution based on the delay and/or power constraints. This determines the selection of PD-points corresponding to each primary output. Using these PD-points a clustering solution is generated by traversing the circuit top-down

(i.e., from primary outputs to primary inputs). Specifically, given that PD-point j is chosen at a gate i, the clustering solution at the corresponding inputs is generated by recursively generating a clustering solution for each element in $\mathcal{LP}_i^j$.

The time complexity of POWER_CLUSTER is given by $|\mathcal{G}|^2 \cdot m \cdot O(D) \cdot (O(m) + O(D)) \cdot number_of_patterns = O(|\mathcal{G}|^2 \cdot (Dm^2 + mD^2) \cdot number_of_patterns)$.

9.4.3 Number of Distinct Cluster Patterns

POWER_CLUSTER crucially depends on effective and efficient enumeration of cluster patterns. In this section, a cluster enumeration algorithm is described and it is shown that the number of cluster patterns with size M is limited by the number of alphabetic trees with $m(T-1)+1$ leaf nodes. Specifically, it is shown that if m and T are bounded by a constant, the runtime of POWER_CLUSTER, though exponential in m and T, is polynomial is the circuit size $|\mathcal{G}|$. Before relating the number of cluster patterns (and hence the time complexity of POWER_CLUSTER) to the number of alphabetic trees, I define alphabetic trees and some relevant notations. Further details on alphabetic trees can be found in Chapter 3.

An **alphabetic tree** is a tree generated on an ordered set of leaf nodes such that internal edges do not cross each other. The number of alphabetic trees on n leaf nodes are denoted by $\Phi_n{}^{|t|}$ where each internal node has arity t (i.e., number of children). Let Φ_n denote the number of alphabetic trees when there is no arity restriction. Likewise, let Π_M denote the number of cluster patterns under size restriction M and by $\Pi_M{}^{|t|}$ the number of cluster patterns if each gate in the circuit has exactly t inputs.

Lemma 9.4.1 *Each distinct cluster pattern generated at a gate corresponds to a distinct alphabetic tree.*

The proof of this lemma follows from the fact that if the original circuit has a tree structure, it can always be laid out such that no internal edges cross each other. On such a tree, each cluster pattern corresponds to a distinct set of leaf gates and hence a distinct alphabetic tree. This is illustrated with an example.

For the sake of illustration, it is assumed that each gate in the original circuit has two inputs and that each gate in the circuit has the same size, namely 1. In that

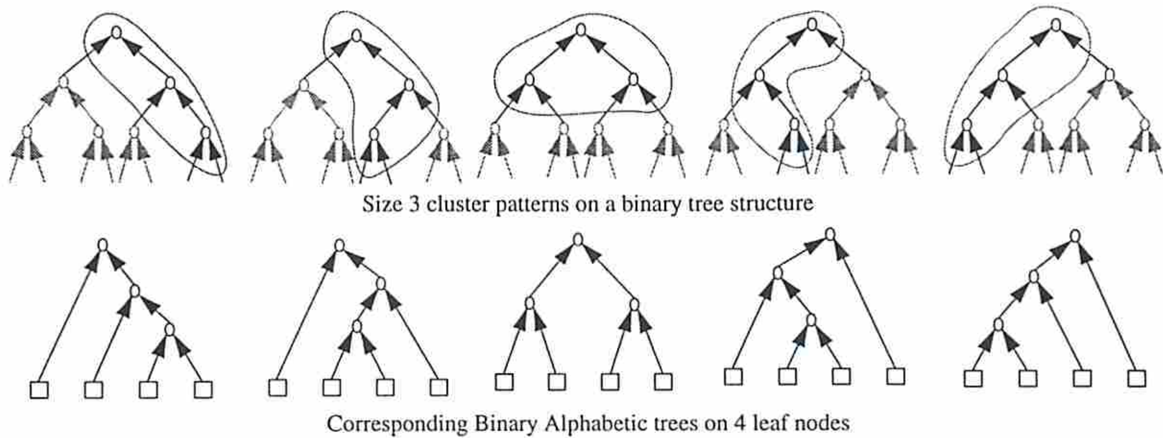


Figure 9.5: One to one correspondence between cluster patterns on binary tree structures and binary alphabetic trees.

case, $m = M$. This implies that the clustering algorithm is applied to a binary tree like the one shown in Figure 9.3 such that the number of internal gates in the cluster is restricted by m. Figure 9.5 shows all cluster patterns generated for $m = 3$ and all the alphabetic trees on $m + 1 = 4$ leaf nodes. As can be seen from the figure, there is a one to one correspondence between each distinct cluster pattern of size 3 and each binary alphabetic tree on 4 leaf nodes. Indeed, it can be shown that there is a one to one correspondence between the number of alphabetic binary trees on $m + 1$ leaf nodes and all size m cluster patterns on a binary tree with minimum height m.

Lemma 9.4.2 $\Pi_m{}^{|2|} = \Phi_{m+1}{}^{|2|}$.

If the original binary tree has height less than m then the number of cluster patterns will be less than $\Phi_m{}^{|2|}$. It can be shown that $\Phi_m{}^{|2|}$ corresponds to the Catalan numbers, i.e.,

$$\Phi_m{}^{|2|} = \begin{pmatrix} 2(m-1) \\ (m-1) \end{pmatrix} / m = O(4^m).$$

When the original circuit is non-binary, exact number of cluster patterns is determined by the tree structure. However, even in this case, an upper bound on the number of cluster patterns can be established by relating them to the enumeration of non-binary alphabetic trees.

For non-binary tree structures, the worst case m is given by $\lceil \frac{M}{w_{min}} \rceil$, which leads to the following.

Theorem 9.4.3

$$\Pi_M \leq \Phi_{m(T-1)+1} = O(6^{m(T-1)+1}) \tag{9.7}$$

where Φ_n denotes the number of alphabetic trees on n leaf nodes without any restriction on the arity of the nodes.

Proof Since $m = \lceil \frac{M}{w_{min}} \rceil$, any cluster pattern generated under size constraint M can not have more than m gates. If the arity of each gate is at most T, the maximum number of leaf node that an m node tree may have is $m(T-1)+1$. From Lemma 9.4.1, each cluster pattern corresponds to an alphabetic tree. Thus, in the worst case, the number of cluster patterns can never be more than $\Phi_{m(T-1)+1}$. It has been shown [111] (see Chapter 3 for further details) that $\Phi_{m(T-1)+1} = O(6^{m(T-1)+1})$. ■

This implies that the number of cluster patterns that need to be enumerated is independent of the circuit size and is exponential only in m and T which are usually fixed to be a small number. However, in the case where clusters of larger size are required, by using a hierarchical clustering approach, the number of cluster patterns can still be kept low. This and other extensions are discussed in Section 9.7.

9.4.4 Cluster Pattern Enumeration

Since different cluster patterns of size M are enumerated based on m which is the maximum number of gates that can be contained in a cluster of size M, it is likely that an infeasible cluster (i.e., a cluster with the actual size greater than M) is returned by the procedures enumerating clusters. However, enumerating based on m guarantees that all feasible clusters are enumerated. In the actual implementation, infeasible clusters can be easily avoided by keeping track of current size of clusters during the cluster pattern enumeration algorithm. Thus, the value of m has significance only in terms of deriving an upper bound on the runtime of the proposed algorithm.

Given the cluster size m in terms of the number of gates in a cluster, two routines are proposed to enumerate all cluster patterns rooted at a gate. The first routine, namely, *first_leaf_set* returns the leaf set corresponding to the first cluster pattern at a gate with a given size. Given the current leaf set, routine *next_leaf_set* returns the next leaf set of the same size. The topological order amongst the cluster pattern is formally defined as follows.

Let P_i and P'_i correspond to two cluster patterns of the same size m at gate i. Let i_1 through i_I correspond to the inputs of i ordered from left to right. The part of cluster P_i rooted at the inputs of gate i is denoted by P_{i,i_1} through P_{i,i_I}, and let W_1 through W_I correspond to the size of each such partial cluster, respectively. I refer to the set of these sizes as $\mathcal{W}_{1,I}$. I define $\mathcal{W}_{j,I} < \mathcal{W}'_{j,I}$ iff:

- $W_j < W'_j$ or
- $\mathcal{W}_{j+1,I} < \mathcal{W}'_{j+1,I}$

Then, I define $P_i > P'_i$ iff:

- $\mathcal{W}_{1,I} < \mathcal{W}'_{1,I}$ or
- $\mathcal{W}_{1,I} = \mathcal{W}'_{1,I}$ and $P_{i,i_{j-1}} = P'_{i,i_{j-1}} \forall j < k$ and $P_{i,i_k} > P'_{i,i_k}$ for any $k \in [1, I]$.

This concept of cluster pattern ordering is illustrated in Figure 9.3 and 9.5. Specifically, Figure 9.3 shows the order between the clusters and Figure 9.5 illustrates the corresponding tree structures. Based on the above order definition, cluster patterns are enumerated starting from the first cluster pattern to the last cluster pattern by routine *first_leaf_set* and routine *next_leaf_set*. I provide efficient implementations of these routines are as follows.

In these routines I use "{ }" as a set forming operator. Procedure "max_size(gate)" returns the number of gates in the transitive fanin which is computed in a preprocessing step. It should be noted that these routines only give a general outline of the actual routines. Certain special considerations, e.g., handling of DAG circuits, are not included in these routines.

```
Routine 1 first_leaf_set (gate, children, size)
begin
01  if (size == 1) return { gate }
02  else (size = size - 1)
03  currentSet = NULL
04  forEach child starting from the right most of children
05    children = children - { child }
06    childSize = min(max_size({ child }), size)
04    grandChildren = get_children(child)
07    currentSet = currentSet ∪ first_leaf_set (child, grandChildren, childSize)
08    size = size - childSize
09    if (size == 0)
10      currentSet = currentSet ∪ children
11      break;
12  return currentSet
end
```

The routine *first_leaf_set* generates the first topological pattern by generating a tree which is as much right-biased as possible. This is achieved by including as many gates as possible starting from the branch that is farthest right and including as many gates as possible before including the sibling to the left of the right most branch.

```
Routine 2 next_leaf_set (gate, children, leafSet, size)
begin
01  if ((size == 1)||(size == 0)) return NULL
02  if (|children| == 1)
03    gate = left_most(children)
04    children = get_children(gate)
05    return next_leaf_set(gate, children, leafSet, size - 1)
06  leftMostChild = left_most(children)
07  children = children - {leftMostChild}
08  leftLeafSet = left_leaf_set(leafSet)
09  rightLeafSet = leafSet - leftLeafSet
10  newRightSet = rightLeafSet
11  if (size(rightLeafSet) > 0)
12    newRightSet = next_leaf_set(gate, children,
            rightLeafSet, size(rightLeafSet))
13    if (newRightSet ≠ NULL) return leftLeafSet ∪ newRightSet
14    newRightSet = first_leaf_set(gate, children, size(rightLeafSet))
15  if (size(leftLeafSet) > 0)
16    newLeftSet = next_leaf_set(gate, {leftMostChild},
            leftLeafSet, size(leftLeafSet))
17    if (newLeftSet ≠ NULL) return newLeftSet ∪ newRightSet
17  if (size(rightLeafSet) == 0) return NULL
18  size(rightLeafSet) = size(rightLeafSet) −1
19  size(leftLeafSet) = size(leftLeafSet) +1
20  if (max_size(leftMostChild) ¡ size(leftLeafSet)) return NULL
21  if (size(rightLeafSet) > 0)
22    newRightSet = first_leaf_set(gate, children, size(rightLeafSet))
23  else newRightSet = NULL
24  newLeftSet = first_leaf_set(gate, {leftMostChild}, size(leftLeafSet))
25  return newLeftSet ∪ newRightSet
end
```

The routine *next_leaf_set* identifies leaf set associated with the next topological cluster pattern by recursively traversing to the right most gate j whose part of the original cluster P_i, namely $P_{i,j}$ can be restructured by removing a gate from one branch to add another gate to a branch on the left of it. In essence, this routine incrementally turns a right-biased clustering pattern to a left-biased clustering pattern.

Next, I show that this enumeration along with proper characterization of power is sufficient for power optimality of the clustering solution.

9.4.5 Optimality of POWER_CLUSTER

Theorem 9.4.4 *For tree structures,* POWER_CLUSTER *generates delay and power optimal clustering solution under the generalized load independent delay model.*

Proof I show this by induction on the original depth (i.e., depth before clustering). The algorithm generates a power and delay optimal solution at original depth 1 as the only solution at depth 1 is to cluster each gate individually. Assume that all the solutions for original depths 1 to k are delay and power optimal. Now, to prove the theorem it needs to be shown that the solution generated by POWER_CLUSTER for a gate i at original circuit depth $k+1$ is delay and power optimal.

Delay Optimality: Since the algorithm evaluates every feasible cluster pattern under the size constraint M, it identifies every cluster pattern that leads to the optimal arrival time value at gate i.

Power Optimality: To guarantee delay and power optimality, POWER_CLUSTER chooses the minimal power solution from the solutions that result in optimal delay at gate i. Thus, to show the power and delay optimality of the algorithm, I need to show that: 1) no other solution with less power for the same delay exists; and 2) a selection based on the power cost of each cluster pattern minimizes the power contribution of the clustering solution at gate i to the power dissipation of the entire circuit.

The first is true based on the assumption that clustering solution at each gate in the transitive fanin cone is delay and power optimal and the fact that all

feasible cluster patterns at gate i are enumerated. 2) is true due to the tree structure of the circuits. By selecting power optimal cluster pattern from all cluster patterns at gate i, POWER_CLUSTER guarantees the power optimality of the sub-circuit rooted at gate i. However, for tree structures, power cost of the sub-circuit rooted at i also corresponds to the power contribution of the sub-circuit to the entire circuit.

Thus, POWER_CLUSTER generates delay and power optimal circuits at gates with original depth $k+1$. ■

In should be noted that if area cost is measured in terms of minimal gate duplication, POWER_CLUSTER also generates area optimal circuits as any clustering on tree structures causes no gate duplication.

Taking into account the maximum number of cluster patterns the final time complexity of POWER_CLUSTER is given by $O(|\mathcal{G}|^2 \cdot (Dm^2 + mD^2) \cdot number_of_patterns) = O(|\mathcal{G}|^2 \cdot (Dm^2 + mD^2) \cdot 6^{m(T-1)+1})$. This complexity may appear prohibitive but it should be noted that the runtime is exponential only in terms of the cluster size and not in terms of the circuit size. By keeping the size of each cluster small relative to the size of individual gate sizes, the runtime penalty can be kept to a minimum. In cases where clusters of larger size are required, the clustering algorithm can be applied in a hierarchical fashion to achieve the larger size as described in Section 9.7

9.5 Low Power Partitioning for DAGs

To be able to handle general combinational circuits (i.e., DAGs) correctly, a few modifications must be made to POWER_CLUSTER.

Cluster Pattern Enumeration: Consider a gate i which fans out to gates x and y that are input to another gate z. Due to the re-convergent fanout at gate z, when enumerating cluster patterns at z, gate i will be encountered in two subtrees of z, namely, the subtree rooted at x and the subtree rooted at y. This is illustrated in Figure 9.6. To ensure correct cluster pattern enumeration in this situation, a flag is introduced at each gate which guarantees that a gate is

included in at most one of the subtrees of the root gate. The number of cluster patterns is still upper-bounded by equation (9.7) as described in Section 9.4.3.

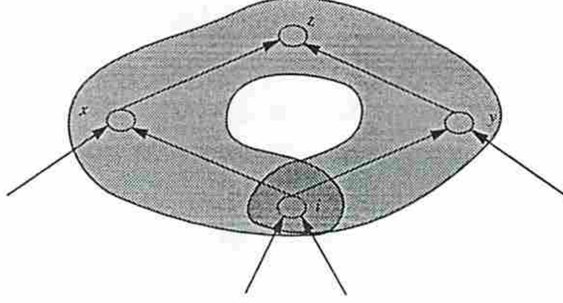


Figure 9.6: Multiple inclusion of a gate in a cluster due to a reconvergent fanout.

Power Calculation: To ensure correctness of power cost calculation for each arrival time, it must be guaranteed that no cluster in the sub-circuit rooted at a gate is counted more than once. This is achieved by extending the definition of PD-point to include one more member, i.e., by extending the definition of jth PD-point of gate i to a 4-tuple, namely, $\{a_i{}^j, p_i{}^j, \mathcal{LP}_i{}^j, \mathcal{M}_i{}^j\}$. The additional element $\mathcal{M}_i{}^j$ is referred to as the **membership set**, which is the set of PD-points for all clusters that are in the transitive fanin cone of gate i. The membership set for a PD-point is generated by merging membership sets of the corresponding PD-points at the leaf gates such that at most one PD-point for each gate in the circuit exist in the membership set. When two membership sets contain one PD-point each for a gate k, to ensure delay optimality, the PD-point with lower delay value is chosen. Using the membership set, the power cost of a PD-point at gate i is now calculated as follows:

$$p_i = N_i + \sum_{l \in \mathcal{M}_i} N_l \tag{9.8}$$

Or in the case where the internal power dissipation is also important:

$$p_i = N_i * C_i{}^{extra} + \sum_{j \in \mathcal{IN}_i} N_j * C_{j,i}{}^{basic} + \sum_{l \in \mathcal{M}_i} \{N_l * C_l{}^{extra} + \sum_{m \in \mathcal{IN}_l} N_m * C_{m,l}{}^{basic}\} \tag{9.9}$$

Thus, the membership set identifies all the clusters in the sub-circuit rooted at a gate. Given these clusters, it is possible to identify gates belonging to

each cluster to capture area cost of the current clustering solution. Hence, apart from capturing the exact power dissipation of the circuit, the concept of membership set can also be used to capture exact values of gate replication corresponding to each clustering solution. Thus, using the membership set, gate replication can be minimized directly during clustering.

For DAGs, the claim of optimality of POWER_CLUSTER does not hold as the principle of dynamic programming (optimal solutions of subproblems are sufficient to obtain optimal solution to the problem) is no longer valid due to logic sharing caused by multiple fanout gates. For example, if gates x and y are in the transitive fanout of gate i, it may not be possible to make correct decisions for power and area optimality at gates x and y. Specifically, since the sub-circuit rooted at gate i is shared by the sub-circuits rooted at x and y, it is likely that a sub-optimal solution in terms of individual power costs of x and y may be optimal when the combined power cost of x and y is considered. This implies that, to guarantee optimality, all clustering solutions at a gate (not only the non-inferior solutions) must be maintained, which grows exponentially with the circuit size. However, the delay optimality of POWER_CLUSTER under a load independent delay model remains valid.

Theorem 9.5.1 POWER_CLUSTER *generates a delay optimal clustering under a generalized load independent delay model.*

Proof The proof of the theorem follows from the observation that the only impact of generalization to DAGs from trees on the delay calculation is due to the effect of multiple fanout nodes. However, under a generalized load independent delay model, the delay is independent of fanout characteristics of gates. Hence, the claim for delay optimality of POWER_CLUSTER remains valid for general DAGs. ■

9.6 Post-Clustering Area Recovery

The gate replication introduced by POWER_CLUSTER can be recovered by employing area recovery mechanism that reduce gate replication at no loss in power dissipation. However, even greater reduction in gate replication can be obtained by allowing

some loss in power dissipation. In this section, three techniques for area recovery are described.

9.6.1 Area Recovery by Forcing a Single Clustering Solution

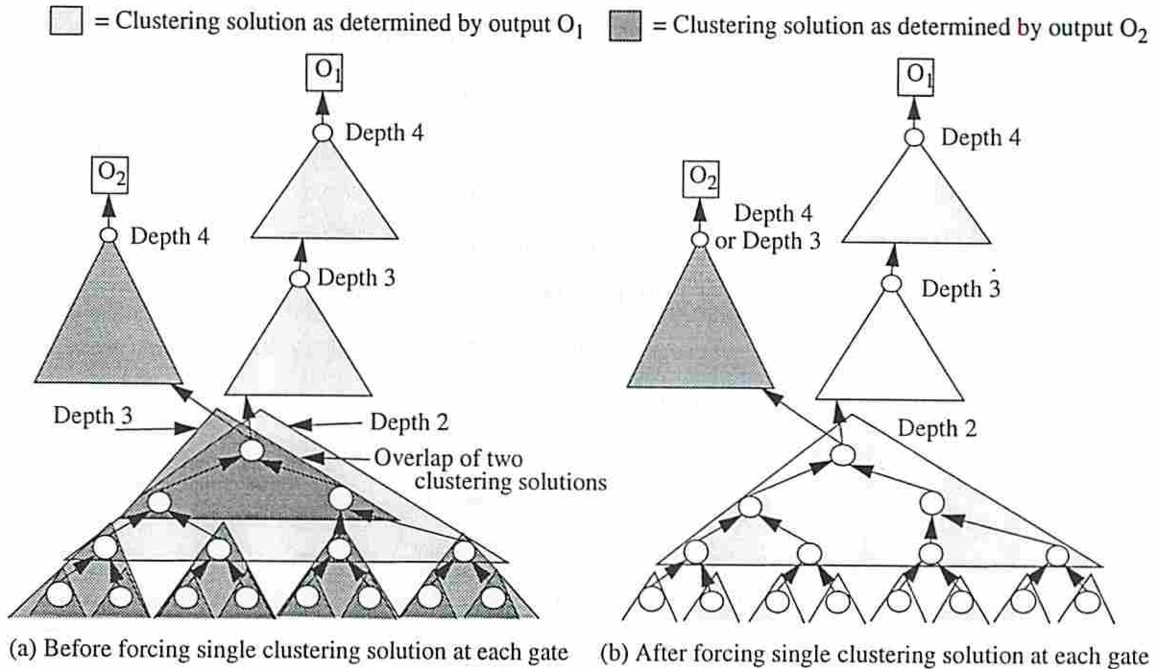


Figure 9.7: Area recovery during low power partitioning by forcing a single clustering solution.

As shown in Figure 9.7(a), due to multiple outputs with different criticality or due to reconvergent fanouts, it is likely that in the final solution more than one clustering solution is required at a gate. In such a situation, both area and power can be recovered without any loss in delay by always selecting the fastest clustering solution required at that gate as shown in Figure 9.7(b). In POWER_CLUSTER, the concept of membership set automatically ensures this type of area recovery by selecting only the best clustering solution at each gate.

9.6.2 Area Recovery by Relabeling Based on Unique Label Assignment

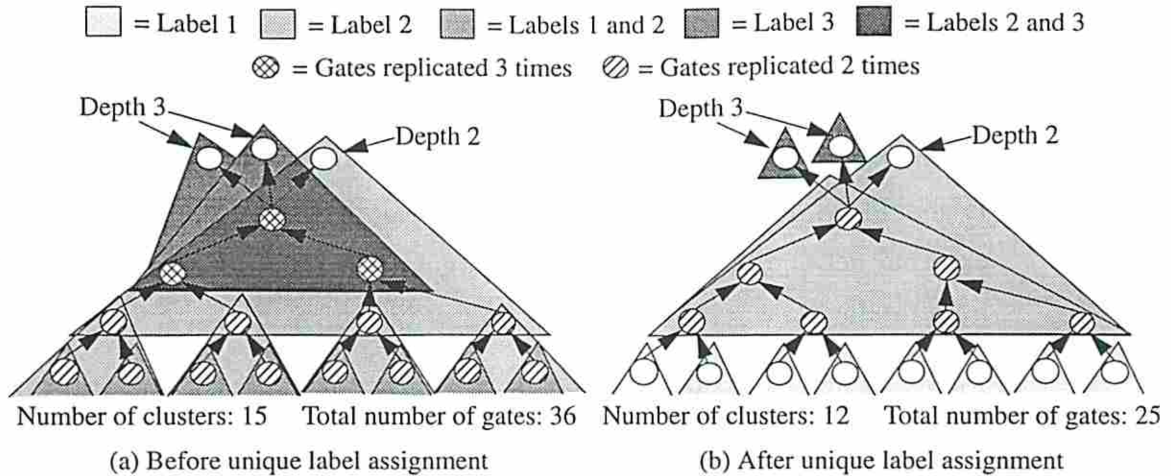


Figure 9.8: Area recovery during low power partitioning by unique label assignment.

An important characteristic of clustering algorithms based on Lawler's clustering algorithm is that each gate in the circuit has exactly one label. With POWER_CLUSTER, however, a gate can have multiple labels as shown in Figure 9.8(a). By forcing each gate to have only one label, the area of the clustered circuit can never get worse. However, the exact impact on the power dissipation needs to be analyzed. For each case where a gate has more than one label, by forcing the fastest label on that gate, a new cluster is introduced as shown in Figure 9.8(b). Since a new visible gate is introduced, the power dissipation of the circuit is likely to increase. However, by giving a unique label some power may also be saved in the following two cases:

- Since different clustering solutions corresponding to other labels for the root of new cluster are not required, the overall power dissipation may decrease.
- When C^{basic} can not be ignored compared to C^{extra}, such unique labeling may reduce the power dissipation due to reduced replication of internal gates.

Thus, for each gate that has multiple labels and has at least one fanout with each label, the impact on power dissipation and gate replication due to unique label assignment should be analyzed. Once this impact on power dissipation and gate replication is available, an informed decision can be made based on the power-area trade-offs desired. Based on the values of power gain or loss and area gain, the following three scenarios may exist.

1. There is no area gain. In this case, it can be shown that no power gain is possible and hence the labels of the gates should not be changed.

2. An area gain is accompanied either by a power gain or by no change in power. In this case, changing the gate label is desirable and hence should be performed. It should be noted that when there is no additional power dissipated on an external net (i.e., $C^{extra} = 0$), an area gain will always be accompanied by a power gain.

3. An area gain is accompanied by a loss in power. In this case, the decision must be made based on the relative importance of area and power costs.

Unfortunately, experimental results indicate that most of the area recovery achievable by this technique is accompanied by a loss in power when C^{basic} is negligible compared to C^{extra}. In this case, if power dissipation is of prime concern, this technique for area recovery should not be used.

9.6.3 Area Recovery by Relabeling Based on Required Times

Once all the gates in the circuit have been assigned unique labels, the relabeling technique proposed by Touati [104] can be applied to recover the area even further. This technique traverses the circuit in a preorder fashion from primary outputs to primary input (processing all the outputs of a gate before processing the gate itself) setting the required time at the gates based on the delay criticality of the gates. When two or more fanouts of a replicated gate have greater required time then the arrival time at the gate, the sub-cluster rooted at that gate can be implemented as separate cluster to reduce the replication. This is shown in Figure 9.9.

Once again, for each case of relabeling, the power cost/gain needs to be analyzed before relabeling the gates. Fortunately, in this case the power impact can be captured easily as it will depend only on the new cluster under consideration (and not on the transitive fanin cone of the cluster as is the case in relabeling by unique label assignment). Depending on the relative values of C^{extra} and C^{basic}, such relabeling may improve the power dissipation or degrade the power dissipation. However, if

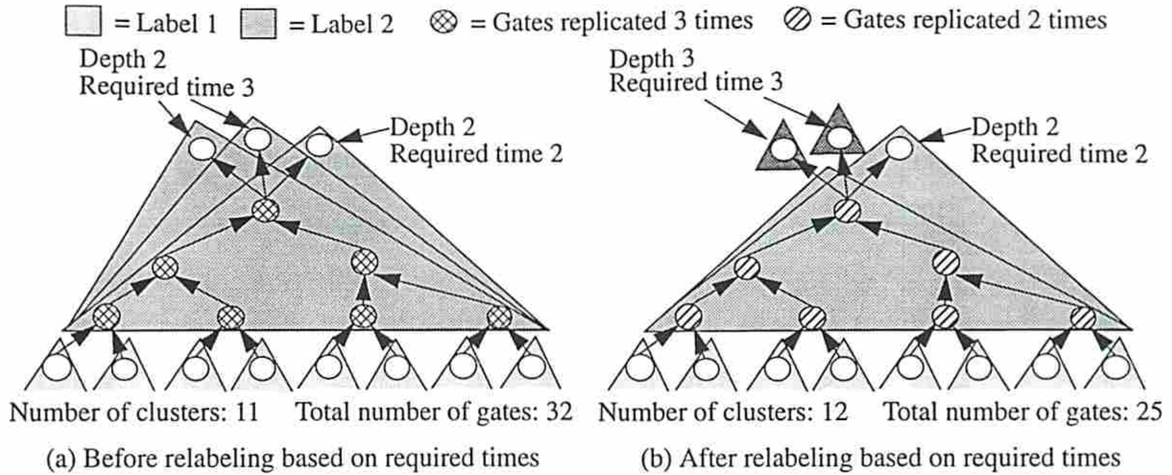


Figure 9.9: Area recovery during low power partitioning by relabeling based on required times.

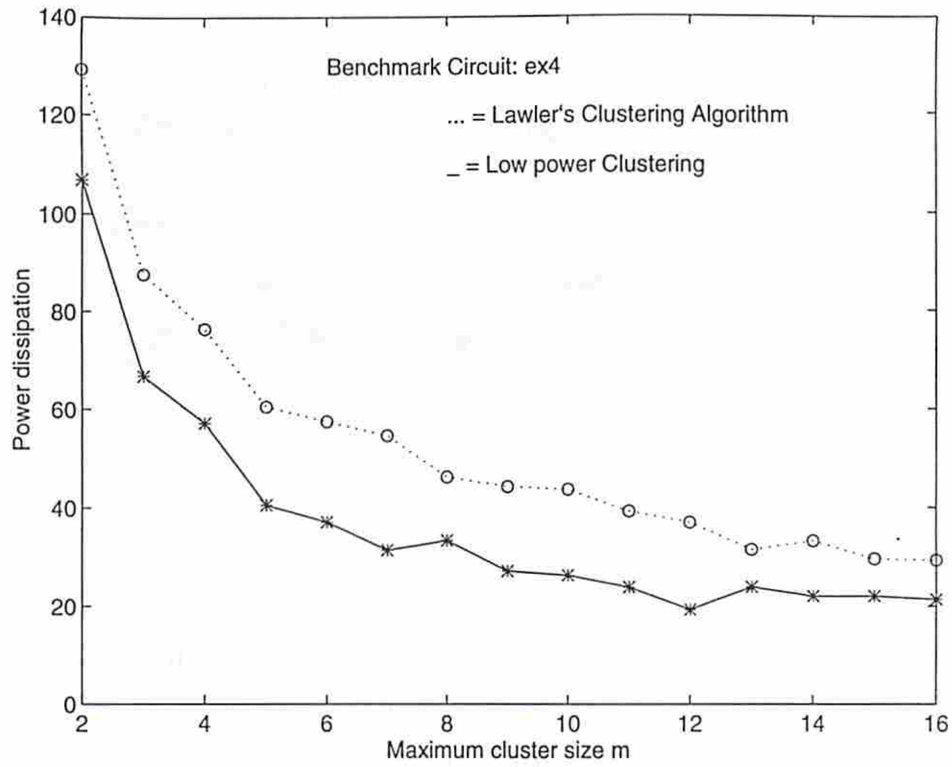
C'^{basic} is negligible compared to C'^{extra}, this technique will also always lead to increased power dissipation and hence should not be used. It should also be noted that this technique always increases the number of clusters.

9.7 Experimental Results and Extensions

I implemented POWER_CLUSTER in the SIS environment and compared my results with the SIS *reduce_depth* command that implements the original Lawler's algorithm [104]. Since *reduce_depth* requires that each gate in the original circuit be of the same size, it is recommended in [104] that the circuit be decomposed into two input NAND gates before clustering. Hence, for the sake of fair comparison, my algorithm also performs clustering on two input NAND gates. Thus, the inputs to both clustering algorithms are identical. The switching activities at each gate were computed using the symbolic simulation method [30] under a zero delay model. Since both algorithms generate delay optimal circuits, the delay values were identical for all results and hence are not reported.

9.7.1 Effect of Cluster Size on Power Dissipation

Considering that the maximum size of clusters m is very crucial to the run-time of the algorithm, an experiment was conducted first to determine the effect of increasing



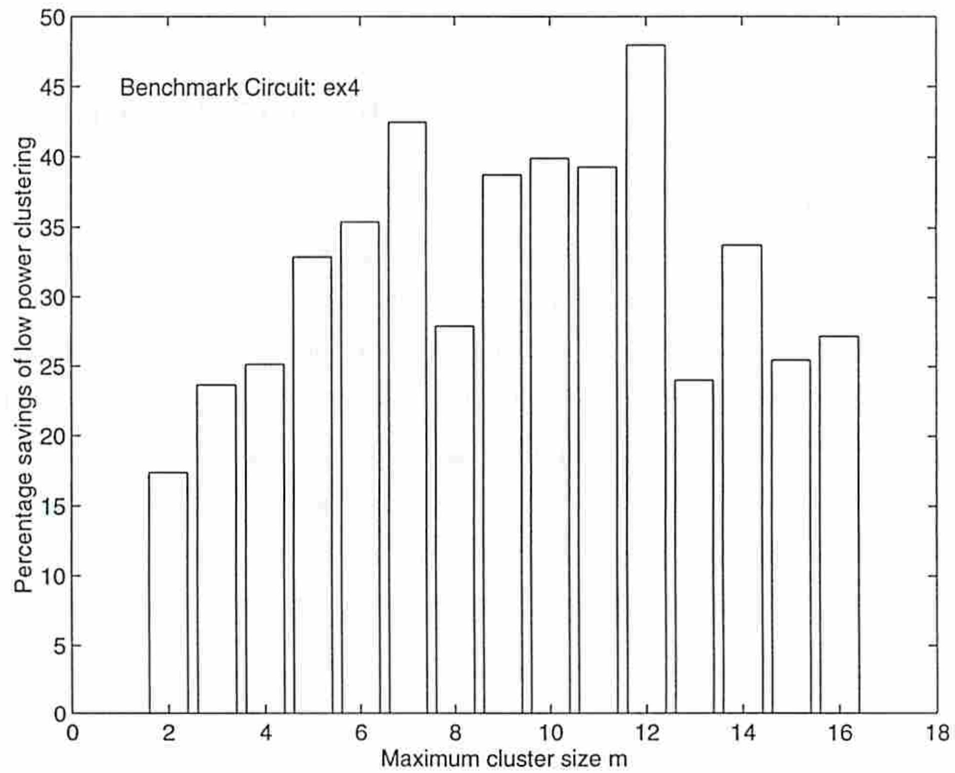


Figure 9.10: Effect of varying the maximum cluster size.

value of m on the power dissipation of the partitioned circuit. For benchmark circuit *ex4*, the value of m was varied from 2 to 16 and the power dissipation was calculated for both Lawler's algorithm and for POWER_CLUSTER. The power dissipation results are reported the first chart in Figure 9.10. As can be seen from these charts, the power dissipation of the clustered circuit for POWER_CLUSTER is significantly less than the power dissipation of circuits generated by Lawler's algorithm for all cluster sizes. Another important observation from these charts is that the power dissipation value drops dramatically with an increase in size until the size of about 6. Beyond size 6, the power dissipation decreases only slightly with an increase in size. The chart also shows that percentage improvement of POWER_CLUSTER over Lawler's algorithm in terms of power dissipation is maximized in the range 6-12.

This empirical study indicates that if clusters of larger size are required, and if the runtime of POWER_CLUSTER for such large cluster sizes is found to be unacceptable, then it is reasonable to perform the clustering hierarchically. Thus, if the required cluster size is z, clustering for low power can first be performed with $m = x$ and then the resultant circuit can be further clustered for low power with $m = y$ where $x \cdot y = z$. For example, if the required value of cluster size is 100, algorithm POWER_CLUSTER can be applied twice with $m = 10$. It should be noted that during the cluster enumeration, minimum cluster size constraints can be enforced easily, thereby avoiding a large number of small size clusters during this hierarchical approach. Furthermore, with such a hierarchical approach, only the external nets of the lower level of hierarchy will be used during clustering at the next level of hierarchy. When minimizing equation (9.3), this guarantees that power dissipation will monotonically decrease at each level of hierarchy, resulting in an effective performance-driven partitioning strategy for low power. Based on this analysis, for rest of the experiments a cluster size of 8 was selected.

9.7.2 Results without Area Recovery

First, I report results of POWER_CLUSTER while minimizing power dissipation under the simplified power model of equation (9.3). In Table 9.1, the results of applying Lawler's clustering algorithm on some two-level and multi-level benchmark circuits are presented. Rest of the results presented in this section are normalized with

Circuit	No. of Clusters	Gate Repl. Ratio	Power Dissipation
Two-level Benchmarks			
b12	26	1.14	9.71
cordic	31	1.45	10.58
cps	631	1.21	140.45
duke2	201	1.28	44.44
ex1010	845	1.77	163.92
ex4	126	1.12	46.25
misex2	40	1.30	5.27
misex3c	211	1.22	60.42
pdc	624	1.37	161.26
rd84	58	1.84	18.40
spla	464	1.40	114.59
Multi-level Benchmarks			
C1355	116	1.18	53.68
C2670	312	1.50	119.76
C432	104	1.53	36.57
C499	136	1.76	55.50
C880	153	1.71	48.72
apex6	250	1.52	75.00
apex7	105	1.64	35.18
b9	38	1.25	11.35
dalu	547	1.71	123.92
des	1396	1.93	366.04
k2	632	1.58	33.71
rot	278	1.48	100.61
t481	572	2.20	18.86

Table 9.1: Results using Lawler's clustering algorithm.

respect to the values in Table 9.1. For this experiment, the maximum cluster size was set to be 8, i.e., $m = 8$. However, it was observed that the results show improvements for other values of m as well. Gate replication ratio reported is the ratio of gate count after clustering to the gate count before clustering. The main reason for choosing cluster size of 8 was to control the run time of power clustering algorithm. As mentioned above, if clusters of larger size are required, then the clustering algorithm can be applied in a hierarchical fashion.

As can be seen from Table 9.2, on average, for two-level benchmark circuits (first set of circuits in Table 9.2), an improvement of about 41% improvement in power dissipation and about 24% in cluster count was obtained. However, the gate replication ratio increased by an average of 11%. This degradation was expected as for this set of results, POWER_CLUSTER does not apply any post-clustering area recovery heuristics while *reduce_depth* does use these heuristics. The reduction in the cluster count is a byproduct of power minimization during clustering, i.e., minimizing power not only amounts to selecting low switching activity external nets, but also to minimizing the number of external nets, thereby reducing the cluster count. For the multi-level benchmark circuits, however, the power dissipation is reduced by 29% but the gate replication has increased by 23%. The higher rate of gate replication can be attributed to high degree of reconvergent fanouts in these circuits. Since reconvergent fanouts are the cause of the area and power non-optimality of POWER_CLUSTER, and since for these results, POWER_CLUSTER is targeting only on minimizing the power, it is quite natural that the area cost is higher on circuits with high reconvergent fanouts. For all circuits, on average, the power dissipation was improved by 35% and the number of clusters was improved by about 19%. The gate replication ratio increased by an average of 18%.

9.7.3 Results with Area Recovery

In the cases where area cost is also of concern, area can be recovered as mentioned in Section 9.6. Since, the area recovery by forcing single solution at each gate is an integral part of POWER_CLUSTER, to achieve further area recovery, we need to apply the relabeling techniques described in Section 9.6. As mentioned in Section 9.6, these techniques are accompanied by an increase in power dissipation and cluster count

Circuit	No. of Clusters	Gate Repl. Ratio	Power Dissipation
b12	0.77	1.09	0.61
cordic	0.90	1.01	0.82
cps	0.72	1.30	0.45
duke2	0.68	1.08	0.44
ex1010	0.89	1.02	0.75
ex4	0.77	1.00	0.72
misex2	0.68	1.03	0.43
misex3c	0.62	1.09	0.46
pdc	0.75	1.32	0.54
rd84	0.83	1.10	0.79
spla	0.75	1.21	0.52
Two level average	0.76	1.11	0.59
C1355	0.86	1.15	0.82
C2670	0.93	1.34	0.81
C432	0.59	1.24	0.51
C499	0.75	0.99	0.79
C880	0.84	1.36	0.73
apex6	0.74	1.08	0.70
apex7	0.77	1.15	0.62
b9	0.82	1.14	0.71
dalu	0.98	1.28	0.73
des	0.95	1.26	0.85
k2	1.00	1.55	0.72
rot	0.74	1.34	0.69
t481	1.15	1.12	0.53
Multi level average	0.86	1.23	0.71
Average for all	0.81	1.18	0.65

Table 9.2: Results using POWER_CLUSTER without area recovery (normalized by corresponding values in Table 9.1).

in most cases. The results obtained by integrating the area recovery mechanisms in POWER_CLUSTER are shown in Table 9.3. As can be seen, by applying relabeling techniques, the replication ratio was in fact improved by 6% while improving the power dissipation by 11%

9.7.4 Results Considering Internal Power Dissipation

As mentioned earlier, algorithm POWER_CLUSTER has the capability of performing low power clustering even in the cases when the internal power dissipation (i.e., value of C^{basic}) is not negligible as compared to the extra power dissipation between clusters (i.e., value of C^{extra}). To show the effectiveness of POWER_CLUSTER in such a case, I report the results obtained for the other extreme case, i.e., when C^{extra} is negligible ($C^{extra} = 0$). It is interesting to note that just by changing the relative importance of C^{extra} with respect to C^{basic} we get quite a different problem. Specifically, in this case, since power dissipation of a net does not differ whether it is an internal net or an external net, the problem is to achieve delay optimality by replicating as many low power consuming gates as possible and by avoiding replication of high power consuming gates. The results obtained in this case is shown in Table 9.4. As can be seen on average, an improvement of about 11% in total power dissipation was obtained, at the same time improving the gate area by 9%.

9.8 Conclusion: Delay Optimal Circuit Partitioning for Low Power

In this chapter, a delay optimal clustering algorithm for low power was proposed. The proposed algorithm is delay and power optimal for tree structures whereas it is delay optimal for general DAGs under a generalized load independent delay model. The optimality claims for this algorithm stem from the underlying clustering mechanism that enumerates all feasible cluster patterns at each gate in the circuit and maintains only the power optimal solutions at each gate for each arrival time value. In the process, it has been shown that enumeration of cluster patterns relates to enumeration of alphabetic trees and provided upper-bounds on the number of

Circuit	No. of Clusters	Gate Repl. Ratio	Power Dissipation
b12	0.92	0.96	0.77
cordic	1.06	0.98	1.10
cps	0.96	1.01	0.81
duke2	0.92	0.98	0.82
ex1010	1.06	0.93	1.09
ex4	0.92	0.98	0.89
misex2	0.82	0.98	0.76
misex3c	0.93	0.99	0.85
pdc	0.99	0.98	0.88
rd84	0.97	1.02	0.95
spla	0.98	0.98	0.86
Two level average	0.96	0.98	0.89
C1355	0.86	1.08	0.82
C2670	0.84	0.87	0.83
C432	0.74	0.98	0.74
C499	0.71	0.82	0.77
C880	0.83	0.94	0.81
apex6	0.82	0.95	0.79
apex7	0.82	0.94	0.72
b9	0.97	1.03	0.90
dalu	0.87	0.76	0.79
des	0.91	0.98	0.87
k2	1.12	0.96	1.18
rot	0.81	0.90	0.82
t481	1.30	0.96	1.61
Multi level average	0.89	0.94	0.90
Average for all	0.92	0.96	0.89

Table 9.3: Results using POWER_CLUSTER with area recovery (normalized by corresponding values in Table 9.1).

Circuit	No. of Clusters	Gate Repl. Ratio	Power Dissipation
b12	1.08	0.95	0.95
cordic	1.19	0.86	0.85
cps	1.01	0.98	0.98
duke2	1.05	0.99	0.98
ex1010	1.05	0.90	0.89
ex4	1.07	0.99	0.99
misex2	1.07	1.00	1.00
misex3c	1.08	0.94	0.94
pdc	0.97	0.90	0.88
rd84	1.00	0.92	0.91
spla	1.00	0.90	0.88
Two level average	1.05	0.94	0.93
C1355	1.00	1.00	1.00
C2670	0.90	0.80	0.79
C432	0.92	1.06	1.07
C499	0.84	0.79	0.74
C880	0.92	0.77	0.79
apex6	0.99	0.95	0.95
apex7	1.01	0.94	0.94
b9	1.13	1.03	1.02
dalu	0.91	0.79	0.83
des	0.96	0.83	0.80
k2	1.28	1.04	0.89
rot	0.95	0.84	0.83
t481	0.94	0.64	0.38
Multi level average	0.98	0.88	0.85
Average for all	1.01	0.91	0.89

Table 9.4: Results using POWER_CLUSTER minimizing total power dissipation (normalized by corresponding values obtained with Lawler's clustering algorithm).

cluster pattern that need to be identified at each gate. Indeed, it is shown that if the cluster size and the maximum number of inputs to any gate are bounded by a constant number, the runtime of the proposed algorithm is polynomial in the circuit size. Also, formal methods were proposed to enumerate all cluster patterns efficiently, specifically with respect to non-binary tree structures.

Experimental results indicate that the proposed algorithm generates circuits with exactly the same delay as traditional delay optimal clustering mechanisms but at a substantial saving in power dissipation and number of clusters. Also, the results indicate that in the case where gate replication needs to be controlled, this algorithm can in fact produce circuits with better gate replication while improving the power dissipation.

The only disadvantage of this algorithm is the increased runtime which may be prohibitive in some cases. Though the time complexity is polynomial in the size of the circuit, it is exponential in the maximum size of the partition due to the pattern enumeration. To speed up the runtime, some quick method of identifying "good" patterns should be used. Indeed, an algorithm that needs to enumerate only $O(m^T)$ clusters at each gate based on the cluster size distribution at immediate fanins of gates can be developed. However, since switching activities at gates do not follow a predictable pattern in terms of circuit structure, any method that does not consider all cluster patterns is likely to be sub-optimal for DAG circuits.

Apart from the power minimization during clustering, this algorithm contributes to the general field of delay optimal clustering. Also, it has the capability of tracking the gate replication during clustering, which can be used to optimize area during clustering. In fact, experimental results indicate that when applied purely in an area driven mode, this algorithm can improve area by about 10% for the same delay and about the same cluster count.

Chapter 10

Standard Cell Placement and Routing for Low Power

This chapter presents PCUBE, a *P*erformance driven *P*lacement tool for low *P*ower. The problem is formulated as a constrained programming problem and is solved in two phases: global optimization and slot assignment. The objective function used during either phase is the total weighted net length where net weights are calculated as the expected switching activities of gates driving the nets. This chapter also presents my experiences with low power global routing and feed-through assignment for standard cell circuits. Since a minimal netlength routing topology also leads to a minimal power dissipation topology for a given net, the focus of a low power routing tool must be on trading off lengths of nets driven by low switching activity gates to reduce the lengths of nets driven by high switching activity gates. I attempt to achieve this by judiciously ordering the nets to be routed and by appropriately modifying the objective functions for certain subtasks of global routing.

The remainder of the chapter is organized as follows. Section 10.1 presents the mathematical formulation of PCUBE and outlines the solution technique under a zero delay model. In Section 10.1.3, I discuss modifications to the basic technique when using a real delay model. Experimental results and concluding remarks for PCUBE are presented in Section 10.1.4 and 10.1.5, respectively. In Section 10.2, I present my approaches for low power global routing. Section 10.2.1 contains an introduction to the problem of low power circuit routing followed by a discussion of the outcome of my experiments with low power routing tools developed. Section 10.2.2 presents the problem of feed-through assignment for low power and contains

a discussion of experimental results. Concluding remarks for low power routing and possible approaches for detailed routing are presented in Section 10.2.3.

10.1 PCUBE: Performance-Driven Placement for Low Power

The objective function for the placement of large scale cell-based ICs is either total wire length [51, 105, 55, 101, 97] or wire density [20]. The objective function for low power placement can be formulated in a similar manner as described next.

Let

$$\begin{aligned} \mathcal{I} &= \{i_1, \ldots, i_I\} \\ \mathcal{M} &= \{m_1, \ldots, m_M\} \\ \mathcal{O} &= \{o_1, \ldots, o_O\} \\ \mathcal{N} &= \{n_{i_1}, \ldots, n_{i_I}, n_{m_1}, \ldots, n_{m_M}\} \end{aligned} \tag{10.1}$$

denote the sets of primary inputs, internal gates, primary outputs, and nets, respectively. The total number of nets N is given by $I + M$. Each gate i has a 6-tuple $(N_i, C_i^{in}, a_i, r_i, C_i^{wire}, C_i^{gate})$ associated with it where $N_i, C_i^{in}, a_i, r_i, C_i^{wire}$, and C_i^{gate} denote the switching rate of gate i, input capacitance of gate i, arrival time at the output of gate i, required time at the output of gate i, wire load due to output net n_i, and gate load of n_i, respectively. Let us denote the set of gates connected by net n_i by ν_i. The switching activity N_i of i is computed under a zero delay model as described in Section 9.2. The primary inputs are assumed to be switching independently at probability of 0.5.

The total power consumption for the circuit is given by

$$P = 0.5 \frac{{V_{dd}}^2}{T_{cycle}} \sum_{i \in \{\mathcal{I}, \mathcal{M}\}} (C_i^{wire} + C_i^{gate}) N_i.$$

The term $\sum_i C_i^{gate} N_i$ is independent of the placement and hence, is dropped from the objective function. After dropping the constant multiplication factor $0.5 \frac{{V_{dd}}^2}{T_{cycle}}$, we get the following objective function for low power placement:

$$L_1 = \sum_{i \in \{\mathcal{I},\mathcal{M}\}} \{C_i^{wire} N_i\}. \tag{10.2}$$

Objective functions for minimizing the total wire length are either linear [51], quadratic [55, 101] or quadratic approximation of linear functions [97]. Linear objective functions, although provide a more accurate model of the wire length, suffer from excessive computation times. Quadratic objective functions do not model wire lengths as accurately, but allow efficient quadratic programming techniques to be applied.

A quadratic programming formulation is adopted for PCUBE, i.e., the objective function is the sum over all nets of the square of power consumption due to each net. As usual, net n_i is modeled by a weighted clique. Therefore, the quadratic objective function for low power placement is

$$L_2 = \sum_{i \in \{\mathcal{I},\mathcal{M}\}} \{(C_i^{wire})^2 N_i^2\} \tag{10.3}$$

which can be rewritten as

$$L_2(\mathrm{x},\mathrm{y}) \quad = \quad (C_{unit})^2 \sum_{i \in \{\mathcal{I},\mathcal{M}\}} \{N_i^2 \tfrac{2}{|\nu_i|} \sum_{j,k \in \nu_i} [(x_j - x_k)^2 + (y_j - y_k)^2]\} \tag{10.4}$$

or after dropping constants and rearranging:

$$L_2(\mathrm{x},\mathrm{y}) = \sum_{j,k \in \{\mathcal{I},\mathcal{M}\}} \{E_{jk}[(x_j - x_k)^2 + (y_j - y_k)^2]\}$$

where $E_{jk} = 0$ if no net contains both gates j and k and $E_{jk} = \sum_i N_i^2 \frac{2}{|\nu_i|}$ where i denotes any net that contains gates j and k.

The objective function can be formulated using the matrix notation as

$$L_2(\mathrm{x},\mathrm{y}) = 1/2(\mathrm{x}^T \mathrm{B}\mathrm{x} + \mathrm{y}^T \mathrm{B}\mathrm{y}) + \mathrm{c}^T \mathrm{x} + \mathrm{d}^T \mathrm{y} \tag{10.5}$$

where $\mathbf{x}$ and $\mathbf{y}$ are vectors of x and y co-ordinates of the gate locations. Matrix $\mathbf{B}$ is a symmetric matrix derived from matrix $\mathbf{E} = [E_{ij}]$ as follows:

$$\mathbf{B} = \mathbf{D} - \mathbf{E}$$

where $\mathbf{D}$ is a diagonal matrix with $d_{ij} = \sum_j E_{ij}$. This objective function is similar to the one derived in PROUD [105], GORDIAN [55] and RITUAL [101]. It was shown in [100] that as long as the whole circuit forms one connected set and some modules are fixed, $\mathbf{B}$ is positive definite. This implies that the objective function is convex and that quadratic optimization techniques can be applied to obtain a global optimal solution.

The approach followed for PCUBE is similar to the one used in GORDIAN and RITUAL. Quadratic optimization is combined with iterative partitioning of the circuit. After each global optimization step, circuit is further partitioned, and the partitioning information is introduced in the subsequent global quadratic optimization step as center of mass constraints. Unlike the divide-and-conquer mechanisms proposed in literature, this mechanism maintains a global view of the circuits beyond the partition boundaries and allows migration of gates across boundaries.

10.1.1 Global Optimization Phase

Standard cell placement requires that the gates be placed on rows. However, solution to equation (10.5) does not attempt to distribute gates so as to satisfy such slot constraints. To distribute gates across the chip, center of mass constraints are incrementally added for the gates assigned to each partition as follows.

During the first phase of global optimization, two center of mass constraints are introduced as follows:

$$\begin{aligned} 1/M \textstyle\sum_{i \in \mathcal{M}} x_i &= C^x \\ 1/M \textstyle\sum_{i \in \mathcal{M}} y_i &= C^y \end{aligned} \qquad (10.6)$$

where C^x and C^y are the co-ordinates of the center of the chip. These two constraints tend to evenly distribute the gates around the center of the chip.

Let P_j denote partition j, and K denote the total number of partitions at some intermediate step. Let $C_j{}^x$ and $C_j{}^y$ denote the coordinates of the center of partition P_j. The slot constraints generated for the next global optimization phase are

$$\begin{array}{l} \frac{1}{|P_j|}\sum_{i\in P_j} x_i = C_j{}^x j = 1,\ldots,K \\ \frac{1}{|P_j|}\sum_{i\in P_j} y_i = C_j{}^y j = 1,\ldots,K \end{array} \tag{10.7}$$

This partitioning strategy could be continued until only a single gate remains in each partition, completely resolving the slot constraints. However, it is often better to stop the global optimization phase when a small number of gates remain in each partition and continue with a slot assignment phase as described in the next subsection.

Performance constraints are provided in terms of required times at primary outputs and arrival times at primary inputs of the circuit as follows. Along with the center of mass constraints given by equation (10.7), the following timing constraints are integrated with the objective function given by equation (10.5).

$$\begin{array}{rcl} a_j & \geq & a_i + d_{n_i} \quad \{\forall i,j| \;\; j \in \nu_i\} \\ a_j & \leq & R_j \;\; \forall j \in \mathcal{O} \\ a_j & \geq & A_j \;\; \forall j \in \mathcal{I} \\ d_{n_i} & = & f(X_i, Y_i) \;\; \forall n_i \in \mathcal{N} \end{array} \tag{10.8}$$

where R_j and A_j are the specified required times and arrival times, respectively. The constraints $a_j \geq a_i + d_{n_i} \;\; \{\forall i,j| \;\; j \in \nu_i\}$ and $d_{n_i} = f(X_i, Y_i) \;\; \forall n_i \in \mathcal{N}$ are introduced to update the delay information. Here, $d_{n_i} = f(X_i, Y_i)$ computes the delay through net n_i depending on positions of the gates in ν_{n_i}. To compute d_{n_i}, the *star* net model was used. It was shown in [100] that the interconnect delay based on the star net model is a convex function of gate positions.

Thus, I have a convex objective function with a set of convex constraints. The resulting constrained optimization problem can be formulated as a Lagrangian function optimization and solved efficiently using Lagrangian relaxation techniques [101, 96].

10.1.2 Slot Assignment Phase

The hierarchical partitioning and global optimization is performed iteratively until 5-10 gates remain in each partition. Assuming that each partition has at least as many slots as gates, the problem of assigning a gate to a slot reduces to a *Linear Assignment* problem. For each partition, a cost matrix with as many rows as the number of gates and as many columns as the number of slots is constructed. Each element C_{ij} in the cost matrix reflects the power cost if gate i is assigned to slot j and is given by

$$C_{ij} = \sum_{i \in \nu_k} N_k \frac{[steiner_approx(\nu_k, i, j)]}{|\nu_k|}.$$

where $steiner_approx(\nu_k, i, j)$ calculates a rectilinear single-trunk steiner approximation for net ν_k when gate i is assigned to slot j.

Iterations of slot assignment are interleaved with the shifting of partition boundaries. This allows migration of gates across partitions even during the slot assignment phase.

10.1.3 Placement for Low Power under a Real Delay Model

So far a zero delay model was assumed for calculating the switching activities. In reality, however, gates and nets have positive delays, giving rise to glitches. The transitions due to glitches occur mainly due to signals arriving at different times at the inputs of a gate. In general, glitching accounts for about 10-50% of dynamic power consumption in a circuit.

Let the switching activity of gate i under a real delay model be denoted by N_i^{real} and under a zero delay model by N_i^{zero}. The glitching activity of the gate N_i^{glitch} is then given by $N_i^{glitch} = N_i^{real} - N_i^{zero}$. Under a real delay model, equation (10.3) can be rewritten as

$$L_2 \quad = \quad \textstyle\sum_{i \in \{\mathcal{I}, \mathcal{M}\}} \{(C_i^{wire})^2 (N_i^{real})^2\}. \qquad (10.9)$$

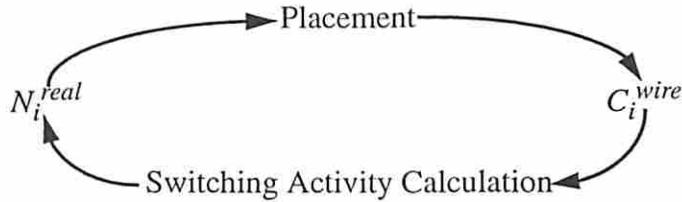


Figure 10.1: The dependency loop involved in minimizing power dissipation under a real delay model during placement.

The switching activity due to glitches, that is, N_i^{glitch} depends on the real delay information, which in turn depends on C_i^{wire} derived from exact netlengths. However, since the netlengths are being modified during placement, it can not be assured that N_i^{real} remains constant during placement. This situation is illustrated in Figure 10.1.

The exact solution to this problem - which requires a complete integration of power analysis and placement - seems to be intractable. Hence, I propose a simple approximate solution by updating switching activities during placement. Updating the switching activities after each iteration of Lagrangian relaxation is one possible mechanism, but is very CPU intensive and will possibly result in convergence problems. The current implementation therefore updates the switching activities only after each partitioning step.

10.1.4 Experimental Results and Discussions

All circuits were first optimized in SIS using the rugged script [92]. The circuits were then mapped using the SIS mapper in timing mode [103]. Switching activities were calculated using the approach of [30] assuming a zero delay model. The circuits were then placed using PCUBE and RITUAL. The power consumption (due to interconnect only) was estimated after placement using the steiner approximation net model and assuming a 20MHz clock.

In Table 10.1, the results of RITUAL are compared with results obtained with PCUBE. PCUBE has improved the power consumption of the circuit by about 7% at the cost of 8% increase in the wire length and 2% increase in the circuit delay.

Circuit	Ritual Placement delay ns	power μW	Wire length	Pcube delay	power μW	Wire length
9symml	27.17	1931	55.9	27.16	1759	52.4
C1355	32.93	5657	167.4	34.32	4904	200.4
C3540	84.04	14334	488.8	82.35	13312	565.1
C432	55.35	1831	60.5	55.85	1795	62.7
C880	57.79	4305	131.1	60.73	4259	141.7
apex6	35.00	10011	334.8	36.16	9388	338.0
apex7	19.77	2420	74.5	20.65	2342	76.5
dalu	96.50	11313	529.4	99.63	10364	615.1
misex3	43.74	5491	226.2	42.39	4724	258.0
rot	41.16	9623	295.9	41.35	9569	305.2
Average	100	100	100	101.6	93.5	107.8

Table 10.1: Results obtained with Ritual and low power placement tool Pcube.

Ckt.	Timing mode delay ns	power μW	Wire length	Normal mode delay ns	power μW	wire length
9symml	27.26	2148	63.2	27.16	1759	52.4
C1355	33.82	6009	177.3	34.32	4904	200.4
C3540	83.85	16308	566.1	82.35	13312	565.1
C432	53.57	2088	69.3	55.85	1795	62.7
C880	55.29	4704	143.3	60.73	4259	141.7
apex6	34.34	10883	362.0	36.16	9388	338.0
apex7	21.41	2720	84.9	20.65	2342	76.5
dalu	96.80	12742	574.2	99.63	10364	615.1
misex3	42.83	6006	246.6	42.39	4724	258.0
rot	40.17	9685	299.0	41.35	9569	305.2
Avg.	100	100	100	102	85.3	98.2

Table 10.2: Pcube: Timing Vs. Normal mode.

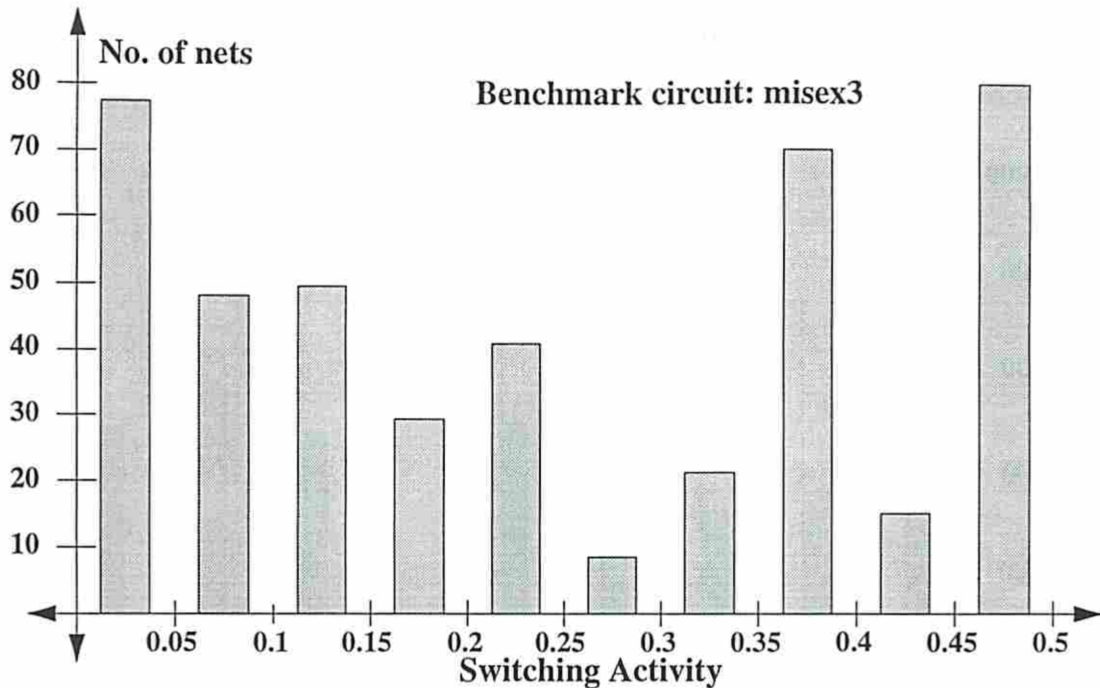


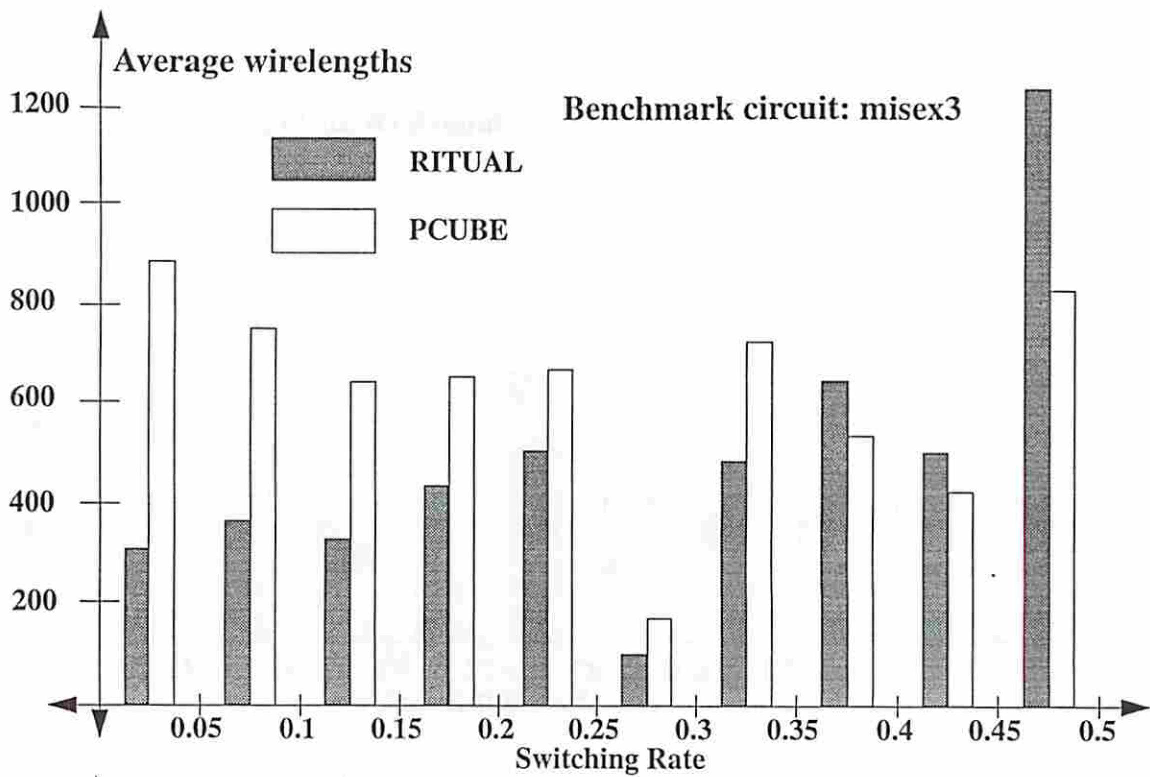
Figure 10.2: Switching activity profile for benchmark circuit *misex3*.

In Table 10.2, I compare PCUBE without timing constraints with PCUBE with timing constraints. PCUBE in normal mode degrades performance of the circuit by 2% while reducing the power consumption by about 15%.

Next, the effect of PCUBE on the switching activity profiles of a circuit is discussed. Figure 10.2 shows histogram representing the the distribution of switching activities in the circuit.

Figure 10.3 shows histograms generated from the results obtained by running PCUBE and RITUAL on the benchmark circuit *misex3*. The first histogram indicates the effect of PCUBE on the wire length distribution compared to RITUAL. It is clear from the histogram that PCUBE has reduced the average wiring load on nets with high switching activities at the cost of increasing the average wiring load on nets with low switching activities. The second histogram shows that the gain from the reduction in power consumption on nets with higher switching activities has more than compensated for the power consumption increase from nets with smaller switching activities, thus resulting in a net power reduction.

For a number of these circuits, the improvement in power consumption was not maintained after detailed routing. I believe this is due to three factors: the circuits



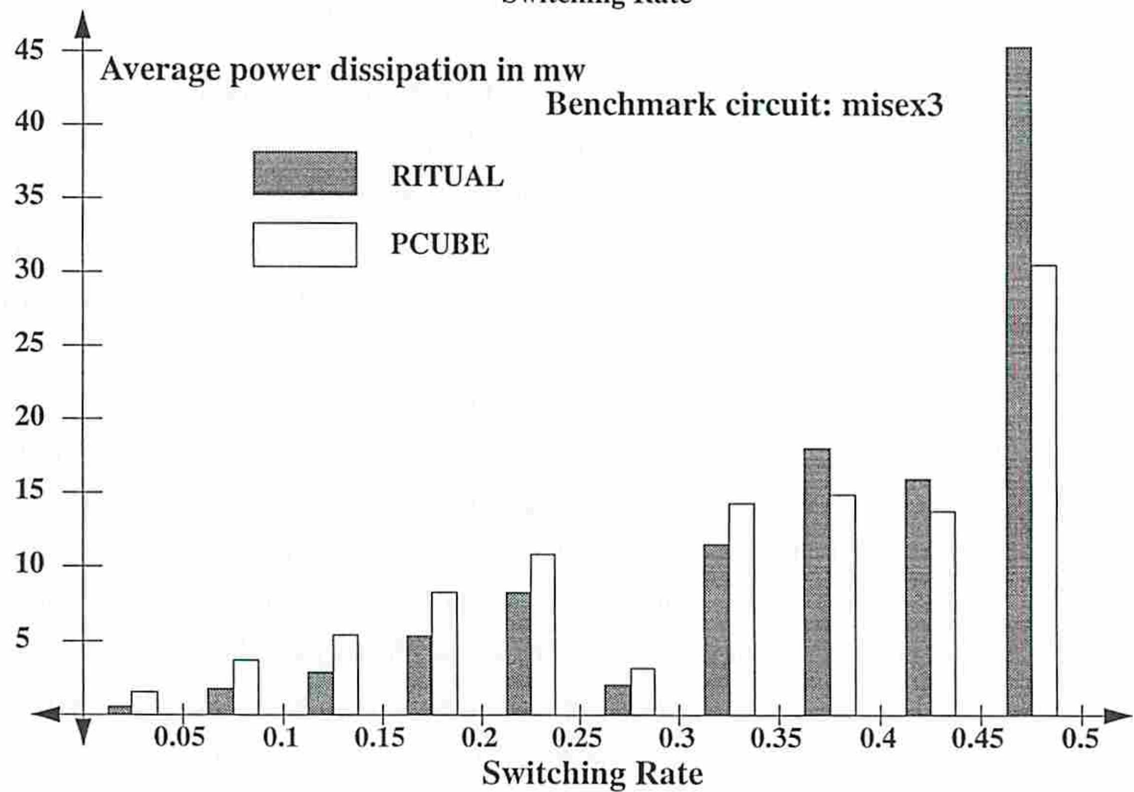


Figure 10.3: Wire Length and power distribution comparison between PCUBE and RITUAL.

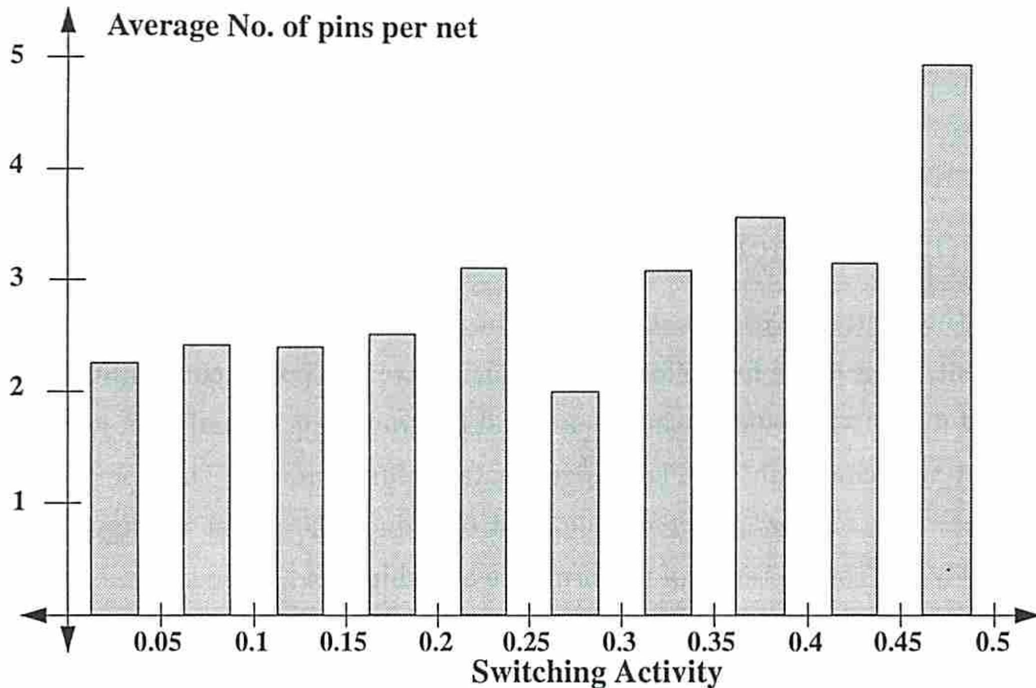


Figure 10.4: Net distribution profile for benchmark example *misex3*.

were not routed using a low power routing program; a star-connected net model was used during placement which overestimates wire lengths of larger nets; a quadratic (instead of a linear) objective function was used. The last two factors affect placement for wire length minimization as well, but their effect is accentuated for low power placement. This is because, as shown in the histogram of Figure 10.4, nets with higher switching activities also have higher number of pins per net. Consequently, the quadratic objective function and star net modeling in PCUBE overestimate power consumption in nets with higher switching activities. This suggests that linear objective function and better net modeling are desirable for placement for low power.

The current implementation of PCUBE under a real delay model does not improve the results obtained by PCUBE under a zero delay model. Hence, alternate approaches to minimize power dissipation under real delay model must be proposed.

A sufficient condition for the output of a gate to be glitch-free is that all its inputs are glitch-free and that they arrive at the same time. Hence, another approach for minimizing the power consumption due to glitches is to identify gates with high glitching power consumption - gates with high $N_i^{glitch}(C_i^{wire} + C_i^{gate})$ - and then

introduce equal arrival time requirements at inputs of these gates as new timing constraints. Alternatively, reduction in glitching-power can be achieved by inserting buffers on short paths during placement. This will result in an increase in the zero delay power consumption of the circuit, and hence should be carefully managed.

10.1.5 Conclusion: Placement for Low Power

In this section, I proposed a placement algorithm for low-power standard cell placement. This is the first effort at minimizing average power consumption during physical design. I provided a mathematical programming formulation with performance and slot constraints. The experimental results indicate that performance driven placement for low power can be used to reduce the power consumption with relatively low cost in terms of performance and chip area. I also showed that even within the power minimization framework, delay could be efficiently traded for power and vice versa by providing a placement algorithm that operates in both *power* and *time* modes.

10.2 Standard Cell Routing for Low Power

The main motivation behind proposing routing tools for low power is to provide a complete performance-driven power minimizing physical design tool, consisting of a partitioning tool, a placement tool, a global routing tool and a detailed routing tool for standard cell circuits. In this section, I describe my approaches for low power global routing and low power feed-through assignment for standard cell circuits. However, as described later in this section, my experience with low power circuit routing indicate that there is very little potential to improve power during global routing for standard cell circuits.

10.2.1 Low Power Global-Routing

The main task of the global router is to generate routing trees while taking into account channel and switch-box utilization so as to minimize the expected chip area after detailed routing. These routing trees are usually generated using a minimal

spanning/steiner tree algorithm along with some heuristic to estimate/minimize final chip area. In a standard cell environment, the main contribution of a global router is to identify the number of feed-throughs and their assignment to the nets such that the chip area is minimum. Minimization of chip area is achieved by minimizing the chip height by reducing some estimate of the total number of tracks and/or by minimizing the chip width by minimizing the number of feed-throughs in the longest rows. Once again, minimal spanning/steiner tree algorithms are used to achieve this along with different penalty values for insertion of a feed-through in different rows.

Considering that the problem of generating optimal steiner tree for each net which is shown to be NP-hard [29], is a subproblem of the global routing problem, it is not surprising that most version of global routing problems are also NP-hard (for references, check [65]). Hence, some sub-optimality is introduced inevitably by any non-optimal polynomial time heuristic for generating routing topology of each net. Apart from that, further sub-optimality is introduced by a standard cell based global routing tool as it can predict neither the number of tracks nor the number of feed-throughs required a-priori (except in some special cases, e.g., when there is a constraint to introduce absolutely minimum number of feed-throughs.). The main reason for this is that the number of tracks is determined only after detailed routing and the number of feed-throughs inserted is highly dependent on the order in which the net segments are routed.

There has been a lot of research on global routing. Most of this research focuses on improving the length of a net by proposing different heuristics to generate better routing trees [48, 40, 12, 53]. The underlying algorithm for these global routers is as follows.

```
Algorithm SeqRoute(G, N)
G is a set of placed gates
N is the set of nets connecting G
01 begin
02   Initialize feed-through penalties
03   ForEach net n ∈ N
04     V = set of gates connected by n
05     E = {edge(i, j)|i, j ∈ V}
06     F = NULL
07     C(e) = cost of edge e; ∀e ∈ E
08     do
09       Identify e with minimum C(e)
10       if F ∪ e does not have a cycle
11         F = F ∪ e
12         Update feed-through penalties and cost of remaining edges
13       E = E − e
14     While E ≠ NULL
15 end
```

As shown, the above algorithm implements Prim's minimum spanning tree algorithm to generate the routing tree. However, any of the other routing tree generation mechanism can be substituted in algorithm SeqRoute. Line 2 of the algorithm initializes the feed-through penalty[1] for each placement row that characterizes the additional cost an edge encounters when it crosses that row, i.e., cost of introducing a feed-through on that row. Different strategies can be employed to set these feed-through penalties. Typically, the longest placement row in the circuit is assigned the highest feed-through penalty so as to discourage use of a feed-through on that row. As seen on Line 12, after introduction of each edge to the net tree, the lengths and hence the feed-through penalty of some rows may change and hence needs to

[1] By feed-through penalty, I imply the penalty that is in addition to the netlength penalty caused by X and Y locations of corresponding pins.

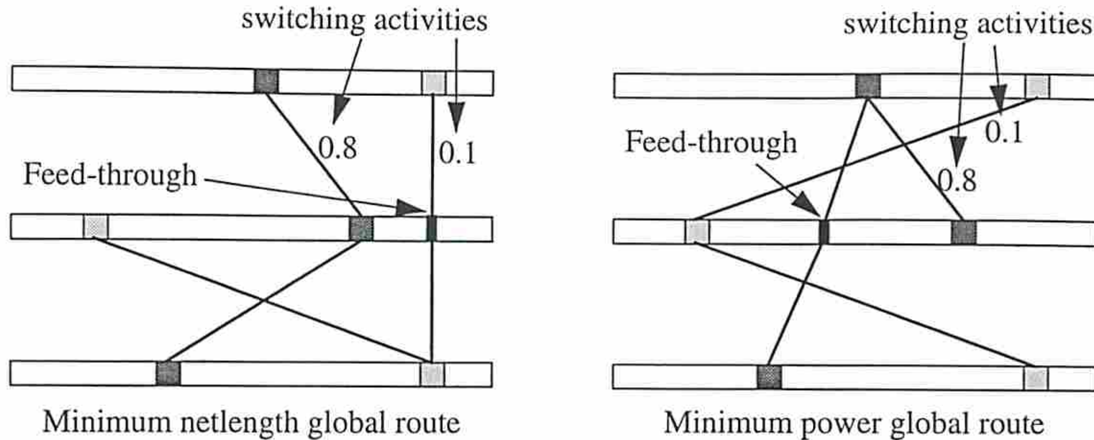


Figure 10.5: A scenario where power dissipation can be improved during global routing.

be updated. A somewhat different strategy has been proposed by Cong et al. [18]. In this approach, spanning trees for all nets are generated in parallel by selecting an edge with minimum cost from an E that is constructed using all nets in the circuit.

Due to the greedy nature of algorithm SEQROUTE, it is highly dependent on the order in which the nets are routed. Also, even within this greedy mechanism the feed-through penalty associated with each row changes dynamically during global routing. In this case, changing the order of routing may have unpredictable effect on the quality of the circuit. Thus, the outcome of above algorithm is crucially dependent on technique used to determine feed-through penalties. Lee et al. [63] have proposed a simulated annealing based global router which assigns feed-through penalties to rows based on an estimate of the length of the longest row after routing. They also utilize a lot of ad-hoc and/or non-deterministic techniques to reduce the estimated number of track during global routing.

As mentioned earlier, for a low power global router, the goal is to minimize total power consumption after global routing by reducing the lengths of nets driven by high switching gates by allowing the lengths of nets driven by low switching gates to increase as shown in Figure 10.5. Under a lumped capacitance model of wires, however, the power dissipation of a net is directly proportional to the length of the net. Thus, all techniques proposed in the papers cited above that minimize the lengths of individual nets are also applicable for low power global routing.

This makes further power minimization during global routing very difficult as the basic objective function is identical to the netlength minimization objective and any additional gain must be obtained by strongly differentiating between nets with high switching activity and nets with low switching activity and by treating them differently. Given two nets with different switching activity, the only way they can be treated differently is by imposing different feed-through penalties on them. Specifically, nets with high switching activity should encounter as little feed-through penalty as possible whereas nets with low switching activity should face more feed-through penalty. Thus, during global routing, improvements in power dissipation must be achieved by manipulating the feed-through penalties for each row. This is performed by simply ordering the nets to be routed by their switching activity. If the feed-through penalty increases as more nets are routed, this technique will result in a circuit with improved power dissipation.

Typically, a very high feed-through penalty is assigned to the longest row as a feed-through insertion in that row will lead to an increase in chip width. If $\mathcal{R}$ corresponds to the set of rows for a placed circuit, then associated with each row $r \in \mathcal{R}$, there is a required number of feed-throughs R_r which corresponds to the feed-throughs required (after subtracting the implicitly available feed-through I_r in that row) to guarantee proper connectivity of the circuit. If L corresponds to the lengths of the longest row in the circuit, then the number of available feed-throughs A_r in a row r with length L_r is calculated as $A_r \frac{L-L_r}{L_f}$ where L_f corresponds to the length of a feed-through cell. By varying the penalty values assigned to these rows and the mechanisms to update these penalty values, several approaches to perform low power global routing can be proposed as described next.

Greedy Approach:

In a greedy method, a very high feed-through penalty is assigned to rows with no available feed-throughs (i.e., $A_r = 0$) and for other rows the penalty is set to zero. However, such greedy method does not guarantee that the feed-through penalty will increase as more nets are routed. This is because a row which is the longest at an intermediate stage may not remain longest at later stages. This leads to a randomized effect on the power dissipation of the resultant circuit. Hence, it was

decided to try a predictive method that assigns feed-through penalties based on an estimate of the number of feed-throughs that will be used at each row.

Predictive Approach:

The predictive approach can be applied at different levels of sophistication. In the simplest case, the required number of feed-throughs R_r is used as an estimate and $R_r * L_f$ is added to the initial length L_r to identify the row with maximum length and to calculate the penalties. In a more sophisticated case, the number of feed-throughs after global routing can be predicted as in TimberWolfSC [63] which can then be used to identify the row with longest expected length and calculate the penalties based on that. Once again, implementations based on both of these approaches resulted in a randomized effect on the power dissipation of the resultant circuits. The reason in this case is that if the prediction is pessimistic, then the effect observed with the greedy method will again randomize the results. On the other end, if the prediction is accurate or optimistic, then the penalties do not change during global routing resulting in identical results. Even in cases where an improvement in total power dissipation was observed, the amount of improvement was less than 5%, which in most cases, disappeared either after shifting the gates to insert the feed-throughs or after feed-through assignment phase. This motivated me to perform the following experiment to see if a greater potential to improve the power dissipation of the circuit exist during global routing.

Linear Cost based Approach:

In the linear cost based approach, the feed-through penalty is linearly increased starting from zero to a large value as more nets are routed. This ideally suits the net ordering based on decreasing values of switching activity as high switching activity nets encounter lower penalties for feed-throughs as compared to low switching activity nets. The circuits obtained with this approach were then compared with circuits generated using a reverse order on the nets. This method is ideal to identify power gain potential from a low power global routing tool. Indeed, the results obtained with this approach showed a more consistent improvement in power after routing tree generation. However, the average improvement was about 2% and the

maximum improvement was about 5%. This clearly indicates that there is *negligible* potential for power improvement during global routing tree generation procedure for standard cell circuits. Furthermore, even this marginal improvement was randomized when shift in gate positions due to newly inserted feed-throughs was taken into account.

On further analysis of the circuits generated by low power global router and netlength minimizing global router, it was observed that the routing topologies of nets remained practically identical for most of the nets. This indicates that, though there may be a large number of "possible" routing topologies for each net, in general, there are very few "reasonable" topologies. Remaining routing topologies are obviously non-optimal and hence discarded by both implementations irrespective of objective functions. In most cases, in fact, there is only one obvious routing topology determined purely based on relative placement of the pins of the net. This reduces the flexibility of a global router to trade off netlength for lower power dissipation in a substantial manner. Furthermore, during global routing, the cost associated with an edge can not be determined exactly. Specifically, the distance between rows is unknown during global routing thereby making any estimate of Y distances (assuming that rows are placed horizontally) inaccurate. Furthermore, even the X distances between gates are likely to be inaccurate as the gate positions may shift subsequent to feed-through insertions. This inaccuracy further randomizes the low power global routing. Finally, as described above, if the power gain is achieved by introducing an additional feed-through on a row, the power penalty due to increased lengths of nets crossing the feed-through location in the channel above and below that row should also be taken into account. Unfortunately, this type of characterization is possible only after detailed routing. Even if such a characterization could be performed during global routing, it will further reduce the potential for power improvement during global routing.

10.2.2 Feed-through Assignment for Low Power

Feed-through assignment (FTA) is one of the key issues in standard cell routing. It is the operation that interfaces the global router with the detailed router. Given global routes on a global routing graph and positions of feed-throughs on rows, the

task of feed-through assignment is to assign these feed-throughs to individual nets such that total number of tracks, total netlengths and/or total number of vias are minimized.

A conventional approach to this problem is to handle feed-through assignment simultaneously during global routing by finding a shortest path routing for global route of each net. However, with this approach, the results once again depend on the net ordering. One of the main motivations for doing a separate feed-through assignment is to remove the effect of dependency on net ordering during global routing. Hence, typically the feed-through assignment problem is solved using *linear assignment* between the feed-throughs on a row and the set of pins connected to them [102, 78, 68]. Linear assignment, along with other heuristic techniques, has also been implemented in TimberWolfSC [63]. Linear assignment based techniques, however, handle each row separately, possibly leading to a globally sub-optimal solution. Hence, Hong et al. [41] have formulated feed-through assignment problem as vertical channel routing problem where they handle all rows simultaneously. However, the experimental results obtained with this approach do not show significant improvements over TimberWolfSC. Recently, Okamoto et al. [77] have proposed a flow based formulation of feed-through assignment. However, they compare their results only with a shortest path based approach and still report only marginal improvements.

In my implementation of feed-through assignment for low power, a linear assignment based approach was used as it produces as good results as other approaches. Furthermore, it provides a robust framework where it is relatively easy to incorporate a low power objective function. Specifically, the netlength minimizing feed-through assignment formulation was modified to perform low power feed-through assignment simply by weighing each element[2] in the linear assignment cost matrix by the corresponding switching activity.

The experimental results showed consistent but minor improvements in power dissipation. Once again, after the detailed routing, the effect of feed-through assignment was randomized. The reason for minimal improvements during feed-through

[2]Each element in linear assignment cost matrix for feed-through assignment problem corresponds to the netlength/power cost associated with assignment of the corresponding net to the corresponding feed-through.

assignment can also be attributed to practically identical feed-through *allocation* and further randomization by allowing any feed-through to be assigned to any net.

10.2.3 Conclusion: Circuit Routing for Low Power

In conclusion, it appears that the potential for power minimization during global routing and feed-through assignment is very limited. In these experiments, the results were compared using identical formulations (with different objective functions) of global routing tools and feed-through assignment tools. Results are likely to be even more randomized when these tools are applied in conjunction with a routing tool like TimberWolfSC which apply a lot of non-deterministic techniques and ad-hoc heuristics to improve the circuit. It is my belief that the potential for improving the power dissipation of a circuit based on a differential treatment of nets diminishes as later stages of physical design are targeted. The main reason for this is that the flexibility available to these tools to trade off lengths of low activity nets for shorter lengths of high activity nets is reduced. This is the main reason I have not attempted to minimize circuit power during detailed routing. However, for sake of completeness, in the following, I describe how detailed routing tools can be targeted to minimize power.

There are two types of detailed-routing problems: channel routing and switch-box routing. Traditionally, the main issue in channel routing is to minimize the channel width by judicious assignment of track to the nets to be routed. In a sequential channel-routing paradigm, the order of selection of the net to be routed can also affect the routability of other nets. Channel routing tools have matured significantly during the last decade and a large number of efficient and fast detailed routing tools are available.

As mentioned above, there are mainly two operations in channel routing, namely, net selection and track assignment. This phase is usually followed by a vertical constraint violation removal phase[3] during which vertical constraint violations are removed by introducing dog-legs in the vertical and/or horizontal segments. Power can be improved during channel routing by appropriate modification of each of

[3]Vertical constraints are caused due to two nets requiring the same vertical segment to complete the routing.

these operations, i.e., (1) nets could be selected to be routed based on the power criticality of the nets; (2) nets could be assigned to the track such that netlength of a high activity net is reduced and such that minimal vertical constraint violations are introduced for power critical nets (which would require them to meander, resulting in additional power consumption); and (3) dog-legs could be introduced such that power critical nets suffer less netlength penalty.

For switch box routing, the issue is to be able to complete the routing task with given resources while optimizing a parameter like number of vias, total wire-length etc. Appropriate net selection and a judicious routing mechanism for power critical nets could be attempted to improved power dissipation during switch-box routing.

Chapter 11

Conclusion: Power-driven Physical Design

In recent years, high performance circuit design for low power has gained enormous significance owing to the growing popularity of portable devices. With the convergence of consumer, computing and communication technologies and extra emphasis on portability of devices, this trend is likely to continue. Even in the absence of the incentive provided by portable devices, reliability and packaging issues demand that average and maximum power dissipation of circuits be reduced. In fact, with the advent of deep-submicron technology, the main bottleneck in putting millions of transistor on one chip is the packaging limitations due to excessive power dissipation. Thus, it is anticipated that for most of the circuits designed in future, optimization of power dissipation will at least be as important as optimization of circuit area and delay. This growing importance of low power circuit design has put tremendous pressure on the design automation community to develop techniques ranging from power estimation/analysis to power optimization methodologies at different levels of design hierarchy.

Recently, research in physical design is coming to the forefront again. The main reason for this is that ever-changing circuit technology has a direct and immediate impact on physical design tools. Furthermore, increasing dominance of interconnect is pushing physical design into the limelight as physical design is the only phase that can handle interconnect accurately. Thus, increasing significance of interconnect and growing demand for low power yet high performance CAD techniques necessitate physical design tools to minimize power dissipation of the circuit.

11.1 Main Results and Contributions

In the second part of my dissertation I have presented performance-driven circuit partitioning, performance-driven circuit placement and circuit routing tools. I briefly summarize my contribution in the following.

11.1.1 Delay Optimal Circuit Partitioning for Low Power

Circuit partitioning is important for a number of reasons during synthesis and physical design, i.e., due to physical limitations on the number of transistors a chip or module can accommodate (e.g., to implement a large design on multiple chip modules or on fixed size PLA/FPGAS), for performing circuit restructuring during synthesis, for reducing complexity of synthesis or layout procedures by reducing the problem size, etc.. I have proposed a clustering/partitioning mechanism [114, 115] that achieves delay optimality by allowing gate replication. In the process, a generic mechanism for implicitly enumerating all clustering solutions of a tree circuit was proposed thereby solving the clustering problem optimally, irrespective of the objective function. In this particular case, the power dissipation of the resulting circuit was minimized. For DAGs, this approach does not guarantee power optimality but still produces results that are significantly better than the existing clustering algorithms. Furthermore, it was shown that the problem of enumerating all possible cluster patterns covering n gates is identical to enumerating all alphabetic trees on $n+1$ leaf nodes. Based on this, an efficient mechanism to incrementally enumerate all cluster patterns at a gate was proposed. Since the total number of clusters that need to be enumerated is exponential in the size of the cluster, a hierarchical clustering/partitioning approach was proposed that not only minimizes power dissipation but is also likely to reduce gate replication.

11.1.2 Standard Cell Placement and Routing for Low Power

For low power placement, I proposed a quadratic programming based placement tool that also handles delay constraints effectively [110]. To obtain a placement solution for standard cell based circuits, a low power slot assignment phase based

on low power linear assignment was proposed. The experimental results indicate that considerable improvements in power dissipation can be achieved by trading off lengths of idle nets to reduce the lengths of active nets by appropriately placing the gates. In fact, the improvements can be substantial when distribution of net switching rate is uniform instead of being clustered around the highest value of switching rates. However, my experiments indicate that once the gates are placed, there is not much potential to improve power dissipation simply by allowing different net topologies in a standard cell based circuit. This is even more true during feed-through assignment where the topologies are also finalized, the only flexibility being in different assignments of feed-throughs to the nets. I believe that this trend of reducing returns for power minimization is going to continue into detailed routing where the only flexibility is in assigning tracks, with the objective in most cases being a reduction of the total number of tracks.

11.2 Discussion and Future Directions

In my experiments with physical design for low power, I noticed a clear decrease in potential for power optimization as later operations of physical design are targeted. Specifically, the improvement in power dissipation was the most during performance-driven partitioning, followed by placement, global routing and the least during feed-through assignment. In retrospect, however, this does not strike as surprising. Clearly, the greater the flexibility to reduce power dissipation, the greater will be the improvements in power dissipation. During performance driven partitioning, the partitioning tool is given a lot of freedom by allowing gate replication. During placement, this freedom is taken away but the placement tool still has complete freedom to place the gates anywhere on the circuit. Since the netlengths are mostly determined by the placement of the nets, a placement tool can still impact the power dissipation of the circuit significantly. This is particularly true considering that most existing routers can route nets almost optimally given the placement positions. Unfortunately, this is also the main reason for reduced potential for further power minimization during routing. Since netlength is directly proportional to power dissipation under the lumped capacitance model, optimal netlength routes are also optimal with respect to power. Thus, the only additional power gain that

can be achieved during routing is by trading off lengths of low activity nets to improve lengths of high activity nets only by modifying the routing topology of the nets. When power gain has to be achieved at this detail, even a slight inaccuracy during netlength cost calculation is likely to undo any power gain anticipated. As my experience indicates, this is particularly true for global routing and feed-through assignment phases for a standard cell based circuit.

An extension that can be considered is to minimize power under detailed circuit models. Specifically, low-power physical design tools could be developed such that power dissipation under a real delay model (i.e., taking into account power dissipation due to glitches) is minimized. Since power dissipation under real delay depends significantly on interconnect delays/load, I feel that the only phase to perform power-minimization under a real delay model is during routing where accurate estimates of interconnect length/delay/load are available. Unfortunately, the only freedom available during routing is via the manipulation of interconnects (e.g., interconnect length, width, etc.). One way to minimize real delay model based power dissipation is by balancing the paths by controlling the interconnect delay using net length/width. If a reasonable assignment of required time at each pin can be obtained which leads to reduced glitching, netlengths and widths can be controlled accordingly to achieve corresponding arrival times. This should be performed during routing such that satisfaction of these required time results in improved power dissipation at little cost in area and delay. Unfortunately, I believe that even in this case, it is unlikely that significant improvement in power dissipation can be achieved.

This leads to exploring possible alternatives to minimizing power during physical design. From my experiences, I believe that to avoid minimization of inaccurate estimates of power and to guarantee an overall power improvement, power optimization should be attempted after detailed routing in a post-layout optimization phase. In a post-layout optimization phase, several techniques can be proposed to tangibly improve the power dissipation. For example, nets may be re-routed or re-connected and gates may be re-placed during post-layout optimization such that each such modification guarantees an improvement in power dissipation. When re-connection is restricted only to feed-throughs, then it is identical to a feed-through reassignment performed in a post layout phase. However, this technique can be applied to

gates also, in which case it may have greater potential to improve power. As mentioned in Chapter 8, in a post-layout optimization phase, the operations need not be restricted to physical design operations mentioned above. Indeed, techniques like driver resizing, buffer optimization etc. along with wire sizing [17] can be applied in a post-layout optimization phase. More powerful Boolean transformations or incremental Boolean modifications targeting power minimization can be applied with a greater effect on placed circuits. Also, power dissipation due to glitches can be minimized significantly using a post-layout optimization operation that filters out glitches at nets with a large number of glitches.

Reference List

[1] M. Alidina, J. Monteiro, S. Devadas, A. Ghosh, and M. Papaefthymiou. Precomputation-based sequential logic optimization for low power. In *Proc. 1994 International Workshop on Logic Power Design*, pages 57–62, April 1994.

[2] H. B. Bakoglu. *Circuits, interconnections, and packaging for VLSI.* Addison-Wesley, 1990.

[3] L. Benini, M. Favalli, and B. Ricco. Analysis of hazard contribution to power dissipation in CMOS IC's. In *Proc. 1994 International Workshop on Logic Power Design*, pages 27–32, April 1994.

[4] C. L. Berman and J. L. Carter. The fanout problem: From theory to practice. In C. L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the 1989 Decennial Caltech Conference*, pages 69–99. MIT Press, May 1989.

[5] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multi-level logic optimization and the rectangular covering problem. In *Proceedings of the IEEE International Conference on Computer Aided Design*, Nov. 1987.

[6] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli. Multilevel logic synthesis. In *Proceedings of the IEEE*, volume 78, pages 264–300, February 1990.

[7] R. K. Brayton and C. McMullen. The decomposition and facorization of Boolean expressions. In *Proceedings of the International Symposium on Circuits and Systems*, pages 49–54, Rome, May 1982.

[8] R. Burch, F. Najm, P. Yang, and D. Hocevar. Pattern-independent current estimation for reliability analysis of CMOS circuits. In *Proceedings of the 25th Design Automation Conference*, pages 294–299, June 1988.

[9] A. Cayley. On the theory of analytical forms called trees. In *The Collected Mathematical Papers of Arthur Cayley: Volume 3*, pages 242–246. Cambridge University Press: Reprint–Johnson Reprint Corporation, New York, New York, 1890. From the Philosophical Magazine, vol. XIII. (1857),172-176.

[10] A. Cayley. On the analytical forms called trees. second part. In *The Collected Mathematical Papers of Arthur Cayley: Volume 4*, pages 112–115. Cambridge University Press: Reprint–Johnson Reprint Corporation, New York, New York, 1891. From The Philosophical Magazine, vol. XVIII. (1859),374-378.

[11] A. Cayley. A theorem on trees. In *The Collected Mathematical Papers of Arthur Cayley: Volume 13*, pages 26–28. Cambridge University Press: Reprint–Johnson Reprint Corporation, New York, New York, 1891. From the Quaterly Journal of Pure and Applied Mathemetics, vol. XXIII. (1889),376-378.

[12] T. H. Chao and Y. C. Hsu. Rectilinear Steiner tree construction by local and global refinement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13:1335–1349, December 1989.

[13] K. Chaudhary and M. Pedram. A near-optimal algorithm for technology mapping minimizing area under delay constraints. In *Proceedings of the 29th Design Automation Conference*, pages 492–498, June 1992.

[14] K. C. Chen, J. Cong, Y. Ding, A. Kahng, and P. Trajmar. DAG-MAP: Graph based FPGA technology mapping for delay optimization. *IEEE Design and Test of Computers*, pages 7–20, September 1992.

[15] F. R. K. Chung and F. K. Hwang. The largest minimal rectilinear Steiner trees for a set of n points enclosed in a rectangle with given perimeter. *Networks*, 9:19–36, 1979.

[16] M. Cirit. Estimating dynamic power consumption of CMOS circuits. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 534–537, November 1987.

[17] J. Cong, C-K. Koh, and K-S. Leung. Wire sizing with driver sizing for performance and power optimization. In *Proc. 1994 International Workshop on Logic Power Design*, pages 81–86, April 1994.

[18] J. Cong and B. T. Preas. A new algorithm for standard cell global routing. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 176–180, November 1988.

[19] D. Coppersmith, M. M. Klawe, and N. J. Pippenger. Alphabetic minimax trees of degree at most t. *SIAM Journal of Computing*, 15:189–192, 1986.

[20] A. E. Dunlop and B. W. Kernighan. A procedure for placement of standard-cell VLSI circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-4(1):92–98, January 1985.

[21] P. Eades and N. C. Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11:379–403, 1994.

[22] P. Erdös and R. K. Guy. Crossing number problems. *American Mathematical Monthly*, 1972.

[23] Shimon Even. *Graph Algorithms.* Computer Science press, Rockville, Maryland, first edition, 1979.

[24] Philippe Flajolet. Personal Communication, August 1993.

[25] A. D. Friedman and P. R. Menon. *Theory and Design of Switching Circuits.* Computer Science Press, 1975.

[26] R. G. Gallager. Variations on a theme by Huffman. *IEEE Transactions on Information Theory*, IT-24(6):668–674, November 1978.

[27] M. R. Garey. Optimal binary identification procedures. *SIAM Journal of Applied Mathematics*, 23(2):173–186, September 1972.

[28] M. R. Garey. Optimal binary search trees with restricted maximal depth. *SIAM Journal of Computing*, 3(2):101–110, June 1974.

[29] M. R. Garey and D. S. Johnson. The rectilinear Steiner tree problem is NP-complete. *SIAM Journal of Applied Mathematics*, 32(4):37–58, 1977.

[30] A. Ghosh, S. Devadas, K. Kuetzer, and J. White. Estimation of average switching activity in combinational and sequential circuits. In *Proceedings of the 29th Design Automation Conference*, pages 253–259, June 1992.

[31] E. N. Gilbert and E. F. Moore. Variable-length binary encoding. *Bell Systems Technical Journal*, 38:933–968, 1959.

[32] C. R. Glassey and R. M. Karp. On the optimality of Huffman trees. *SIAM Journal of Applied Mathematics*, 31(2):368–378, September 1976.

[33] Richard Goering. Deep-submicron: Coping with complexity. *Electronics Engineering Times*, pages 49–72, October 1994.

[34] M. C. Golumbic. Combinatorial merging. *IEEE Transactions on Computers*, 25(11):514–526, 1976.

[35] L. Gotlieb. Optimal multi-way search trees. *SIAM Journal of Computing*, 10:422–433, 1981.

[36] L. Gotlieb and D. Wood. The construction of optimal multiway search trees and the monotonicity principle. *International J. Computer Maths*, 9:17–24, 1981.

[37] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Publishing Company, Reading, Massachusetts, first edition, 1988.

[38] R. K. Guy. Crossing numbers of graphs. In Y. Alavi, D. R. Lick, and A. T. White, editors, *Lecture Notes in Mathematics #303: Proceedings of the conference on Graph Theory and Applications at Western Michigan University*, pages 111–124. Springer-Verlag, New York, U.S.A., May 1972.

[39] R. K. Guy. Latest results on crossing numbers. In M. Capobianco, J. B. Frechen, and M. Krolik, editors, *Lecture Notes in Mathematics #186: Proceedings of the First New York City Graph Theory Conference*, pages 143–156. Springer-Verlag, New York, U.S.A., June 1972.

[40] J. M. Ho, G. Vijayan, and C. K. Wong. New algorithms for the rectilinear Steiner tree problem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9:185–193, February 1990.

[41] X. L. Hong, J. Huang, C. K. Cheng, and E. S. Kuh. FARM: an efficient feed-through pin assignment algorithm. In *Proceedings of the 29th Design Automation Conference*, pages 530–535, June 1992.

[42] H. J. Hoover, M. M. Klawe, and N. J. Pippenger. Bounding fanout in logical networks. *Journal of the Association for Computing Machinery*, 31(1):13–18, January 1984.

[43] Y. Horibe. An improved bound for weight-balanced tree. *Information and Control*, 34:148–151, 1977.

[44] T. C. Hu, D. J. Kleitman, and J. K. Tamaki. Binary trees optimum under various criteria. *SIAM Journal of Applied Mathematics*, 37(2):246–256, October 1979.

[45] T. C. Hu and M. T. Shing. Computation of matrix chain products. part I. *SIAM Journal of Computing*, 11(2):362–373, May 1982.

[46] T. C. Hu and A. C. Tucker. Optimal computer search trees and varaible-length alphabetical codes. *SIAM Journal of Applied Mathematics*, 21(4):514–532, December 1971.

[47] D. A. Huffman. A method for the construction of minimum redundancy codes. In *Proceedings of the IRE*, volume 40, pages 1098–1101, September 1952.

[48] F. K. Hwang. The rectilinear Steiner problem. *J. Design Automation and Fault-Tolerant Computing*, pages 303–310, 1978.

[49] S. Iman and M. Pedram. Multi-level network optimization for low power. In *Proceedings of the IEEE International Conference on Computer Aided Design*, November 1994.

[50] A. Itai. Optimal alphabetic trees. *SIAM Journal of Computing*, 5(1):9–18, 1976.

[51] M. A. B. Jackson and E. S. Kuh. Performance-driven placement of cell based IC's. In *Proceedings of the 26th Design Automation Conference*, pages 370–375, June 1989.

[52] D. S. Parker Jr. Conditions for optimality of the Huffman algorithm. *SIAM Journal of Computing*, 9(3):470–489, 1980.

[53] A. B. Kahng and G. Robins. A new class of iterative Steiner tree heuristics with good performance. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(7):893–902, July 1990.

[54] D. G. Kirkpatrick and M. M. Klawe. Alphabetic minimax trees. *SIAM Journal of Computing*, 14(3):514–526, 1985.

[55] J. M. Kleinhans, G. Sigl, F. M. Johannes, and K. J. Antreich. GORDIAN: VLSI placement by quadratic programming and slicing optimization. *IEEE Transactions on Computer-Aided Design*, CAD-10:356–365, March 1991.

[56] D. J. Kleitman. The crossing number of $k_{5,n}$. *Journal of Combinatorial Theory*, 9:315–323, 1971.

[57] D. Knuth. *The Art of Computer Programming.* Addison-Wesley, 1973.

[58] D. E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971.

[59] F. J. Kurdahi and A. C. Parker. Techniques for area estimation of VLSI layouts. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-8(1):81–92, January 1989.

[60] Y-T. Lai, K-R. R. Pan, and M. Pedram. FPGA synthesis using function decomposition. In *Proceedings of the International Conference on Computer Design*, pages 30–35, October 1994.

[61] E. L. Lawler. An approach to multilevel Boolean minimization. *Journal of the Association for Computing Machinery*, 11, July 1964.

[62] E. L. Lawler, K. N. Levitt, and J. Turner. Module clustering to minimize delay in digital networks. In *IEEE Transactions on Computers*, volume C-18, pages 47–57, January 1969.

[63] K. W. Lee and C. Sechen. A new global router for row-based layout. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 180–183, November 1988.

[64] F. T. Leighton. New lower bound techniques for VLSI. *Mathematical Systems Theory*, 17:47–70, 1984.

[65] T. Lengauer. *Combinatorial algorithms for integrated circuit layout.* Computer Science. Wiley-Teubner, 1990.

[66] Y. Lirov and O. Yue. Circuit pack troubleshooting via semantic control I: Goal selection. In *Proceedings of the International Workshop on Artificial Intelligence for Industrial Applications*, pages 118–122, 1988.

[67] P. McGeer. *On the interaction of functional and timing behavior in combinational circuits.* PhD thesis, University of California, Berkeley, 1989.

[68] G. Meixner and U. Lauther. A new global router based on a flow model and linear assignment. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 44–47, 1990.

[69] J. Monteiro, S. Devadas, and A. Ghosh. Retiming sequential circuits for low power. In *Proceedings of the IEEE International Conference on Computer Aided Design*, November 1993.

[70] J. W. Moon. *Counting Labelled Trees.* Canadian Mathematical Congress, Montreal, Canada, 1970.

[71] C. R. Morrison, R. M. Jacoby, and G. D. Hachtel. TECHMAP: Technology mapping with delay and area optimization. In *Proceedings of the International*

Workshop on Logic and Architecture Synthesis for Silicon Compilers, pages 53–64, Grenoble, France, May 1988.

[72] T. Motzkin. Relations between hypersurface cross ratios, and a combinatorial formula for partitions of a polygon, for permanent preponderance, and for non-associative products. *Bulletin of American Mathematical Society*, 54:352–360, 1948.

[73] R. Murgai, R. K. Brayton, and A. Sangiovanni-Vincentelli. On clustering for minimum delay/area. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 6–9, November 1991.

[74] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic synthesis for programmable gate arrays. In *Proceedings of the 27th Design Automation Conference*, pages 620–625, June 1990.

[75] F. N. Najm. Transition density, a stochastic measure of activity in digital circuits. In *Proceedings of the 28th Design Automation Conference*, pages 518–521, June 1991.

[76] N. Nakatsu. Bounds on the redundancy of binary alphabetic codes. *IEEE Transactions on Information Theory*, 37(4):1225–1229, July 1991.

[77] T. Okamoto, M. Ishikawa, and T. Fujita. A new feed-through assignment algorithm based on a flow model. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 775–778, November 1993.

[78] T. M. Parng and R. S. Tsay. A new approach to sea-of-gates global routing. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 52–55, November 1989.

[79] K. R. Pattipati and M. G. Alexandridis. Application of heuristic search and information theory to sequential fault diagnosis. *IEEE Transactions on Systems, Man, and Cybernetics*, 40(4):872–887, 1990.

[80] M. Pedram. *An integrated approach to logic synthesis and physical design.* PhD thesis, University of California, Berkeley, August 1991. Technical Report UCB/ERL M91/69.

[81] M. Pedram and N. Bhat. Layout driven logic restructuring / decomposition. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 134–137, November 1991.

[82] M. Pedram and N. Bhat. Layout driven technology mapping. In *Proceedings of the 28th Design Automation Conference*, pages 99–105, June 1991.

[83] M. Pedram and B. T. Preas. Accurate prediction of physical design characteristics of random logic. In *Proceedings of the International Conference on Computer Design*, pages 100–108, October 1989.

[84] M. Pedram and H. Vaishnav. Technology decomposition using optimal alphabetic trees. In *Proceedings of the European Conf. on Design Automation*, pages 573–577, March 1993.

[85] R. Rajmohan and D. F. Wong. Optimal clustering for delay minimization. In *Proceedings of the 30th Design Automation Conference*, pages 309–314, June 1993.

[86] J. Rajski and J. Vasudevamurthy. The testability-preserving concurrent decomposition and factorization of Boolean expressions. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 11, pages 778–793, June 1992.

[87] J. Reed, A. Sangiovanni-Vincentelli, and M. Santamauro. A new symbolic channel router: YACR2. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 208–219, July 1985.

[88] J. Riordan. *Combinatorial Identities.* Robert E. Krieger Publishing Company, Huntington, New York, revised edition, 1979.

[89] J. Roth and R. Karp. Minimization over Boolean graphs. *IBM Journal of Research and Development*, 6(2):227–238, April 1962.

[90] S. Sastry and A. C. Parker. Stochastic models for wirability analysis of gate arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-5(1):52–65, January 1986.

[91] G. Saucier, J. Fron, and P. Abouzeid. Lexicographical expressions of Boolean functions with applications to multilevel synthesis. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 12, pages 1642–1654, November 1993.

[92] H. Savoj, H. Y. Wang, and R. Brayton. Improved scripts in MIS-II for logic minimization of combinational circuits. In *Proceedings of the International Workshop on Logic Synthesis*, May 1991.

[93] I. Schur. Über eine Klasse von Mittelbildungen mit Anwendungen auf die Determinantentheorie. *Berl. Math. Ges.*, 22:9–20, 1923. In German.

[94] C. Sechen. Average interconnection length estimation for random and optimized placements. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 190–193, November 1987.

[95] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovani-Vincentelli. Sequential circuit design using synthesis and optimization. In *Proceedings of the International Conference on Computer Design*, October 1992.

[96] Jeremy F. Shapiro. *Mathematical Programming: Structures and Algorithms.* John Wiley and Sons, New York, New York, 1979.

[97] G. Sigl, K. Doll, and F. Johannes. Analytical placement: A linear or a quadratic objective function? In *Proceedings of the 28th Design Automation Conference*, pages 427–431, June 1991.

[98] K. J. Singh and A. Sangiovanni-Vincentelli. A heuristic algorithm for the fanout problem. In *Proceedings of the 27th Design Automation Conference*, pages 357–360, June 1990.

[99] K. J. Singh, A. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Timing optimization of combinational logic. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 282–285, November 1988.

[100] A. Srinivasan. *Performance Optimization of Large-Scale Integrated Circuits.* PhD thesis, University of California, Berkeley, November 1991. Memorandum No. UCB/ERL M91/104.

[101] A. Srinivasan, K. Chaudhary, and E. S. Kuh. RITUAL: An algorithm for performance-driven placement of cell-based IC's. In *Proceedings of the Third Physical Design Workshop*, May 1991.

[102] B. S. Ting and B. N. Tien. Routing techniques for gate array. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2(1):301–312, 1983.

[103] H. J. Touati, C. W. Moon, R. K. Brayton, and A. Wang. Performance-oriented technology mapping. In *Proceedings of the Sixth M.I.T. Conference on Advanced Research in VLSI*, pages 79–97, April 1990.

[104] Herve Touati. *Performance Oriented Technology Mapping.* PhD thesis, University of California, Berkeley, 1990.

[105] R. S. Tsay, E. S. Kuh, and C. P. Hsu. PROUD: A sea-of-gates placement algorithm. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 318–323, November 1988.

[106] C. Tsui, M. Pedram, and A. Despain. Efficient estimation of dynamic power dissipation. In *Proceedings of the IEEE International Conference on Computer Aided Design*, November 1993.

[107] C. Tsui, M. Pedram, and A. Despain. Technology decomposition and mapping for low power. In *Proceedings of the 30th Design Automation Conference*, pages 68–73, June 1993.

[108] C-Y. Tsui, M. Pedram, C-A. Shen, and A. M. Despain. Low power state assignment targeting two and multi-level logic implementations. In *Proceedings of the IEEE International Conference on Computer Aided Design*, November 1994.

[109] H. Vaishnav and M. Pedram. Alphabetic fanout optimization. Technical Report CEng 92-15, University of Southern California, 1992.

[110] H. Vaishnav and M. Pedram. PCUBE: a performance-driven placement algorithm for low power designs. In *Proceedings of the European Design Automation Conference*, September 1993.

[111] H. Vaishnav and M. Pedram. Routability-driven fanout optimization. In *Proceedings of the 30th Design Automation Conference*, pages 230–236, June 1993.

[112] H. Vaishnav and M. Pedram. Enumeration and optimization of alphabetic trees under various criteria. Submitted for publication, 1994.

[113] H. Vaishnav and M. Pedram. An exact framework for post-layout timing correction. Technical Report CEng 94-36, University of Southern California, 1994.

[114] H. Vaishnav and M. Pedram. Delay optimal partitioning targeting low power VLSI circuits. In *Proceedings of the IEEE International Conference on Computer Aided Design*, November 1995.

[115] H. Vaishnav and M. Pedram. Delay optimal partitioning targeting low power VLSI circuits. Technical Report CEng 95-18, University of Southern California, 1995.

[116] H. Vaishnav and M. Pedram. Logic extraction based on normalized netlengths. In *Proceedings of the International Conference on Computer Design*, October 1995.

[117] H. Vaishnav and M. Pedram. Logic extraction using fanin ranges. In *5th European Workshop on Power and Timing Modeling (PATMOS'95)*, October 1995.

[118] H. Vaishnav and M. Pedram. Minimizing the routing cost during logic extraction. In *Proceedings of the 32nd Design Automation Conference*, June 1995.

[119] H. Vaishnav and M. Pedram. A new method for performance oriented logic extraction. In *International Workshop on Logic Synthesis*, May 1995.

[120] H. Vaishnav and M. Pedram. Normalized netlengths: A measure of routing cost for logic synthesis. Technical Report CEng 95-17, University of Southern California, 1995.

[121] V. K. Vaishnavi, H. P. Kriegel, and D. Wood. Optimum multiway search trees. *Acta Informatica*, 14:119–133, 1980.

[122] A. Wang. *Algorithms for multi-level logic optimization.* PhD thesis, University of California, Berkeley, April 1989. Technical Report UCB/ERL M89/50.

[123] R. L. Wessner. Optimal alphabetic search trees with restricted maximal height. *Information Processing Letters*, 4(4):90–94, 1981.

[124] Douglas B. West. *The Art of Combinatorics.* Unpublished, In preparation.

[125] T. W. Williams, B. Underwood, and M. R. Mercer. The interdependence between delay-optimization of synthesized networks and testing. In *Proceedings of the 28th Design Automation Conference*, pages 87–92, June 1991.

[126] S. Yang. Logic synthesis and optimization benchmarks user guide: Version 3.0. Jan 1991.

[127] R. W. Yeung. Alphabetic codes revisited. *IEEE Transactions on Information Theory*, 37(3):564–572, May 1991.