# LAYOUT AWARE SYNTHESIS

by

Wilsin Gosti

Memorandum No. UCB/ERL M00/67

21 December 2000

# LAYOUT AWARE SYNTHESIS

by

Wilsin Gosti

Memorandum No. UCB/ERL M00/67

21 December 2000

**ELECTRONICS RESEARCH LABORATORY**

College of Engineering
University of California, Berkeley
94720

**Layout Aware Synthesis**

by

Wilsin Gosti

B.S. (University of Southwestern Louisiana) 1987
M.S. (Iowa State University) 1990

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering-Electrical Engineering and Computer Sciences

in the

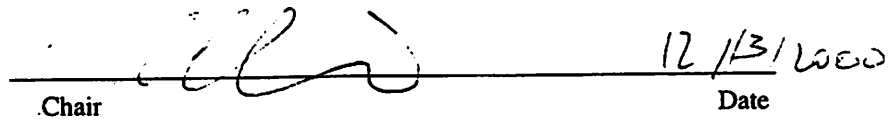
GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Alberto L. Sangiovanni-Vincentelli, Chair
Professor Robert K. Brayton
Professor Steven N. Evans

Fall 2000

The dissertation of Wilsin Gosti is approved:



Chair 12/13/2000
Date

R. K. Brayton 12/18/2000
Date

Steven Evans 12/21/2000
Date

University of California, Berkeley

Fall 2000

# Abstract

Layout Aware Synthesis

by

Wilsin Gosti

Doctor of Philosophy in Engineering-Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Alberto L. Sangiovanni-Vincentelli, Chair

As technology scales into smaller feature sizes, the gate delay scales down, the capacitance of the interconnect per unit feature size in length scales down, but its resistance per unit feature size in length scales up. As a result, the delay component due to the resistance of the interconnect increases with scaling when compared with the gate delay. Not only that this is having adverse effects on global wires which are wires that connect gates that are far apart, but also at the local wires, which are wires that connect gates within a functional module as our results in this thesis show.

The increase in interconnect delay requires that assumptions that have been traditionally accepted be scrutinized. In particular, logic synthesis which assumes that majority of the circuit delay is contributed by gates in the circuit needs to be re-visited. We propose a novel approach that assumes all the circuit delay is contributed by the circuit interconnect. Under this assumption, we show that conventional logic synthesis can produce a circuit that if placed produces a placement that requires long wires. We show a theoretical framework to identify nodes in the Boolean network representing the circuit that will cause long wires in placement, and an operation that eliminates such nodes. We introduce a set of logic operations that optimizes the Boolean network under the constraint that nodes produced do not require long wires.

Technology scaling enables the integration of many millions of devices on a single die. Conventional design flow, which treat logic synthesis and physical design separately, exhibit an inability to achieve timing closure. Timing closure problems occur when timing estimates computed during logic synthesis do not match with timing estimates computed from the layout of the circuit. In such a situation, logic synthesis and layout synthesis are iterated until the estimates match. The

# Abstract

Layout Aware Synthesis

by

Wilsin Gosti

Doctor of Philosophy in Engineering-Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Alberto L. Sangiovanni-Vincentelli, Chair

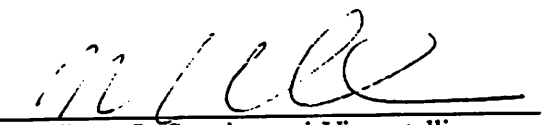
As technology scales into smaller feature sizes, the gate delay scales down, the capacitance of the interconnect per unit feature size in length scales down, but its resistance per unit feature size in length scales up. As a result, the delay component due to the resistance of the interconnect increases with scaling when compared with the gate delay. Not only that this is having adverse effects on global wires which are wires that connect gates that are far apart, but also at the local wires, which are wires that connect gates within a functional module as our results in this thesis show.

The increase in interconnect delay requires that assumptions that have been traditionally accepted be scrutinized. In particular, logic synthesis which assumes that majority of the circuit delay is contributed by gates in the circuit needs to be re-visited. We propose a novel approach that assumes all the circuit delay is contributed by the circuit interconnect. Under this assumption, we show that conventional logic synthesis can produce a circuit that if placed produces a placement that requires long wires. We show a theoretical framework to identify nodes in the Boolean network representing the circuit that will cause long wires in placement, and an operation that eliminates such nodes. We introduce a set of logic operations that optimizes the Boolean network under the constraint that nodes produced do not require long wires.

Technology scaling enables the integration of many millions of devices on a single die. Conventional design flow, which treat logic synthesis and physical design separately, exhibit an inability to achieve timing closure. Timing closure problems occur when timing estimates computed during logic synthesis do not match with timing estimates computed from the layout of the circuit. In such a situation, logic synthesis and layout synthesis are iterated until the estimates match. The

number of such iterations is becoming larger as technology scales. Timing closure problems occur mainly due to the difficulty in accurately predicting the interconnect delay during logic synthesis. This is aggravated by the increase of interconnect delay relative to gate delay.

To address the timing closure problem, we propose an algorithm that integrates logic synthesis and global placement. We introduce technology independent optimization and technology dependent algorithm that interleave their logic operations with incremental local and global placement, in order to maintain a consistent placement while the algorithm is run. In this integrated approach, we introduce wire-planning based heuristics to minimize interconnect delay. We show that by integrating logic synthesis and placement, we avoid the need to predict interconnect delay during logic synthesis. We demonstrate that our scheme significantly enhances the predictability of wire delays, thereby minimizing the timing closure problem. Our results show that the integrated approach result in a significant reduction in both interconnect delay and circuit delay.



Professor Alberto L. Sangiovanni-Vincentelli
Dissertation Committee Chair

To my parents Patrice and Siti, my wife Paylin, and my daughter Emily

# Contents

**List of Figures** v

**List of Tables** vii

**1 Introduction** 1
- 1.1 Thesis Outline . . . 2
- 1.2 Technology and Design Trend . . . 4
  - 1.2.1 Technology Models Used in this Thesis . . . 4
  - 1.2.2 Technology Scaling . . . 4
    - 1.2.2.1 Predicting Sizes of Synthesized Blocks . . . 6
    - 1.2.2.2 Predicting Interconnect Delay . . . 8
  - 1.2.3 The Timing Closure Problem . . . 9

**2 Preliminaries** 16
- 2.1 Logic Functions . . . 16
- 2.2 Logic Synthesis . . . 17
  - 2.2.1 Technology Independent Optimization . . . 18
  - 2.2.2 Technology Dependent Optimization . . . 21
- 2.3 Circuit Delay . . . 22
  - 2.3.1 Delay Models . . . 22
  - 2.3.2 Timing Analysis . . . 23

**3 Wire-planning** 25
- 3.1 Previous Work . . . 26
- 3.2 Preliminaries . . . 27
- 3.3 Logic Synthesis and Interconnect Delay: An Example . . . 27
- 3.4 Constraint Generation . . . 30
  - 3.4.1 Region Placement Constraints . . . 31
  - 3.4.2 Node Placement Constraints . . . 32
  - 3.4.3 Properties of Placement Constraints on Boolean Networks . . . 32
  - 3.4.4 Make-Legal . . . 39
- 3.5 Constraint-Driven Synthesis . . . 40
  - 3.5.1 Fast Extract . . . 40
  - 3.5.2 Re-substitution . . . 42

3.5.3 Full_Simpfily . . . 43
3.5.4 Synthesis Flow . . . 44
3.6 Experimental Results . . . 44
3.6.1 Discussion of Results . . . 47
3.7 Conclusions . . . 47

**4 Integrating Logic Synthesis and Placement** **48**
4.1 Design Flow . . . 48
4.2 Previous Work . . . 52
4.2.1 Pre-Layout Interconnect Estimation . . . 52
4.2.2 Post-Layout Optimization . . . 53
4.2.3 Integrated Logic Synthesis and Layout . . . 53
4.3 Net Topology and Interconnect Delay Model . . . 54
4.3.1 Semi-Perimeter Estimate . . . 54
4.3.2 Steiner Tree Estimate . . . 54

**5 Global Placement** **57**
5.1 Quadratic Placement . . . 58
5.2 Kraftwerk Algorithm . . . 59
5.3 Implementation of Kraftwerk Algorithm . . . 61
5.3.1 Conjugate Gradient . . . 62
5.3.2 Net Weights . . . 62
5.3.3 Discretization . . . 62
5.3.4 Poisson Equation . . . 63
5.3.5 Dimensionless Cells . . . 64
5.3.6 Iteration Control . . . 64
5.3.7 Incremental Kraftwerk Algorithm . . . 65

**6 Technology Dependent Optimization** **68**
6.1 Local Placement . . . 69
6.2 Technology Decomposition . . . 70
6.3 Technology Mapping . . . 75
6.3.1 Area and Wire-Length Minimization . . . 76
6.3.2 Updating Placement . . . 79
6.3.3 Delay Optimization . . . 82
6.3.3.1 Fixed Load Method . . . 83
6.3.3.2 Single Match Method . . . 84
6.3.3.3 Multiple Match Method . . . 85
6.4 Experimental Results . . . 88
6.4.1 Delay Correlation . . . 88
6.4.2 Area and Wire-Length Minimization . . . 89
6.4.3 Delay Optimization . . . 93
6.5 Conclusions and Future Work . . . 93

**7 Technology Independent Optimization** **101**
7.1 Preliminaries . . . . . 102
7.1.1 Value of a Kernel . . . . . 102
7.1.2 Placement Interaction . . . . . 102
7.2 Wired Kernel Extraction . . . . . 103
7.3 Wire-planned Kernel Extraction . . . . . 103
7.3.1 Weight of a Kernel . . . . . 104
7.3.2 Wire-planned Duplication . . . . . 105
7.4 Experimental Results . . . . . 106
7.4.1 Area and Wire-Length Minimization . . . . . 106
7.4.2 Delay Optimization . . . . . 107

**8 Conclusions and Future Research** **115**
8.1 Conclusions . . . . . 115
8.2 Future Work . . . . . 117

**Bibliography** **118**

# List of Figures


1.1 Buffer delay model. . . . . 6
1.2 Actual vs estimated net length (industry2). . . . . 11
1.3 Actual vs estimated net length (industry3). . . . . 12
1.4 Actual vs estimated net length (avq.small). . . . . 12
1.5 Actual vs estimated net length (avq.large). . . . . 13

2.1 A Boolean network. . . . . 18
2.2 Arrival and required time. . . . . 24

3.1 Network $\mathcal{N}_{\min}$ and its optimal placement. . . . . 28
3.2 Network $\mathcal{N}'$ and its optimal placement. . . . . 28
3.3 Network $\mathcal{N}''$ and its optimal placement. . . . . 29
3.4 Legal region of node $z$. . . . . 30
3.5 Conflicting legal region requirements for $x$. . . . . 31
3.6 Regions and labels of regions. . . . . 32
3.7 Region intersection for node $z$ of $\mathcal{N}'$. . . . . 34
3.8 Figure for proof of Lemma 3.2. . . . . 36
3.9 Fast extract example. . . . . 41
3.10 (a) Pin positions of fast extract example. (b) Legal region of node $n$. . . . . 42
3.11 Number of literals vs number of nodes legalized for C1355. . . . . 46

4.1 Conventional design flow. . . . . 50
4.2 Integrated logic synthesis and placement design flow. . . . . 51
4.3 Semi-perimeter estimate of a net. . . . . 55
4.4 Topology model of a net. . . . . 56
4.5 Delay model of a net. . . . . 56

5.1 Working area, placement area, and grid points. . . . . 63
5.2 Weight schedule. . . . . 65
5.3 Illustration of spreading phase of Kraftwerk. . . . . 66
5.4 Execution of Kraftwerk. . . . . 67

6.1 Incremental placement. . . . . 69
6.2 Technology Decomposition and Fanin Ordering Problem. . . . . 72

6.3 Angle Ordering Solution. . . . . 74
6.4 Furthest Pair Solution. . . . . 75
6.5 Definition of Cost Elements. . . . . 78
6.6 Creation of Boolean network for placement. . . . . 80
6.7 Removing parallel inverters. . . . . 81
6.8 Single match method. . . . . 85
6.9 Delay models and maximum computation of piece-wise linear functions. . . . . 87
6.10 Delay and area variation with respect to $\alpha$, the wire cost coefficient. . . . . 92

7.1 Before and after extracting kernel $k$ in *wired* kernel extraction. . . . . 104
7.2 Primary output and fanout angles in wire-planned kernel extraction. . . . . 105

# List of Tables


1.1 Strawman technologies . . . . . 5
1.2 Critical length and delay of strawman technologies. . . . . 7
1.3 Critical count of $0.25\mu$ and $0.10\mu$ technologies. . . . . 8
1.4 Extrapolated critical length ($l_{crit}$) and critical count ($k_{crit}$) for $0.05\mu$ technology. . . 8
1.5 Gate delay vs interconnect delay for $\tau_{crit}$. . . . . 9
1.6 wire-load model. . . . . 10
1.7 Circuits. . . . . 10
1.8 Number of nets below and above than the wire-load estimated values. . . . . 14
1.9 Percentages of nets below and above than the wire-load estimated values. . . . . 15

3.1 Path length comparison of *script.rugged*, *script.delay*, and *script.wire*. . . . . 45
3.2 CPU time comparison of *script.rugged*, *script.delay*, and *script.wire*. . . . . 45
3.3 Delay comparison of *script.rugged*, *script.delay*, and *script.wire*. . . . . 45

6.1 Estimated delay vs actual delay using wire-load model and our approach. . . . . 89
6.2 Area and wire-length minimization. . . . . 90
6.3 Area of area and wire-length minimization (in $\mu^2$) . . . . . 91
6.4 Fixed load delay minimization method. . . . . 94
6.5 Area of the fixed load delay minimization method. (in $\mu^2$) . . . . . 95
6.6 Single match delay minimization method. . . . . 96
6.7 Area of the single match delay minimization method. (in $\mu^2$) . . . . . 97
6.8 Multiple match delay minimization method. . . . . 98
6.9 Area of multiple match delay minimization method. (in $\mu^2$) . . . . . 99

7.1 Comparing interconnect delay for different kernel extraction algorithms in area minimization (delays in ps). . . . . 108
7.2 Comparing total delay for different kernel extraction algorithms in area minimization (delays in ps). . . . . 109
7.3 Comparing area for different kernel extraction algorithms in area minimization (in $\mu^2$). . . . . 110
7.4 Comparing interconnect delay for different kernel extraction algorithms in delay minimization (delays in ps). . . . . 112
7.5 Comparing total delay for different kernel extraction algorithms in delay minimization (delays in ps). . . . . 113

7.6 Comparing area for different kernel extraction algorithms in delay minimization (in $\mu^2$). . . . . 114

## Acknowledgements

My life as a graduate student at Berkeley has been an experience that I will treasure forever. It has been made possible in great part by the support and encouragement of wonderful family members, professors and friends.

I am indebted to my parents for their support and encouragement not only during my Ph.D. program at Berkeley, but also my M.S. program at Ames and my B.S. program at Lafayette. My father taught my siblings and I the value of education since we were very young. He always reminds us that a good education is hard to come by and we should look for it and seize the opportunity when one comes around. His tenacity, wisdom, and perseverance have been a constant inspiration and guidance for me. My mother has been a source of support and comfort in hard times and a place of sharing in good times.

The person that sacrifices the most during my program at Berkeley has been my wife. She has been supportive and understanding throughout my program. I owe her for her warmth, patience, and understanding and I thank her for her support and for being my best friend.

I am grateful to my advisor Professor Alberto Sangiovanni-Vincentelli for giving me the opportunity to learn from the best in research in general and in CAD in particular. I thank him for allowing me to explore different research avenues, and his guidance in my research. He never stops to amaze me with his quickness in grasping my presentation. He is often a step ahead of what I say. His unmatched level of energy fascinates me and often inspires me to do my best. Although Professor Robert Brayton is not my advisor officially, he is like a second advisor to me and I am grateful for that. Bob's meticulous and perseverance attitude in research is something I admire greatly. I would also like to thank Professor Richard Newton for his interests in my research and for his advices in my career search. I have been impressed with his enthusiasm towards the betterness of not only his students but also of other students. I am grateful to Professor Steven Evans, who together with Professor Richard Newton, served on my qualifying examination committee.

I have been fortunate to have interacted with many wonderful fellow students. I enjoyed the company of Marlene Wan, Min Zhou, Shaz Qadeer, and Alok Agrawal during my early years at Berkeley. Amit Narayan amazed me with his ability to quickly grasp ideas and his practical views of life. I was fortunate to have interacted with Sunil Khatri. I admire his attention to details and his wisdom and I enjoyed greatly his company. I enjoyed chatting with Harry Hsieh and Sriram Krishnan in various topics. Yuji Kukimoto's practical views of research impressed me. I was fortunate to have worked with and learned from Rajeev Ranjan and Jagesh Sanghavi. I enjoyed the

interesting and sometimes animated discussions with Luca Carloni and Amit Mehrotra. I enjoyed the company of Lixin Su. I acknowledge Edoardo Charbon's helpfulness with obtaining licenses for the CADENCE software which I use in my experiments. I was fortunate to have interacted with visiting scholars Eugene Goldberg and Andreas Kuehlmann. I learned a lot about research and work in the industry from them.

I would also like to thank Semiconductor Research Corporation (SRC) and California MICRO program for supporting my research at Berkeley.

# Chapter 1

# Introduction

A typical design flow for an integrated digital circuit starts with a functional description of the circuit written in a high-level hardware description language like Verilog and a set of constraint specifications. The objective of the design process is typically to minimize a cost function[1] under constraints which could include delay, area, power, etc. The functional description is first passed to a *logic synthesis* tool to generate an optimized *logic* circuit meeting the constraints according to some cost model. The logic circuit is then placed and routed on a two-dimensional plane by a *placer* and a *router* minimizing the cost function while meeting the constraints according to their cost model. Although a design can be carried strictly by hand without using any of these *computer-aided design* (*CAD*) tools, no design today is carried out without any help of CAD tools.

Over the past several years, two major trends have been shaping up to push the envelope of the typical *design methodology* and current CAD tools. The first trend is the ever increasing pressure of *time-to-market* considerations. This has put a tremendous amount of pressure on designers to rely more heavily on CAD tools. Yet with the decreasing feature size of technology, certain effects, like increasing wire delay due to an increase in wire *resistance* are becoming increasingly important. The second trend is the increasing amount of *integration* into a single chip, especially with the explosion of the need for networking chips to support the internet infrastructure. This increase in integration also means that designers will have to rely more heavily on CAD tools to handle the increased design complexity. Although these two trends mean that there will be an increasing need for CAD tools, it also means that some assumptions and abstractions that CAD tools have been operating on will have to be revisited. In particular, the assumption that gates in a circuit contribute to the majority of the delay of the circuit needs to be re-evaluated. Also, the increasingly large

[1] A typical cost function include area, delay, and power.

number of iterations between logic synthesis and physical synthesis, which is typically called the *timing closure* needs to be studied.

This thesis addresses the increasing importance of wire delay in logic synthesis and the timing closure problem.

## 1.1 Thesis Outline

The remaining of this chapter describes the technology trends and their implications to the CAD problems. Several technologies ranging from $0.25\mu$ to $0.05\mu$ minimum feature sizes obtained after several feedback iterations from the industry is presented. The technologies are used throughout this thesis. The questions of interconnect delay contribution to the total delay are analyzed and studied, together with the size of a module that can safely be designed without accounting for interconnect resistance. The problem of a large number of iterations that need to be performed in the industry, often referred to as the *timing closure* problem is discussed. The interconnect delay estimates commonly used in logic synthesis called the *wire-load model* is analyzed and discussed.

In Chapter 2, background on logic synthesis and terminologies are reviewed. Basic definitions about logic functions are stated. Algebraic and Boolean division operations which play a major role in logic synthesis are discussed. Logic optimization operations based on algebraic division are defined. The mapping procedure from logic functions to logic gates is described. This chapter also defines the delay model with and without interconnect. The topological timing analysis used to compute the delay of the circuit is described.

Conventional logic synthesis minimizes the number of literals in the circuit when synthesizing a logic circuit. The number of literals is a measure of the effectiveness of the synthesis. Chapter 3 describes our approach that takes the diametrically opposing view which minimizes the interconnect length. The proposed approach is called the *wire-planning* approach. With this perspective, the characteristics of when a logic circuit is easier to place than another is determined. Based on the characteristics of good circuits, the notion of legal nodes is introduced. Intuitively, a legal node is a cell or a group of cells, which if found, can produce an easy-to-place circuit. The notion of legality of a node is then extended to the whole circuit, which says that if every node is legal, then circuit is easy to place. The wire-planning approach forms the basics of several heuristics in the approaches described in later chapters of this thesis.

Chapter 4 describes a practical approach of accounting for interconnect delay in logic synthesis. It is an approach that integrates logic synthesis and global placement procedures. This

chapter describes the design flow of the proposed approach. A literature survey of related work is described here. Models used in the approach like net topology and interconnect delay model are described here. The detail of the approach is described in the remaining chapters of this thesis.

The integration of logic synthesis approach uses a placement algorithm called *Kraftwerk* which is discussed in Chapter 5. Kraftwerk is a mixed quadratic and force-directed placement algorithm. Its suitability for the integrated approach is discussed, followed by the detail of the algorithm. Kraftwerk implementation uses numerical computation methods like conjugate gradient, and Fast Fourier Transform. It is clear then that the algorithm is iterative. How to implement and control the behavior of the algorithm in the work of this thesis is described in detail in this chapter.

As in the conventional logic synthesis, the proposed approach is separated into two phases: the technology independent phase and the technology dependent phase. Technology dependent phase is closer to the final implementation of the circuit being synthesized in logic gates. This means that the gate and interconnect delays computed during this phase are closer to the actual delay of the final logic circuit. Because of this, this phase is presented in Chapter 6 before the technology independent phase. The other reason is that the technology independent optimization can then use the algorithms in the technology dependent phase to measure more accurately the performance of the circuit. Two different optimization algorithms are presented: the area and wire-length minimization, and the delay optimization algorithms. In the area and wire-length minimization, an attempt is made to reduce the area and wire-length of the circuit simultaneously. In the delay optimization, three different approaches corresponding to how the load of a gate is approximated are described and compared.

In Chapter 7, the technology independent optimization phase of the proposed integrated approach is described. This phase includes extracting common sub-expressions in the logic functions that describe the circuit such that the sizes of the logic functions, measured in the number of literals in them, are reduced. Depending on how the value of a sub-expression of a node is evaluated, two different algorithms are introduced and compared. In one algorithm, the value of the sub-expression is computed directly using the structure of the circuit and the positions of their components. In the other algorithm, a heuristic based on the wire-planning approach is used to reduce the interconnect delay contribution to the circuit. As a result of the wire-planning based approach, a duplication operation is introduced. It duplicates logic when the interconnect delay is likely to decrease.

Finally, this thesis will be concluded in Chapter 8. The summary of the thesis and future directions for research are given.

## 1.2 Technology and Design Trend

In the present day IC design, the ever-shrinking time-to-market trend has put pressure on designers to use CAD tools to increase productivity. A company can no longer design an entirely custom chip because the penalty of late introduction of the product into the market has often meant failure for the product. At the same time, design quality is ever important. Further, there is the ability to integrate more and more functionality on a single IC than ever before. Hence, not only do CAD tools need to be faster, they also need to effective.

The time-to-market pressure and increasing scale of integration have led CAD researchers to predict the future of technology and re-examine the assumptions and abstractions that have been valid thus far. This section looks into two particular problems: the effect of wire delay as technology scales to smaller feature sizes, and the timing closure problem which will be aggravated by larger scales of integration.

### 1.2.1 Technology Models Used in this Thesis

The analysis and experiments in this thesis uses the "strawman" technologies developed in [KMB+99]. The strawman technologies are a set of parameters for technologies with minimum feature sizes ranging from $0.25\mu$ to $0.05\mu$. They are based on the parameter values predicted by the the Semiconductor Industrial Alliance's National Technology Roadmap for Semiconductor for 1997 (SIA NTRS) [Ass97] with feedbacks from leading semiconductor companies like IBM and Motorola.

The technology parameters are shown in Table 1.1.

### 1.2.2 Technology Scaling

The 1997 SIA NTRS [Ass97] predicts that interconnect delay will start dominating the total gate delay as we move down to $0.15\mu$ technology and below. This is true mainly for global wires (which are wires that are used to connect different blocks of design) as opposed to local wires (which are used to interconnect cells in a block). In this section, we quantify this claim, showing that interconnect delay will become significant due to technology scaling even within a relatively small module.

For CMOS circuits, a gate driving another gate through a homogeneous wire of length $l$ can be modeled by the circuit in Figure 1.1 [OB98]. The voltage source $v_{tr}$ is controlled by the

Table 1.1: Strawman technologies

| Process ($\mu$) | | 0.25 | 0.18 | 0.13 | 0.10 | 0.07 | 0.05 |
|---|---|---|---|---|---|---|---|
| $V_{DD}$ (V) | | 2 | 1.8 | 1.5 | 1.2 | 0.9 | 0.6 |
| $L_{eff}$ (nm) | | 160 | 100 | 70 | 50 | 35 | 25 |
| $t_{ox}$ (Å) | | 60 | 45 | 35 | 30 | 20 | 12 |
| Levels | | 6 | 6 | 7 | 8 | 9 | 9 |
| Poly | H ($\mu$) | 0.2 | 0.15 | 0.13 | 0.1 | 0.07 | 0.07 |
| | W ($\mu$) | 0.25 | 0.18 | 0.13 | 0.1 | 0.07 | 0.05 |
| | space ($\mu$) | 0.25 | 0.18 | 0.13 | 0.1 | 0.07 | 0.05 |
| | sheet $\rho$ ($\Omega/\square$) | 4 | 5.3 | 6.2 | 8 | 11.4 | 11.4 |
| M1–2 | H ($\mu$) | 0.5 | 0.46 | 0.34 | 0.26 | 0.2 | 0.14 |
| | W ($\mu$) | 0.30 | 0.23 | 0.17 | 0.13 | 0.1 | 0.07 |
| | space ($\mu$) | 0.30 | 0.23 | 0.17 | 0.13 | 0.1 | 0.07 |
| | sheet $\rho$ ($\Omega/\square$) | 0.044 | 0.048 | 0.065 | 0.085 | 0.11 | 0.16 |
| | $t_{ins}$ (nm) | 650 | 500 | 360 | 320 | 270 | 210 |
| M3–4 | H ($\mu$) | 2.0 | 2.0 | 1.2 | 1.0 | 0.6 | 0.6 |
| | W ($\mu$) | 1.0 | 1.0 | 0.6 | 0.5 | 0.3 | 0.3 |
| | space ($\mu$) | 1.0 | 1.0 | 0.6 | 0.5 | 0.3 | 0.3 |
| | sheet $\rho$ ($\Omega/\square$) | 0.011 | 0.011 | 0.018 | 0.0224 | 0.036 | 0.036 |
| | $t_{ins}$ (nm) | 900 | 900 | 900 | 900 | 900 | 900 |
| M5–6 | H ($\mu$) | 2.5 | 2.5 | 2.0 | 2.0 | 1.5 | 1.5 |
| | W ($\mu$) | 2.0 | 2.0 | 1.0 | 1.0 | 0.75 | 0.75 |
| | space ($\mu$) | 2.0 | 2.0 | 1.0 | 1.0 | 0.75 | 0.75 |
| | sheet $\rho$ ($\Omega/\square$) | 0.009 | 0.009 | 0.011 | 0.011 | 0.015 | 0.015 |
| | $t_{ins}$ (nm) | 1400 | 1400 | 900 | 900 | 900 | 900 |
| M7–8 | H ($\mu$) | – | – | 2.5 | 2.5 | 2.4 | 2.4 |
| | W ($\mu$) | – | – | 2.0 | 2.0 | 1.2 | 1.2 |
| | space ($\mu$) | – | – | 2.0 | 2.0 | 1.2 | 1.2 |
| | sheet $\rho$ ($\Omega/\square$) | – | – | 0.009 | 0.009 | 0.0094 | 0.0094 |
| | $t_{ins}$ (nm) | – | – | 1400 | 1400 | 900 | 900 |
| M9 | H ($\mu$) | – | – | – | – | 2.5 | 2.5 |
| | W ($\mu$) | – | – | – | – | 2.0 | 2.0 |
| | space ($\mu$) | – | – | – | – | 2.0 | 2.0 |
| | sheet $\rho$ ($\Omega/\square$) | – | – | – | – | 0.009 | 0.009 |
| | $t_{ins}$ (nm) | – | – | – | – | 1400 | 1400 |
| Via (M1–M2) | size ($\mu$) | 0.5 | 0.36 | 0.26 | 0.2 | 0.14 | 0.1 |
| | R ($\Omega$) | 0.46 | 0.69 | 0.95 | 1.43 | 2.16 | 3.27 |
| $\kappa$ | | 3.3 | 2.7 | 2.3 | 2 | 1.8 | 1.5 |

voltage $v_{st}$ stored at its input capacitance. It switches instantaneously once $v_{st}$ reaches $x$ fraction of the supply voltage. The delay between when $v_{tr}$ switches and when the voltage at the receiver reaches $x \times 100\%$ of the supply voltage is described by the following equation [Bak90]:

$$\tau = b(x)R_{tr}(C_L + C_p) + b(x)(R_{tr}c + rC_L)l + a(x)rcl^2 \tag{1.1}$$

where $a(x)$ and $b(x)$ are constants, $C_p$ is the parasitic (diffusion) capacitance of the driver, $C_L$ is the gate capacitance of the receiver, and $r$ and $c$ are resistance and capacitance per unit length of the wire. For $x = 0.5$, $a = 0.4, b = 0.7$, and for $x = 0.9$, $a = 1.0, b = 2.3$. For this thesis, we will use $x = 0.5$.

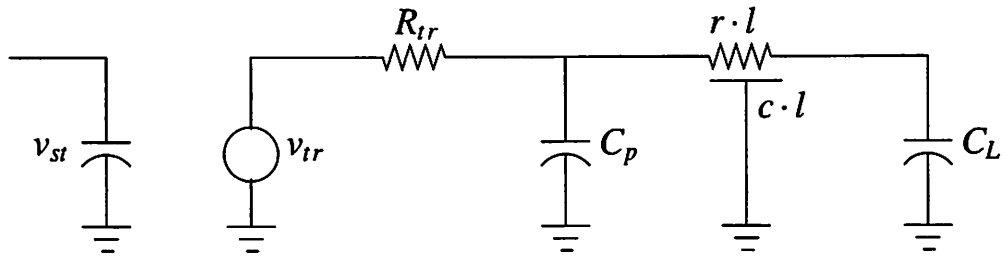



Figure 1.1: Buffer delay model.

If the homogeneous line is too long, then buffers can be inserted to minimize the delay. A *segment* is defined as the part of the buffer inserted circuit from the input of a buffer to the input of the next buffer. The work in [OB98] finds that for this model, the minimum overall delay is achieved when $n_{opt}$ buffers are inserted. The optimum length of each section is

$$l_{crit} = \frac{l}{n_{opt}} = \sqrt{\frac{\frac{b}{a} r_0 c_0 \left(1 + \frac{C_p}{c_0}\right)}{rc}} = \frac{P}{\sqrt{rc}}$$

It is also shown in [OB98] that the delay of each segment only depends on the device parameters and not interconnect parameters. As a consequence, this delay is the same for all metal layers. The critical delay $\tau_{crit}$, which is the delay of a section is described by

$$\tau_{crit} = 2br_0c_0\left(1 + \frac{P}{\sqrt{r_0c_0}}\right)$$

#### 1.2.2.1 Predicting Sizes of Synthesized Blocks

Using the strawman technology parameters, Table 1.2 lists the values of $l_{crit}$ and $\tau_{crit}$ for $0.25\mu$ and $0.10\mu$ technologies. The numbers have been conveniently expressed in terms of feature

size units. Hence, direct comparison of the numbers in the two technologies can be done.

Table 1.2: Critical length and delay of strawman technologies.

| Critical parameter | Feature size units | |
|---|---|---|
| | $0.25\mu$ | $0.10\mu$ |
| $l_{crit}$ (metal 1) | 10,440 | 6,757 |
| $l_{crit}$ (metal 2) | 10,600 | 7,162 |
| $l_{crit}$ (metal 3) | 36,000 | 43,446 |
| $l_{crit}$ (metal 4) | 38,400 | 45,135 |
| $l_{crit}$ (metal 5) | 63,200 | 64,932 |
| $l_{crit}$ (metal 6) | 62,000 | 56,892 |
| $l_{crit}$ (metal 7) | — | 97,581 |
| $l_{crit}$ (metal 8) | — | 93,378 |
| $\tau_{crit}$ | 205 ps | 80 ps |

As seen in Table 1.2, the critical lengths for metal layer 1 and metal layer 2 are smaller for $0.25\mu$ as compared to $0.10\mu$. If we assume that the length $l_{crit}$ is the length of a side of a square block/module, which we call a *critical square*, we can then compute the number of cells that can be placed in that square. We call this number the *critical count*. We would like to find out if interconnect delay is negligible within this critical square for both $0.25\mu$ and $0.10\mu$ technologies, and also study and how it scales in the future. Hence, we are only interested in what is typically called a *module* in which only metal layer 1 and 2 are used for routing.

A minimum size inverter typically has an equivalent $R_{tr}$ of about $10k\Omega$. Such an inverter is not widely used in a design. A typical cell is usually a 2-input NAND gate, as suggested in recent study by Sylvester and Keutzer [SK98]. This typical gate has $R_{tr} \approx 1k\Omega$ and $\frac{W_n}{L_n} \approx 20$. This device has a dimension of about $80 \times 10$ or 800 feature size units. We believe using this 2-input NAND gate to compute the critical counts is reasonable. The reason for this is that most of the cells in a design are larger than this NAND gate (for speed purposes) and there are typically larger cells as well (like latches) in a design.

Using the values in Table 1.2 and a typical 2-input NAND gate size, the critical count for metal layer 1 and 2 are shown in Table 1.3. The formula used to compute the critical count ($k_{crit}$) is $k_{crit} = l_{crit}^2/800$.

The size of a block that is typically synthesized today is about 3,000 to 4,000 cells. The $k_{crit}$ values in Table 1.3 are much larger. Hence, no buffers need to be added. However, the reason that larger blocks are not synthesized is not that state-of-the-art logic synthesis tools cannot handle

Table 1.3: Critical count of $0.25\mu$ and $0.10\mu$ technologies.

| Layer | $k_{crit}$ | |
|---|---|---|
| | $0.25\mu$ | $0.10\mu$ |
| Metal 1 | 136,242 | 57,071 |
| Metal 2 | 140,450 | 64,117 |

larger designs. In fact, state-of-the-art logic synthesis tools can handle much larger blocks. The reason is that the delay computed by logic synthesis tools is too inaccurate to be useful [She98]. These 3,000 to 4,000 cell blocks are typically part of a larger functional block. Functional blocks of 10,000 to 30,000 cells are commonly designed today. Once synthesized, these blocks of 3,000 to 4,000 cells are then placed and routed as a single block. Now, if we attempt to synthesize larger functional blocks directly, then the length $l_{crit}$ will fall within a synthesized block within the next few process generations. For the lack of a good SPICE model for the $0.05\mu$ technology, we linearly extrapolate the $l_{crit}$ values for metal layer 1 and 2 into this technology[2]. The results of this extrapolation are shown in Table 1.4. It is clear that for $0.05\mu$ technology, $k_{crit}$ is close to the size of a typical functional block designed today. Since there is a continuous trend towards larger scales of integration, the size of a synthesized functional block could very well exceed the numbers shown in this table.

Table 1.4: Extrapolated critical length ($l_{crit}$) and critical count ($k_{crit}$) for $0.05\mu$ technology.

| Layer | $l_{crit}$ | $k_{crit}$ |
|---|---|---|
| Metal 1 | 5,467 | 37,346 |
| Metal 2 | 6,049 | 45,738 |

#### 1.2.2.2 Predicting Interconnect Delay

In addition to the values of $k_{crit}$, we are interested in the portion of $\tau_{crit}$ that is contributed by interconnect delay. We simulate a ring oscillator circuit using SPICE to compute the gate delay. We note that this gate delay varies very little with the size of the buffer in a ring oscillator because the increase in the drive strength is balanced by the increase in the input capacitance of the next stage. As seen from Table 1.5, the interconnect delay dominates the gate delay for each section of

[2]Although there are concerns in the research community that we may not be able to scale further. Prof. Chenming Hu claimed that with the advances of silicon-on-insulator and double gate devices, the speed of devices should scale according to Moore's law until the year 2025 [Hu99]

an optimally buffer line. This means that interconnect delay starts dominating gate delay well before $l_{crit}$ is reached. The contribution of interconnect delay becomes larger if we consider that when a driver drives a long line followed by a receiver, the driver is usually sized up and the receiver sized down to reduce the delay. This has the effect of reducing the delay contribution of gates and therefore increasing the delay contribution of the interconnect. In other words, the term $R_{tr}(C_L+C_p)$ of Equation 1.1 decreases quadratically, $(cR_{tr}+rC_L)l$ decreases linearly, and $rcl^2$ remains the same due to sizing.

Table 1.5: Gate delay vs interconnect delay for $\tau_{crit}$.

| | | Gate delay | | Interconnect Delay | |
|---|---|---|---|---|---|
| Technology | $\tau_{crit}(ps)$ | Value (ps) | Percent | Value (ps) | Percent |
| $0.25\mu$ | 205 | 53 | 26% | 152 | 74% |
| $0.10\mu$ | 80 | 20 | 25% | 60 | 75% |

This discussion motivates the need for logic synthesis technologies that optimizes not only for gate delay, but also for wire delay.

### 1.2.3 The Timing Closure Problem

Using a conventional design flow, it is becoming increasingly difficult to predict interconnect delay during logic synthesis. This results in a large number of iterations between logic synthesis, physical synthesis, parasitic extraction, and timing analysis. This is commonly known as the *timing closure* problem. These iterations are time consuming.

During logic synthesis, the timing behavior of a gate is generally characterized by its worst case behavior. This can be done accurately by using a circuit simulation tool because the layout of the gate is known. On the other hand, interconnect layout information is not available while performing logic synthesis. To estimate the contribution of interconnect to delay, a *wire-load model* is typically used. A wire-load model is a statistical estimate of the length of a net given the number of cells connected to the net. From this length estimate, the capacitance and resistance of the net are computed. Hence, all nets with the same fanout count are estimated to be of the same length during conventional logic synthesis.

To determine the validity of such a wire-load model, we conduct an experiment to study the correlation between the length of a net computed using the wire-load model and the actual length of the net after global and detailed placement. Since the delay of a net is directly proportional to its

length, we will be using net length as a measure.

The wire-load model used here is listed in Table 1.6 for nets with 2 to 10 pins. For nets with more than 10-pins, net length is extrapolated by a line with slope equal to 1.6. This wire-load model is consistent with an industrial $0.18\mu$ process technology. This table is interpreted as follows. If the length of a net with 2 pins is $l(2)$, then the length of a net with 3 pins is estimated to be $3 \cdot l(2)$. For a net with 11 pins, its length is estimated to be $(27 + 1.6)l(2)$. For our $0.1\mu$ strawman technology and the size of the circuits that we run our experiments on, $l(2)$ is set to be $55\mu$.

Table 1.6: wire-load model.

| # Pins | Multiplier | Length |
|---|---|---|
| 2 | 1 | $55\mu$ |
| 3 | 3 | $165\mu$ |
| 4 | 7 | $385\mu$ |
| 5 | 11 | $605\mu$ |
| 6 | 15 | $825\mu$ |
| 7 | 19 | $1045\mu$ |
| 8 | 22 | $1210\mu$ |
| 9 | 25 | $1375\mu$ |
| 10 | 27 | $1485\mu$ |

In order to evaluate the effectiveness of the wire-load model, we compute the net length using the wire-load model and compare them to the distribution of actual net length after global and detail placement. The actual length of a net is computed using its half-perimeter estimate. We use GORDIAN [KSJA91] as the global placer and DOMINO [DJS91] as the detailed placer. We run both GORDIAN and DOMINO on the four large circuits from the 1992 Layout Synthesis benchmark set: *industry2*, *industry3*, *avq.small*, and *avq.large*.

Table 1.7 shows information for all 4 circuits used in this experiment.

Table 1.7: Circuits.

| Circuit | # cells | # nets |
|---|---|---|
| industry2 | 12637 | 13420 |
| industry3 | 15433 | 21968 |
| avq.small | 21918 | 30039 |
| avq.large | 25178 | 33299 |

From the results of global and detailed placement for the four circuits, we analyze the

distribution of net length for different nets.. Figures 1.2–1.5 show the scatter plot of the actual net length superimposed with the net length estimated using the wire load model. The $x$-axis represents net size in terms of the number of pins on the net. The $y$-axis represents the length of each net in microns. The dashed lines represent the net length estimated using the wire-load model.

Figures 1.2–1.5 show that many small nets (i.e. nets with few pins) are not accurately estimated. For example, the length of 2-pin nets range from $0.5\mu$ to $1500\mu$ for *industry2*. The wire load model estimates it to be $55\mu$. Large nets are pessimistically estimated. However, a pessimistic estimation is necessary for large nets because they typically lie on critical paths. Besides, while a semi-perimeter estimate is exact for 2-pin and 3-pin nets, it is increasingly pessimistic for larger nets.

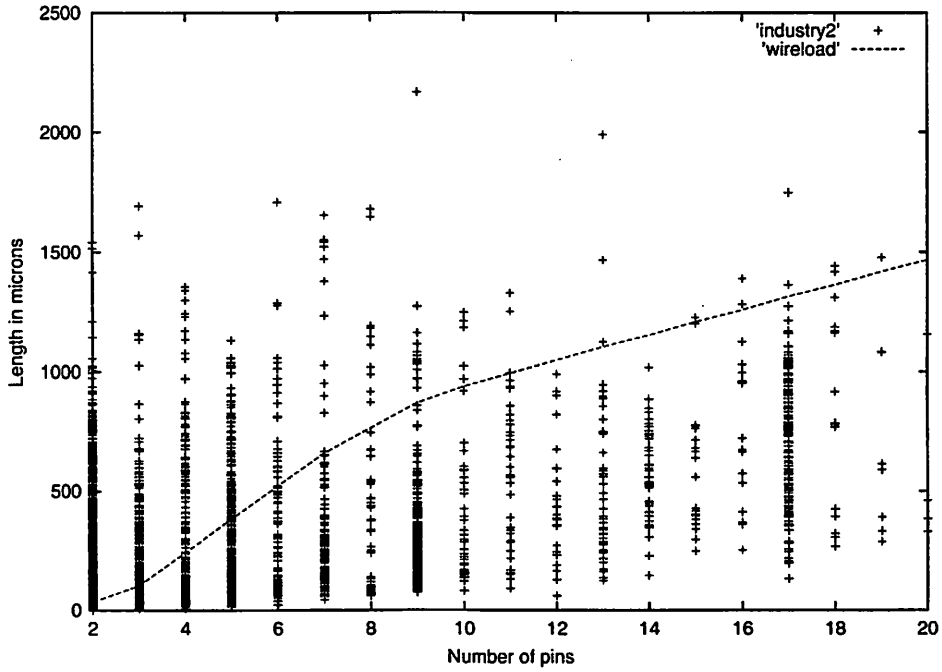



Figure 1.2: Actual vs estimated net length (industry2).

Having demonstrated the inaccuracy of wire-load model, we would like to find out the fraction of nets that are under and over estimated. If only a few nets are under-estimated, then the delay can be optimized by applying gate-sizing and/or buffer insertion. The number of nets whose actual net length that is below and above the estimated length for each circuit are tabulated in Table 1.8 and Table 1.9. Column 1 lists the sizes of nets. The remaining pairs of columns are the data for each circuit. The first column for each circuit contains the number of nets that are below

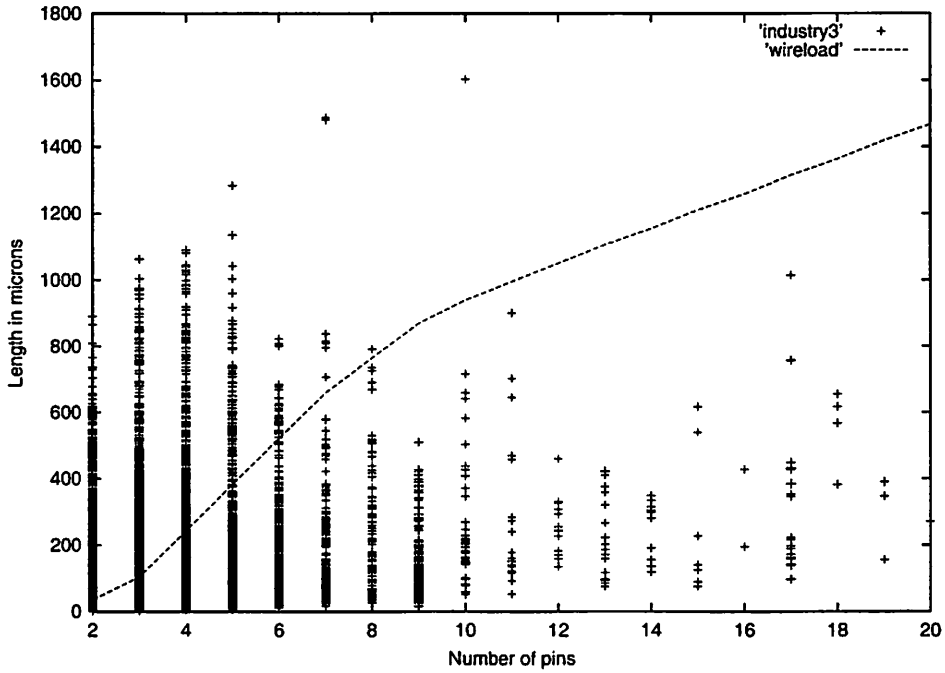



Figure 1.3: Actual vs estimated net length (industry3).

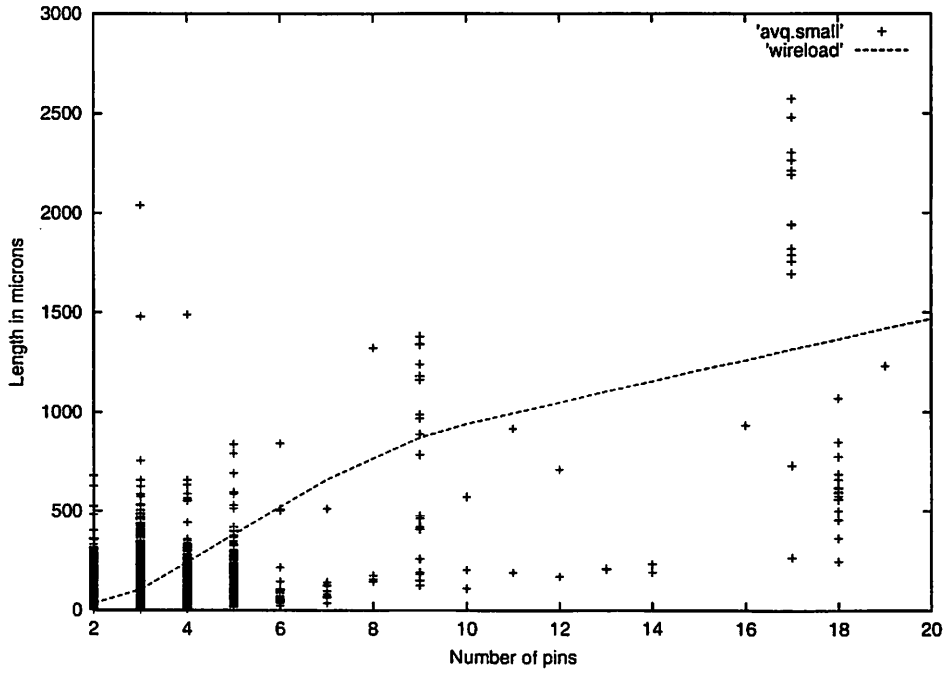



Figure 1.4: Actual vs estimated net length (avq.small).

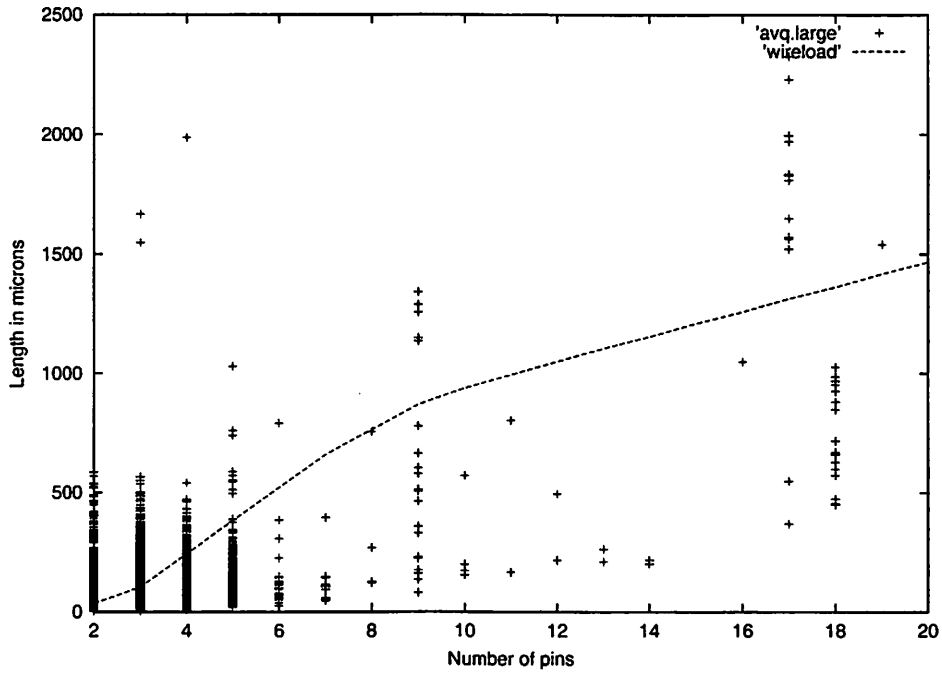



Figure 1.5: Actual vs estimated net length (avq.large).

the estimated length and the second column contains those that are above the estimated length. As seen from Table 1.9, over 20% of small (i.e. $\leq 3$ pin) nets are underestimated. Many of these nets are very long and consist of few (3-pin or 4-pin nets) or no branches (2-pin nets). These nets have higher wire resistance than larger nets, which have more branches, and hence higher RC delay. Since the difference between the actual and estimated length is very large for these nets, nets that are not regarded to be on critical paths by logic synthesis tools can turn out to be critical nets after placement. If the number of such nets is large, as in all these four circuits, gate sizing and buffer insertion may not be enough to meet the timing requirements.

In summary, we have shown that the wire-load model, widely used in state-of-the-art logic synthesis tools, can dramatically underestimate the delay of nets, especially small nets. This results in an inaccurate determination of the critical paths in logic synthesis which can cause the **timing closure** problem. The timing closure problem will likely be aggravated if the current methodology is used to synthesize larger functional blocks.

Table 1.8: Number of nets below and above than the wire-load estimated values.

| Net | industry2 | | industry3 | | avq.small | | avq.large | |
|---|---|---|---|---|---|---|---|---|
| | below | above | below | above | below | above | below | above |
| 2-pin | 6660 | 2747 | 7483 | 3476 | 11694 | 1960 | 14626 | 2288 |
| 3-pin | 1598 | 426 | 5072 | 1040 | 5430 | 706 | 5359 | 777 |
| 4-pin | 203 | 94 | 1458 | 401 | 1422 | 56 | 1409 | 69 |
| 5-pin | 331 | 118 | 1696 | 112 | 702 | 10 | 701 | 11 |
| 6-pin | 93 | 24 | 393 | 34 | 28 | 1 | 28 | 1 |
| 7-pin | 125 | 13 | 194 | 7 | 10 | 0 | 10 | 0 |
| 8-pin | 41 | 11 | 104 | 1 | 3 | 1 | 4 | 0 |
| 9-pin | 279 | 14 | 242 | 0 | 12 | 9 | 15 | 6 |
| 10-pin | 36 | 5 | 46 | 1 | 4 | 0 | 4 | 0 |
| 11-pin | 32 | 3 | 21 | 0 | 2 | 0 | 2 | 0 |
| 12-pin | 27 | 0 | 14 | 0 | 2 | 0 | 2 | 0 |
| 13-pin | 43 | 3 | 18 | 0 | 2 | 0 | 2 | 0 |
| 14-pin | 45 | 0 | 12 | 0 | 2 | 0 | 2 | 0 |
| 15-pin | 19 | 1 | 7 | 0 | 0 | 0 | 0 | 0 |
| 16-pin | 18 | 2 | 3 | 0 | 1 | 0 | 1 | 0 |
| 17-pin | 171 | 2 | 23 | 0 | 2 | 11 | 2 | 11 |
| 18-pin | 14 | 2 | 4 | 0 | 18 | 0 | 18 | 0 |
| 19-pin | 7 | 1 | 4 | 0 | 1 | 0 | 0 | 1 |
| 20-pin | 4 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

Table 1.9: Percentages of nets below and above than the wire-load estimated values.

| Net | industry2 | | industry3 | | avq.small | | avq.large | |
|---|---|---|---|---|---|---|---|---|
| | below | above | below | above | below | above | below | above |
| 2-pin | 71.0 | 29.0 | 68.0 | 32.0 | 86.0 | 14.0 | 86.0 | 14.0 |
| 3-pin | 79.0 | 21.0 | 83.0 | 17.0 | 88.0 | 12.0 | 87.0 | 13.0 |
| 4-pin | 68.0 | 32.0 | 78.0 | 22.0 | 96.0 | 4.0 | 95.0 | 5.0 |
| 5-pin | 74.0 | 26.0 | 94.0 | 6.0 | 99.0 | 1.0 | 98.0 | 2.0 |
| 6-pin | 79.0 | 21.0 | 92.0 | 8.0 | 97.0 | 3.0 | 97.0 | 3.0 |
| 7-pin | 91.0 | 9.0 | 97.0 | 3.0 | 100.0 | 0.0 | 100.0 | 0.0 |
| 8-pin | 79.0 | 21.0 | 99.0 | 1.0 | 75.0 | 25.0 | 100.0 | 0.0 |
| 9-pin | 95.0 | 5.0 | 100.0 | 0.0 | 57.0 | 43.0 | 71.0 | 29.0 |
| 10-pin | 88.0 | 12.0 | 98.0 | 2.0 | 100.0 | 0.0 | 100.0 | 0.0 |
| 11-pin | 91.0 | 9.0 | 100.0 | 0.0 | 100.0 | 0.0 | 100.0 | 0.0 |
| 12-pin | 100.0 | 0.0 | 100.0 | 0.0 | 100.0 | 0.0 | 100.0 | 0.0 |
| 13-pin | 93.0 | 7.0 | 100.0 | 0.0 | 100.0 | 0.0 | 100.0 | 0.0 |
| 14-pin | 100.0 | 0.0 | 100.0 | 0.0 | 100.0 | 0.0 | 100.0 | 0.0 |
| 15-pin | 95.0 | 5.0 | 100.0 | 0.0 | — | — | — | — |
| 16-pin | 90.0 | 10.0 | 100.0 | 0.0 | 100.0 | 0.0 | 100.0 | 0.0 |
| 17-pin | 99.0 | 1.0 | 100.0 | 0.0 | 15.0 | 85.0 | 15.0 | 85.0 |
| 18-pin | 88.0 | 12.0 | 100.0 | 0.0 | 100.0 | 0.0 | 100.0 | 0.0 |
| 19-pin | 88.0 | 12.0 | 100.0 | 0.0 | 100.0 | 0.0 | 0.0 | 100.0 |
| 20-pin | 100.0 | 0.0 | 100.0 | 0.0 | — | — | — | — |

# Chapter 2

# Preliminaries

This chapter introduces the basic terminologies of logic synthesis and timing analysis. Section 2.1 states the definitions of logic functions. Section 2.2 overviews the logic synthesis process. Delay models and topological timing analysis are described in Section 2.3

## 2.1 Logic Functions

**Definition 2.1** *Let $B = \{0,1\}$. An n-input,* **completely specified logic function** *$f$ is a mapping $f : B^n \mapsto B$. Each element in the domain $B^n$ is called a* **minterm** *of $f$. $f^{-1}(1) = \{v \in B^n \mid f(v) = 1\}$ is the* **on-set** *of $f$, and $f^{-1}(0) = \{v \in B^n \mid f(v) = 0\}$ the* **off-set** *of $f$.*

**Definition 2.2** *An n-input,* **incompletely specified logic function** *$f$ is a mapping $f : B^n \mapsto \{0,1,*\}$. $f^{-1}(*) = \{v \in B^n \mid f(v) = *\}$ is the* **don't care set** *of $f$.*

**Definition 2.3** *A* **literal** *is a variable or its complement. A* **cube** *or a* **product term** *is a product or conjunction of one or more literals such that if $x$ appears in the product, $x'$ does not, and vice versa.*

A literal $a$ ($a'$) represents the set of all minterms for which the variable $a$ takes on the value 1 (0). A cube represents the intersection of the sets of minterms represented by all the literals in it. If a variable and its complement are present in a cube, the cube becomes identically 0.

**Definition 2.4** *A* **sum-of-products** *(SOP) is a Boolean sum or disjunction of cubes.*

An SOP represents the union of sets of minterms represented by the cubes in it.

**Definition 2.5** *A* factored form *is defined recursively as follows:*

- *a literal is a factored form*
- *the sum of two factored forms is a factored form, and*
- *the product of two factored forms is a factored form.*

**Definition 2.6** *An n-input, k-output logic function $f$ is a mapping $f : B^n \mapsto B^k$.*

**Definition 2.7** *A* Boolean network $\mathcal{N}$ *is a representation of a multiple-output logic function. It is a directed acyclic graph (DAG), with primary inputs $PI(\mathcal{N})$, primary outputs $PO(\mathcal{N})$, and internal (intermediate) nodes $IN(\mathcal{N})$.* **Primary inputs** *have no incoming arcs and* **primary outputs** *have no outgoing arcs. Associated with each* **internal node** *$i$ is a variable $y_i$ and a representation of a logic function $f_i$. The logic at each node is typically stored as a sum-of-products form. There is an arc from node $i$ to node $j$ in the graph if $j$ uses either $y_i$ or $y_i'$ explicitly in the representation of $f_j$. In that case, $i$ is called a* **fanin** *of $j$, and $j$ a* **fanout** *of $i$. The set of fanins of a node $i$ is denoted as $FI(i)$ and the set of fanouts as $FO(i)$. It there exists a path from node $i$ to node $j$, then node $i$ is said to be a* **transitive fanin** *of $j$, and $j$ a* **transitive fanout** *or $i$. The set of transitive fanins of a node $i$ is denoted as $TFI(i)$, whereas its transitive fanout set is denoted as $TFO(i)$. The* **net** *driven by node $i$ is the set of edges of the type $(i, f^o), f^o \in FO(i)$.*

In this thesis, node $i$ with variable $y_i$ and logic function $f_i$ is synonymously referred to as node $y_i$ or node $f_i$. The net driven by node $i$ is called **net** $i$. Figure 2.1 shows a Boolean network with four primary inputs $a, b, c, d$, primary output $z$, and internal nodes $p, q, r$.

## 2.2 Logic Synthesis

Logic synthesis is a process of reading a *high-level* description of a circuit and generating an implementation of the circuit in terms of logic gates. The synthesis is done for an objective function, such as minimizing area and delay.

Since logic synthesis is a complex process, it is typically divided into two phases: **technology independent optimization phase** followed by **technology dependent optimization phase** [BRSVW87]. The technology independent optimization phase attempts to generate an optimum abstract representation of the circuit according to the objective function. The most commonly used representation of the circuit is a Boolean network and the most commonly used measure is the number of literals of the network in factored form, which is the sum over all the internal nodes of the

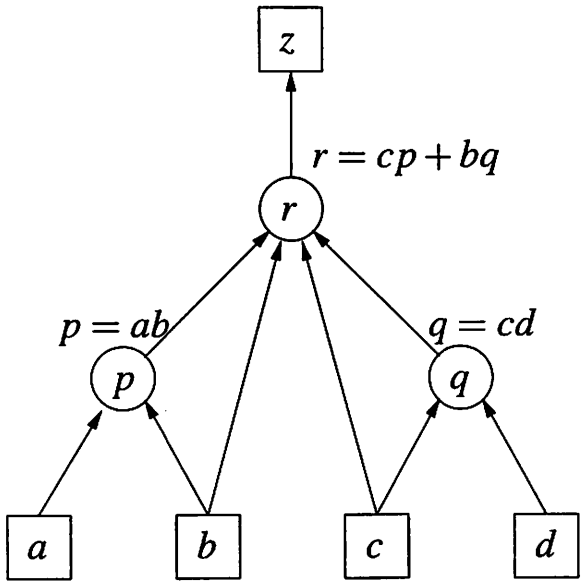



Figure 2.1: A Boolean network.

network of the number of literals in the factored form representation of each node. The dependent optimization phase attempts to implement the optimized Boolean network using a pre-defined library of gates while optimizing the objective function.

### 2.2.1 Technology Independent Optimization

Technology independent optimization is also referred to as *logic optimization* in this thesis. The main idea in this phase is to find sub-functions that can be shared by multiple functions in the Boolean nodes. The basic operation to generate the sub-functions is **division.**

**Definition 2.8** *An* **algebraic expression** *is a sum of products representation of a logic function which is minimal with respect to single cube containment (SCC), i.e. no cube is contained within another cube.*

**Definition 2.9** *The* **product** *of two algebraic expressions $f$ and $g$, $fg$ is a $\sum c_i d_j$ where $\{c_i\} = f, \{d_j\} = g$, made irredundant with respect to single cube containment, e.g. $ab + a = a$. If $f$ and $g$ have disjoint support, it is an algebraic product. Otherwise, it is a Boolean product.*

**Definition 2.10** *Given two functions $f$ and $d$, a* **division** *is an operation that generates a quotient $q$ and a remainder $r$ such that $f = dq + r$. If $dq$ is an algebraic product, then it is called an* **algebraic**

**division**. *Otherwise, it is a* **Boolean division**.

Although Boolean division is more powerful, i.e. it can generate fewer literals in the expression $dq + r$ than an algebraic division can, it is computationally difficult. Because of this, algebraic division is mainly used in logic optimization.

**Definition 2.11** *Given two algebraic expressions $f$ and $d$, a division is called* **weak division**, *denoted by $f/d$, if $f = dq + r$ such that*

1. *$dq$ is algebraic and*
2. *$r$ has as few cubes as possible*

If $n$ is the number of product terms in $f$ and $d$, then weak division can be performed in $O(n \log n)$ [BM82].

**Definition 2.12** *An expression is* **cube-free** *if no cube divides the expression evenly.*

As an example, $ab + c$ is cube-free; while $ab + ac$ and $abc$ are not cube-free. By this definition, a cube-free expression must have more than one cube because a literal in a cube divides the cube evenly.

**Definition 2.13** *The* **primary divisors** *of an expression $f$ are the set of expressions*

$$\mathcal{D}(f) = \{f/c \mid c \text{ is a cube}\}$$

**Definition 2.14** *The* **kernels** *of an expression $f$ are the set of expressions*

$$\mathcal{K}(f) = \{g \mid g \in \mathcal{D}(f) \text{ and } g \text{ is cube free}\}.$$

In other words, the kernels of an expression $f$ are the cube-free primary divisors of $f$.

**Definition 2.15** *A cube $c$ used to obtain the kernel $k = f/c$ is called a* **co-kernel** *of $k$, and $C(F)$ is used to denote the set of co-kernels of $f$.*

Division has been used as the basic operation of logic optimization operations in a Boolean network which include decomposition, extraction, re-substitution, and elimination. Decomposition is the logic operation that expresses a given logic function in terms of simpler sub-functions.

A node associated with each sub-function is added into the Boolean network if it does not already exist. For example, the function

$$f = abc + abd + a'c'd' + b'c'd'$$

can be decomposed into

$$\begin{aligned} f &= xy + x'y' \\ x &= ab \\ y &= c + d \end{aligned}$$

and $x$ and $y$ are added as new nodes if nodes with the same functionality do not already exist. Extraction is the operation that identifies common sub-expressions among different logic functions in the Boolean network. Nodes associated with the sub-expressions are created if they do not already exist. For example, extracting $f$, $g$, and $h$ below

$$\begin{aligned} f &= (a+c)cd + e \\ g &= (a+b)e \\ h &= cde \end{aligned}$$

yields

$$\begin{aligned} f &= xy + e \\ g &= xe \\ h &= ye \\ x &= a + b \\ y &= cd \end{aligned}$$

and $x$ and $y$ are added as new nodes if necessary. Re-substitution is the operation that re-expresses a function $f$ in terms of another function $g$, where both $f$ and $g$ already exist in the Boolean network. For example, if

$$\begin{aligned} g &= a + b \\ f &= ac + bc \end{aligned}$$

then re-substituting $g$ into $f$ yields

$$f = gc$$

Eliminating or collapsing a function $g$ into $f$ is the logic operation that re-expresses a function $f$ without explicitly using $g$. As a result, Boolean node $g$ is removed from the fanin set of $f$. At the

first glance, elimination seems to contradict other logic operations. In fact, it is purposely introduced to get the technology independent optimization out of local minima.

The above logic operations are used to restructure the Boolean network. Another operation which minimizes the logic function stored at each Boolean node is called **node minimization** or **simplification**. It employs two-level logic minimization techniques to minimize each node [BHMSV84]. However, nodes are not minimized independently of other nodes since some flexibility is not used otherwise. The flexibility exists because the fanins of a Boolean node $n$ are related to each other by the nodes of the network in the transitive fanin set of $n$. Hence, the fanins may not be free to take any combination of values. The set of combination of values, which the fanins of $n$ can not take, form the *satisfiability don't cares* (SDC) of $n$. Also, for some combination of values the primary inputs take, the value evaluated at $n$ may not be observable at any primary outputs. In other words, the primary outputs remain unchanged by the change in value at $n$. This set of combination of values is called the *observability don't cares* (ODC) of $n$. In addition, the circuit being synthesized is usually a module in a system. For certain combination of values the primary outputs take, the behavior of the system does not change. This set of combination of values is called the *external don't cares* (XDC).

The set of all SDCs and ODCs of a node is typically large and cannot be efficiently computed. In that case, a suitable subset of both SDCs and ODCs, together with the XDCs, are used as don't cares in the two-level logic minimization of the node [Sav92].

After restructuring and simplification, the Boolean network is optimized and the next step is the technology dependent phase or also called the mapping phase.

### 2.2.2 Technology Dependent Optimization

The optimized Boolean network is to be implemented using a set of gates that have been carefully designed and characterized. This set of gates is referred to as the **gate library**. Each gate has a cost that represents its area and/or delay. Hence, when a gate is being considered as a match of a set of Boolean nodes in the network, the cost can be computed. To reduce the complexity of having to map nodes with arbitrary number of fanins into gates in the library, the Boolean network is first decomposed into basic gates like two-input NAND/NOR gates and inverters. The decomposed Boolean network is called the **subject graph**. This step is often call **technology decomposition**. Each gate in the library is also represented as a Boolean network where each node is also of the same basic gates. These graphs of gates are called the **pattern graphs**. A **cover** of the subject

graph is a collection of pattern graphs such that every node in the subject graph is contained in one or more of the pattern graphs. The cover is constrained such that each primary output is an output of a pattern graph, and each input required by a pattern graph is either a primary input or an output of another pattern graph in the cover. Finding a cover of the subject graph is often referred to as **technology mapping**.

Technology mapping problem is NP-hard, which means that heuristics are used to solve the problem. The most commonly used heuristic partitions the subject graph into trees and a dynamic programming approach is used to find the optimum mapping of the trees like in [?].

## 2.3 Circuit Delay

### 2.3.1 Delay Models

The delay of a circuit depends on the delay of both gates and interconnect in the circuit. As seen in Section 1.2.2, the delay of a circuit segment, i.e. a CMOS gate driving a piece of wire with length $l$ and another CMOS gate, can be described by Equation 1.1. For $x = 1, a(x) = 0.5$, and $b(x) = 1.0$ and the equation becomes

$$\tau = R_{tr}(C_L + C_p) + (R_{tr}c + rC_L)l + \frac{1}{2}rcl^2 \tag{2.1}$$

which is the widely used Elmore delay [Elm48]. If there is no interconnect between the driver and the receiver, the equation becomes

$$\tau_g = R_{tr}(C_L + C_p) \tag{2.2}$$

which is defined as the **gate delay** of the circuit segment. The difference between these two equations is defined as the **interconnect delay** of the circuit segment , i.e

$$\tau_w = (R_{tr}c + rC_L)l + \frac{1}{2}rcl^2 \tag{2.3}$$

The conventional logic synthesis model the delay of a gate $g$ in the library as

$$d_g(i,g) = \alpha(i) + \beta(i)\gamma \tag{2.4}$$

where $\alpha(i)$ and $\beta(i)$ are the **intrinsic delay** and **drive strength** (or **output resistance**) of the gate from pin $i$ to the output of the gate, and $\gamma$ is the load capacitances seen at the output of $g$. In this model, interconnect capacitance is lumped into $\gamma$, but interconnect resistance is neglected because it has been negligible for circuits typically synthesized. Hence, $d_g(i,g)$ includes the terms

$R_{tr}C_L, R_{tr}C_p$, and $R_{tr}cl$ of Equation 2.1. In this thesis, this equation is still used as the delay of $g$ although the term $R_{tr}cl$ is a part of the interconnect delay. The remaining terms $rlC_L$ and $\frac{1}{2}rcl^2$ are called the **interconnect resistance delay**, and denoted as $d_r$. Hence, the interconnect resistance delay at the input pin $j$ of a receiver $h$ for a signal traveling from $g$ to the receiver is

$$d_r(g,j) = rlC_L + \frac{1}{2}rcl^2 \tag{2.5}$$

These definitions of gate delay and interconnect resistance delay allow the definitions of arrival times and required times at the input pins and output of a node in the network, which are described next.

### 2.3.2 Timing Analysis

Timing-driven logic synthesis of a combinational circuit takes as input a timing specification along with a functional specification to generate a circuit satisfying both the functionality and timing. The timing specification is given as **arrival time** at each primary input and **required time** at each primary output. The arrival time of a node is defined as the earliest time the signal at the output of the node becomes stable. In other words, it is the latest time a signal can arrive at the output of the node. The arrival time of a node $g$ is denoted as $a(g)$ and computed as follows

$$a(g) = \max_{f \in FI(g)} \{a(f) + d_r(f,i) + d_g(i,g)\}$$

where $i$ is the input pin of gate $g$ driven by the fanin node $f$. The terms in this equation is illustrated in Figure 2.2. Similarly the required time of a node $f$ is denoted as $r(f)$ and computed as follows

$$r(f) = \min_{g \in FO(f)} \{r(g) - d_g(i,g) - d_r(f,i)\}$$

and the slack at node $g$ denoted as $s(g)$ is the difference between the required time and arrival time at $g$

$$s(g) = r(g) - a(g)$$

The arrival times of the nodes in the Boolean network can be computed by traversing the network in topological order, and the required times in reverse topological order.

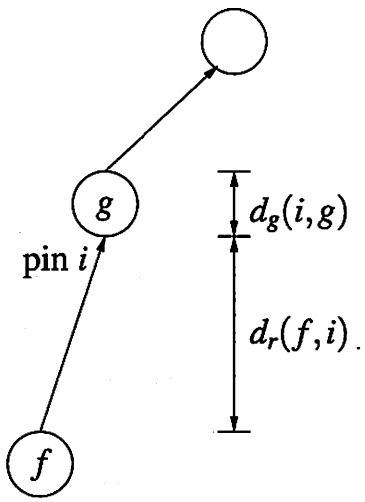



Figure 2.2: Arrival and required time.

# Chapter 3

# Wire-planning

In Section 1.2.2, we demonstrated that interconnect delay is becoming more important as we scale down the process feature size and as we increase the size of the functional block that we synthesize. In this chapter, we propose a new logic synthesis methodology to deal with the increasing contribution of the interconnect delay. Our focus is on logic synthesis. We first show that conventional logic synthesis techniques can produce circuits which have long paths even if placed optimally. Then, we characterize the conditions under which this can happen and propose logic synthesis techniques which produce circuits which are "better" for placement.

Conventional logic synthesis assumes that the delay of a circuit depends only on the delays of the gates in the circuit and mostly ignores the effect of interconnect delay[1]. However, we saw in Section 1.2.2 that interconnect delay is becoming more important not only due to technology scaling, but also due to the increase in size of blocks that we would like to synthesize. Therefore, logic synthesis needs to account for the effect of interconnect delay during optimization.

In this chapter, we adopt a diametrically opposite approach to that of conventional logic synthesis. We perform logic synthesis to optimize only for interconnect delay, ignoring the effect of gate delays. We assume that the interconnect delay from an input $i$ to an output $o$ is a linear function of the length of the path which connects $i$ to $o$. This is supported by the work in [OB98]. Our approach is based on the simple observation that if an output $o$ depends on an input $i$, then the best way to connect $i$ to $o$ is through a path which is monotonic from $i$ to $o$, that is, there are no "diversions" in the path from $i$ to $o$ (In other words, the length of the path is exactly the Manhattan distance between $i$ and $o$). We first show, by means of an example, that conventional logic synthesis

[1]The interconnect delay is typically estimated using a wire-load model that underestimates many nets, as we have demonstrated in Section 1.2.3

can produce a circuit for which it is impossible to find a placement with no diversions in the input-output paths. Therefore, no place & route tool will be able to produce a circuit which is optimal in terms of interconnect delay.

We define the notion of *illegal* nodes. Intuitively speaking, a node is illegal if it introduces a diversion in the circuit no matter where it is placed. We characterize the condition under which a node is illegal and provide a procedure to convert an arbitrary circuit into a circuit which has only legal nodes. We call such a circuit a *legal* circuit. We show that for a legal circuit, there always exists a point placement of the nodes such that every input-output path is monotonic. We also provide a set of logic synthesis transformations which are guaranteed to preserve the "legality" of a circuit.

The chapter is structured as follows. Section 3.1 discusses related previous work. In Section 3.2, we state our definitions and terminologies. In Section 3.3, we show examples of a circuit that has monotonic placement and circuits that do not have monotonic placement. In Section 3.4, the constraints placed on regions of the core area and Boolean nodes are described in detail. We also describe a logic operation that transforms an illegal Boolean network into a legal one. In Section 3.5, we present logic operations that optimizes the Boolean network while preserving its legality. In Section 3.6, we present results of this approach and discuss the advantages and disadvantages of wire-planning approach. Finally, we conclude this chapter in Section 3.7.

## 3.1 Previous Work

So far very little work has been done to model the effect of interconnect delay at the logic level. This is mainly due to the fact that at the logic level, very little information is available about the interconnect. Most of these approaches [PB91b, PB91a, VP93] use a rough companion placement to estimate the cost of various logic synthesis operations and make decisions based on this cost. In [SRRJ97a] an iterative approach to combine synthesis and placement is presented. Instead of using a companion placement to guide synthesis, they use actual placement which can be modified incrementally based on the netlist changes. In [VP95] a heuristic to minimize the layout cost is proposed which doesn't employ a companion placement solution. The method in [VP95] is based on minimizing the average fanout range and evenly distributing fanouts in the chip. It was shown that the chip delay could be reduced by this approach if all the input pins are located on one side of the chip and all the output pins on the opposite. The wire-planning approach also does not employ a companion placement. Instead, it provides a procedure to transform a Boolean network that has diversions when placed into another Boolean network that does not.

## 3.2 Preliminaries

Given a placed circuit where every cell in the circuit is treated as a point, a path, $p_{(i,o)}$, from a primary input $i$ to a primary output $o$ is a sequence of connected nodes from $i$ to $o$. The length of path $p_{(i,o)}$, $d_{(i,o)}$, is the length of all the wires along the path from $i$ to $o$. The path $p_{(i,o)}$ is called *monotonic* if its length is equal to the Manhattan distance from $i$ to $o$.

Given a circuit represented as a Boolean network, the goal of the wire-planning approach is to find a synthesized and placed circuit such that the interconnect delay of the circuit is minimized. Rather than placing the circuit, the wire-planning approach finds an optimized Boolean network, which when placed optimally, leads to a circuit with minimum interconnect delay. It is up to the placement tool to find the optimal placement for such a network. Intuitively speaking, we are trying to create a circuit for which a "good" placement exists.

We assume that the core of the circuit is represented by a rectangle $R$ with width $w_R$ and height $h_R$ and the input and output pin positions of the given circuit are known. We assume that the delay of a path is a linear function of its length. In general, the interconnect delay depends quadratically on the length of the interconnect. However, it can be made linear by buffer insertion and wire sizing, as shown in the studies in [OB98] and [CP98]. A circuit is said to be optimal in terms of interconnect delay if the length of a path from any primary input $i$ to any primary output $o$ is its Manhattan distance (monotonic), i.e.

$$d_{(i,o)} = |x_i - x_o| + |y_i - y_o|$$

In this chapter, the Boolean network being synthesized is denoted as $\mathcal{N}$.

## 3.3 Logic Synthesis and Interconnect Delay: An Example

To understand the problem better, let us first look at an example where the conventional logic synthesis which considers only gates during optimization may not be able to find a circuit with minimum interconnect delay.

Figure 3.1(a) shows a minimum literal Boolean network $\mathcal{N}_{\text{min}}$. This network has 10 literals. The primary inputs of $\mathcal{N}_{\text{min}}$ are $a, b, c, d, e$, and $f$. The primary outputs of $\mathcal{N}_{\text{min}}$ are $y_1$ and $y_2$. The given positions of all primary input and output pins, and the optimal placement of $\mathcal{N}_{\text{min}}$ is shown in Figure 3.1(b). Pins $e$ and $f$ are not shown and are assumed to be close to $y_1$. In this solution, there are two longest paths of equal length, i.e. one path from $b$ to $y_1$ and the other from

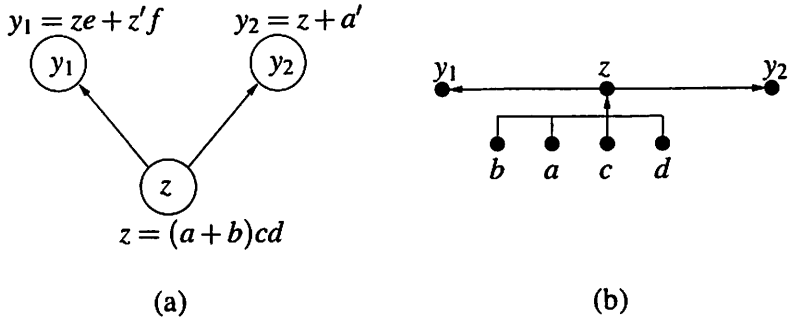



Figure 3.1: Network $\mathcal{N}_{\min}$ and its optimal placement.

$b$ to $y_2$. This circuit is not optimal because there is a better decomposition of the circuit that produces shorter longest paths. The better decomposed network $\mathcal{N}'$ has 11 literals and is shown in Figure 3.2(a). Its optimal placement is shown in Figure 3.2(b). Although network $\mathcal{N}_{\min}$ has fewer literals than $\mathcal{N}'$, it has a path from $b$ to $y_2$. Consequently, an optimal placement tool places node $z$ of $\mathcal{N}_{\min}$ in the position shown in Figure 3.1(b) in order to minimize the longest paths from $b$ to $y_1$ and $y_2$. However, as we see in Figure 3.2(a), $y_2$ is independent of $b$ and therefore, $b$ can be removed from the support of $y_2$. There are three longest paths in the optimal placement of network $\mathcal{N}'$: the path from $a$ to $y_2$, the path from $c$ to $y_1$, and the path from $d$ to $y_1$. The length of each of these longest paths is smaller than the length of the longest path in network $\mathcal{N}_{\min}$.

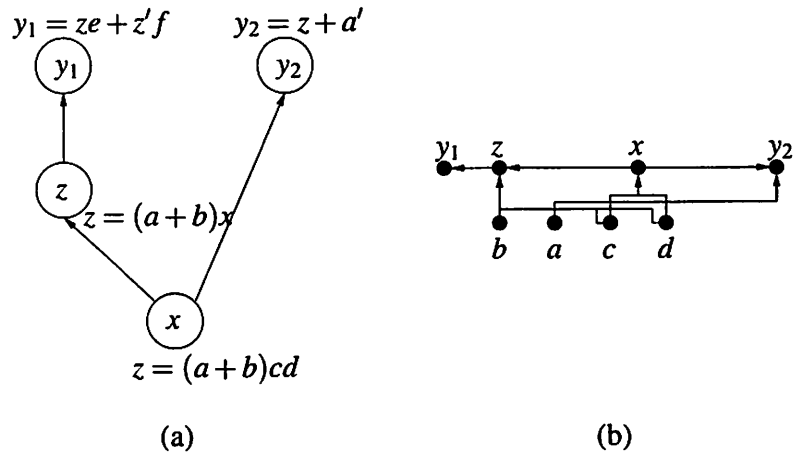



Figure 3.2: Network $\mathcal{N}'$ and its optimal placement.

Although the length of the longest paths in network $\mathcal{N}'$ are shorter than those of $\mathcal{N}_{\min}$, there is another decomposition with fewer longest paths. In $\mathcal{N}'$, the path from $c$ to $y_1$ is greater than

its Manhattan distance. The same is true for the path from $d$ to $y_2$. A better decomposed network $\mathcal{N}''$ with 11 literals is shown in Figure 3.3(a). For this network, its optimal placement is shown in Figure 3.3(b). As seen from this figure, there are only two longest paths in the optimal placement of this network: the path from $a$ to $y_2$ and the path from $d$ to $y_1$. All paths in this network are monotonic.

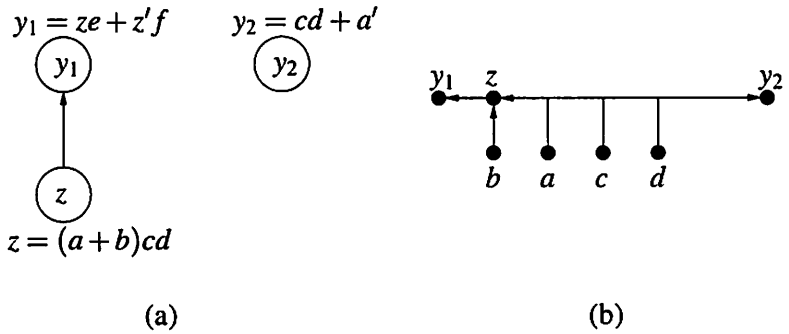



Figure 3.3: Network $\mathcal{N}''$ and its optimal placement.

From the example above, we see that sometimes the output of a logic synthesis is not "good" for placement, i.e. no matter how we place the nodes, there is at least one path which is longer than its Manhattan distance. In our approach, the aim is to guide logic synthesis such that it produces a circuit which is good for placement. It is up to the placement tool to find the optimal placement for the decomposed circuit in the placement phase.

In Section 3.4 we define what we mean by a circuit which is "good" for placement and then give a set of transformation rules which can find such a circuit. Our approach can be divided into two broad stages: constraint generation and constraint driven synthesis. In the constraint generation step, we partition the die into regions and identify the types of functions that are allowed to fill them. We define the notion of *illegal* nodes. Intuitively speaking, a node is illegal if it can not be placed somewhere on the die without causing a diversion in the circuit. We show that if a circuit consists of only legal nodes then there exists a point placement of the nodes such that every input-output path is monotonic. We call such a circuit a *legal* circuit. We characterize the condition under which a node is illegal and give a procedure to convert an arbitrary circuit into a legal circuit.

Since nodes have areas, in the constraint driven synthesis step, we synthesize the legal circuit to find another legal circuit with minimum area. We extend the algebraic transformations and don't care minimization such that they operate on legal nodes and produce legal nodes. As in the conventional logic synthesis case, we use the number of factored-form literals as our area

estimates since it has been proven to be a good indication of the size of a Boolean network.

## 3.4 Constraint Generation

Since the length of every path from a primary input to a primary output is restricted to its Manhattan distance (monotonic), there is a well defined region where a Boolean node can be placed. Let us define region formally.

**Definition 3.1** *A region* $r = \{x_l, y_t, x_r, y_b\}$, *where* $x_l \leq x_r$ *and* $y_t \leq y_b$, *is the set of all points in the rectangle bounded at opposite corners by the points* $(x_l, y_t)$ *and* $(x_r, y_b)$. *Mathematically,* $r = \{(x,y) \mid x_l \leq x \leq x_r \text{ and } y_t \leq y \leq y_b\}$.

**Definition 3.2** *Given two points* $p_1 = (x_1, y_1)$ *and* $p_2 = (x_2, y_2)$, *the region defined by* $p_1$ *and* $p_2$ *is region* $r_{(p_1,p_2)} = \{\min(x_{p_1}, x_{p_2}), \min(y_{p_1}, y_{p_2}), \max(x_{p_1}, x_{p_2}), \max(y_{p_1}, y_{p_2})\}$.

With these definitions, we analyze why node $z$ of the Boolean network $\mathcal{N}'$ in Figure 3.2 is "good" but not $x$. Node $z$ fans out to $y_1$ and its support set is $\{a,b,c,d\}$. If $z$ is placed in the region defined by $b$ and $y_1$, $r_{(b,y_1)}$, which is equal to region $r_z$ in Figure 3.4, then the path from any primary input in the support set, i.e. $a$, $b$, $c$, or $d$, to $y_1$ is monotonic. Node $x$ fans out transitively to $y1$ and $y_2$ and its support set is $\{c,d\}$. For the path from $c$ to $y_1$ to be monotonic, node $x$ needs to be placed in the region defined by $c$ and $y_1$, $r_{(c,y_1)}$, which is equal to region $r_1$ in Figure 3.5. For the path from $d$ to $y_2$ to be monotonic, node $x$ needs to be placed in the region defined by $d$ and $y_2$, $r_{(c,y_2)}$, which is equal to region $r_2$ in Figure 3.5. As shown in the figure, $x$ can not be placed in both $r_1$ and $r_2$. One of the two paths (the path from $c$ to $y_1$, and the path from $d$ to $y_2$) can not be monotonic due to $x$. Hence, $x$ is not a desirable factor.

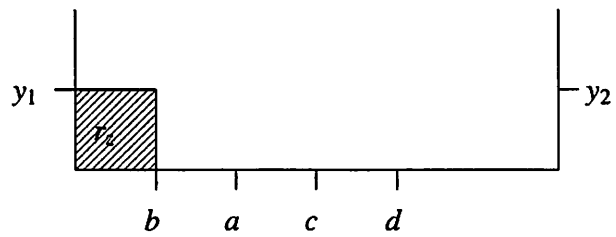



Figure 3.4: Legal region of node $z$.

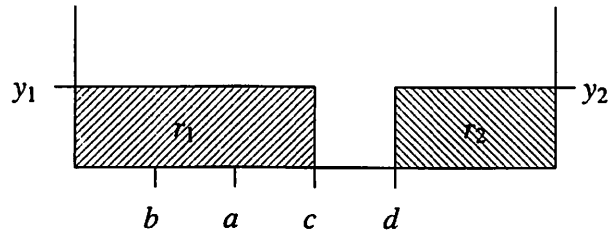



Figure 3.5: Conflicting legal region requirements for $x$.

### 3.4.1 Region Placement Constraints

The example above illustrates that if there is a path from a primary input $i$ to a primary output $o$, then for the path to be monotonic, all the logic gates along the path should be placed in the region $r_{(i,o)}$. This leads us to first partition the die into rectangles along the pin positions and label each region with functions that can be placed in it. Continuing with our example, the core area associated with $y_1, y_2, a, b, c$, and $d$ is partitioned into regions $\mathcal{R} = \{r_1, r_2, r_3, r_4, r_5\}$ as shown in Figure 3.6. Region $r_1$ is labeled with $\{a,b,c,d\}_{y_1}$. This label denotes that if the set of the primary inputs of a factor $f$ is a subset of $\{a,b,c,d\}$ and its primary output is $y_1$, then factor $f$ can be placed in region $r_1$ without violating the monotonicity of any path through it. Region $r_3$ is labeled with $\{c,d\}_{y_1}$ and $\{a,b\}_{y_2}$, This label denotes that if the set of the primary inputs of a factor $f$ is a subset of $\{c,d\}$ and its primary output is $y_1$ or the set of its primary inputs is a subset of $\{a,b\}$ and its primary output is $y_2$, then factor $f$ can be placed in region $r_3$ without violating the monotonicity of any path through it. Other regions are labeled in a similar fashion. For the Boolean network $\mathcal{N}'$, we see that node $z$ is a "good" node and can be placed in $r_1$ because its support set is $\{a,b,c,d\}$ and its primary output is $y_1$. This matches the label of $r_1$. Node $x$ is not a "good" node because there is no region whose label contains the set of its primary inputs $\{c,d\}$ and the set of its primary outputs $\{y_1, y_2\}$.

**Definition 3.3** *A placement constraint $d$ is a 2-tuple $(O^d, \sigma^d)$, where $O^d \subseteq PO(\mathcal{N})$, and $\sigma^d \subseteq PI(\mathcal{N})$. $O^d$ is called the output set and $\sigma^d$ the support set of $d$. We also write $d$ as $\{i_1, i_2, \ldots\}_{o_1,o_2,\ldots}$, where $\sigma^d = \{i_1, i_2, \ldots\}$ and $O^d = \{o_1, o_2, \ldots\}$.*

Each region is labeled with a set of placement constraints, e.g. $r_1$ is labeled with $\{a,b,c,d\}_{y_1}$ and $r_3$ is labeled with $\{c,d\}_{y_1}$ and $\{a,b\}_{y_2}$ as shown in Figure 3.6. A placement constraint on a region $r$ is called its *region placement constraint.*

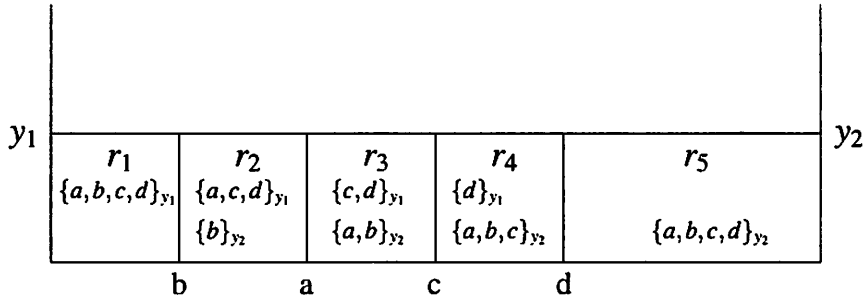



Figure 3.6: Regions and labels of regions.

Hence, each region placement constraint $d_r = (O^r, \sigma^r)$ in a region $r$ denotes that Boolean nodes that fan out only to a subset of the primary outputs in $O^r$ and have at most $\sigma^r$ in their support can be placed in $r$.

### 3.4.2 Node Placement Constraints

We see that given a region $r$, only certain types of nodes can be placed in $r$ and this is captured in its *region placement constraint*. We now define the dual for nodes. Given a node $n$, it can only be placed in certain regions. For example, node $z$ of Boolean network $\mathcal{N}'$ in Figure 3.2 can only be placed in region $r_1$ as shown in Figure 3.6. Hence, we label each node with a placement constraint and it is called its *node placement constraint*. The node placement constraint of node $n$ denotes the support of $n$ and its transitive primary outputs. For example, the node placement constraint of $z$ of Boolean network $\mathcal{N}'$ is $\{a,b,c,d\}_{y_1}$.

The node placement constraints of nodes of a Boolean network can be easily computed by traversing the Boolean network in a breadth-first manner from the primary inputs to compute the support sets and from the primary outputs to compute the output sets.

### 3.4.3 Properties of Placement Constraints on Boolean Networks

In this section, we show what "good" nodes mean and having a Boolean Network with only "good" nodes can lead to a monotonic point placement of the network.

Intuitively, a "good" node is one that can be placed in a region. We define such "good" nodes as legal. However, before we can formally define the legality of a node, we need the definition of containment of placement constraints.

**Definition 3.4** *Placement constraint* $d_a = (O^a, \sigma^a)$ *is contained in placement constraint* $d_b = (O^b, \sigma^b)$, *denoted as* $d_a \subseteq d_b$, *if* $O^a \subseteq O^b$ *and* $\sigma^a \subseteq \sigma^b$.

**Definition 3.5** *Boolean node n with node placement constraint* $d_n$ *is* **legal** *with respect to region r with region placement constraints* $\{d_{r_1}, d_{r_2}, \ldots\}$, *denoted as* $n \downarrow r$, *if there exists a j such that* $d_n \subseteq d_{r_j}$.

Definition 3.5 says that node $n$ is legal with respect to region $r$ if $n$ can be placed in $r$.

**Definition 3.6** *A Boolean node n is* **legal** *if there is a region r such that* $n \downarrow r$.

Definition 3.6 says that node $n$ is legal if there is a region $r$ where $n$ can be placed. This definition and Definition 3.5 are about the legality of a Boolean node. Now given a node, the next definition defines the region in which the node is legal.

**Definition 3.7** *The legal regions of a node n, denoted as* $R(n)$, *is the set of regions* $\mathcal{R} = \{r_1, r_2, \ldots, r_l\}$ *such that for any region* $r_j \in \mathcal{R}$, $n \downarrow r_j$ *and* $r_j \notin \mathcal{R}$, $n \not\downarrow r_j$.

For clarity purposes, we denote the legal region of a node $n$ with node placement constraint $d_n$ as $R(d_n)$. We will then assume that given a node placement constraint, the node is implicitly defined.

It can be easily seen that $R(\{i_k\}_{o_l})$ is the region defined by points $i_k$ and $o_l$, $r_{(i_k,o_l)}$. If we define $R(d_1) \cap R(d_2)$ to be the overlapping region between $R(d_1)$ and $R(d_2)$, then it is easy to see that $R(\{i_1, i_2, \ldots, i_m\}_{o_1,o_2,\ldots,o_n})$ is equal to:

$$\begin{aligned} &R(\{i_1\}_{o_1}) \cap R(\{i_2\}_{o_1}) \cap \cdots \cap R(\{i_m\}_{o_1}) \cap \\ &R(\{i_1\}_{o_2}) \cap R(\{i_2\}_{o_2}) \cap \cdots \cap R(\{i_m\}_{o_2}) \cap \\ &\qquad\qquad\qquad\qquad\qquad\qquad \cdots \cap \\ &R(\{i_1\}_{o_n}) \cap R(\{i_2\}_{o_n}) \cap \cdots \cap R(\{i_m\}_{o_n}). \end{aligned}$$

$R(\{i_1, i_2, \ldots, i_m\}_{o_1,o_2,\ldots,o_n})$ is a rectangle since it is a region defined by overlapping rectangles. This is called the *intersection rule*. For example, as shown in Figure 3.7, for node $z$ of Boolean network $\mathcal{N}'$,

$$\begin{aligned} R(z) &= R(\{a,b,c,d\}_{y_1}) \\ &= R(\{a\}_{y_1}) \cap R(\{b\}_{y_1}) \cap R(\{c\}_{y_1}) \cap R(\{d\}_{y_1}) \\ &= r_{z_1} \cap r_{z_2} \cap r_{z_3} \cap r_{z_4} \\ &= r_z \end{aligned}$$

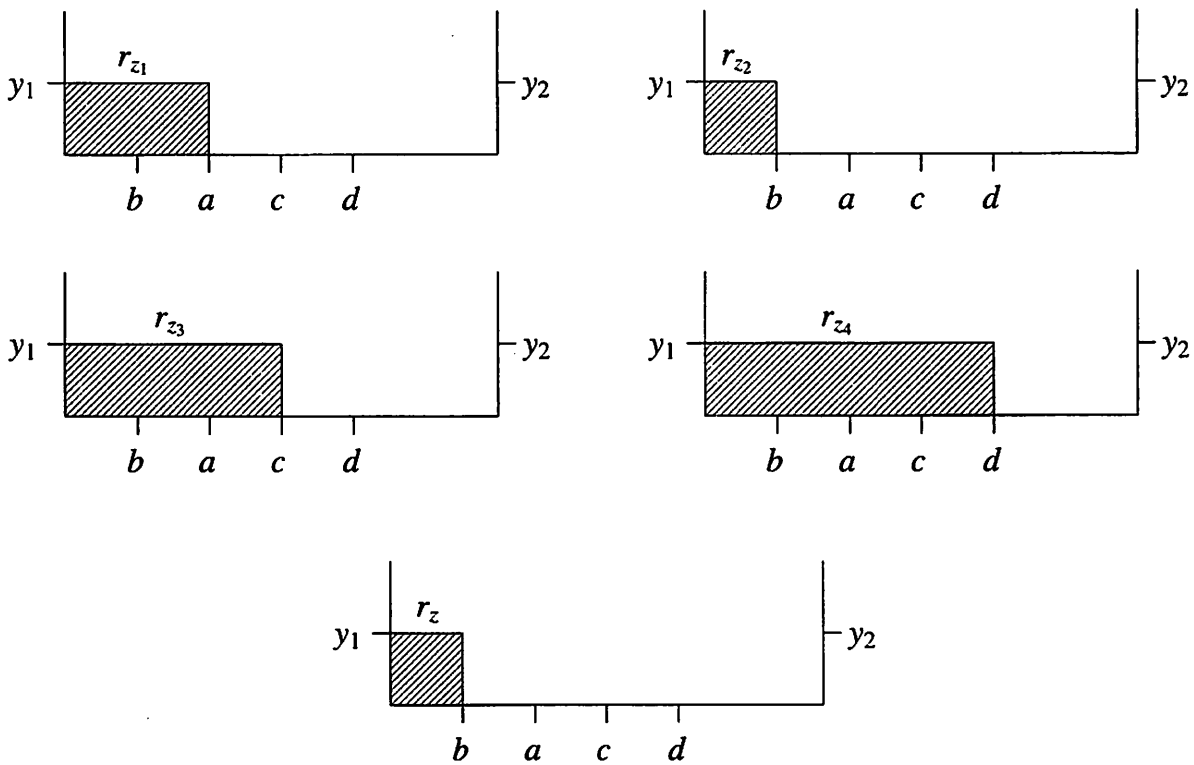



Figure 3.7: Region intersection for node $z$ of $\mathcal{N}'$.

Based on Definition 3.6, the legality of a node $n$ with node placement constraint $d_n = (O^n, \sigma^n)$ can be checked by traversing all regions and check if $n$ is legal for each region. Assuming $|PI(\mathcal{N})| > ||PO(\mathcal{N})|$, the complexity of this algorithm is $O(|PI(\mathcal{N})|^2\ |PO(\mathcal{N})|)$ because the number of regions is $O(|PI(\mathcal{N})|\ |PO(\mathcal{N})|)$ and the number of region placement constraints in a region is $O(|PI(\mathcal{N})| + |PO(\mathcal{N})|)$. A better algorithm would be to check if the legal region of $n$ is empty or not. This can be done by using the intersection rule defined above. The complexity is then $O(|O^n|\ |\sigma^n|)$ (Note that $n$ here is not an exponent, but $O^n$ and $\sigma^n$ are the output set and the support set of the placement constraint of $n$, respectively.), which is much smaller. However, there is a linear algorithm with complexity $O(|O^n| + |\sigma^n|)$ according to the next three lemmas.

Lemma 3.1 below says that nodes that transitively fan out to only one output are always legal.

**Lemma 3.1** *For a node placement constraint* $\{i_1, i_2, \ldots, i_m\}_{o_1,o_2,\ldots,o_n}$ *with* $n = 1$, $R(\{i_1, i_2, \ldots, i_m\}_{o_1,o_2,\ldots,o_n}) \neq 0$.

**Proof**: From the definition of regions in Definition 3.1 and the intersection rule, the point $(x_{o_l}, y_{o_l})$ is in $R(\{i_1, i_2, \ldots, i_m\}_{o_l})$. ∎

Lemma 3.2 below enumerates the cases when nodes that transitively fan out to two outputs are legal.

**Lemma 3.2** *For a node placement constraint* $\{i_1, i_2, \ldots, i_m\}_{o_1,o_2,\ldots,o_n}$ *with* $m \geq 2$ *and* $n = 2$, $R(\{i_1, i_2, \ldots, i_m\}_{o_1,o_2,\ldots,o_n}) \neq 0$ *iff*

1. $(\forall i \forall o\ x_i \geq x_o \wedge y_i \geq y_o) \vee (\forall i \forall o\ x_i \geq x_o \wedge y_i \leq y_o) \vee (\forall i \forall o\ x_i \leq x_o \wedge y_i \geq y_o) \vee (\forall i \forall o\ x_i \leq x_o \wedge y_i \leq y_o)$, *or*

2. $R(\{i_1, i_2, \ldots, i_m\}_{o_l})$ *is a point, i.e.* $x_{o_1} = x_{o_2} \wedge \forall i\ y_i = C$, *or* $y_{o_1} = y_{o_2} \wedge \forall i\ x_i = C$, *for some* $C \in \mathcal{N}$.

**Proof: If part**:

1. Let us assume without loss of generality that $(\forall i \forall o\ x_i \geq x_o \wedge y_i \geq y_o)$, and let $i_{\min} = (\min\{x_i\}, \min\{y_i\})$ and $o_{\max} = (\max\{x_o\}, \max\{y_o\})$, then the legal region is $r_{(i_{\min}, o_{\max})}$ and it is not empty.

2. If the legal region is a point, then it is not empty.

**Only if part:** Without loss of generality assume that the legal region is not empty and it is not a point, but $x_{i_1} < x_{o_1} < x_{i_2}$, i.e. $o_1$ is on the top side of the die, then $R(\{i_1, i_2\}_{o_1})$ is a point if both $i_1$ and $i_2$ are on the top side as well (Figure 3.8a); it is a line otherwise (Figure 3.8b). Since $R(\{i_1, i_2\}_{o_1,o_2}) = R(\{i_1, i_2\}_{o_1}) \wedge R(\{i_1, i_2\}_{o_2})$, it is not empty iff $y_{i_1} = y_{i_2}$ and $x_{o_1} = x_{o_2}$, i.e. $o_1$ and $o_2$ are at opposite side (Figure 3.8c). If we have more than two inputs, then they all have to be either on the top or the bottom side of the chip for the legal region to be non-empty and the legal region has to be a point (Figure 3.8d). Hence, it is a contradiction. ∎

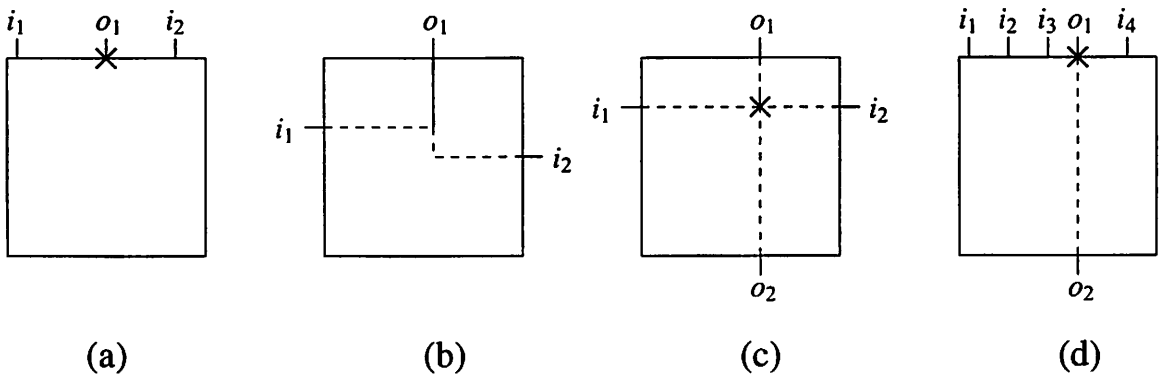



Figure 3.8: Figure for proof of Lemma 3.2.

The following lemma says if a node transitively fans out to more than two outputs, then there can only be one case where it is legal.

**Lemma 3.3** *For a node placement constraint* $\{i_1, i_2, \ldots, i_m\}_{o_1,o_2,\ldots,o_n}$ *with* $m \geq 2$, *and* $n > 2$, $R(\{i_1, i_2, \ldots, i_m\}_{o_1,o_2,\ldots,o_n}) \neq 0$ *iff* $(\forall i \forall o\ x_i \geq x_o \wedge y_i \geq y_o) \vee (\forall i \forall o\ x_i \geq x_o \wedge y_i \leq y_o) \vee (\forall i \forall o\ x_i \leq x_o \wedge y_i \geq y_o) \vee (\forall i \forall o\ x_i \leq x_o \wedge y_i \leq y_o)$.

**Proof**: The proof is similar to the proof of Lemma 3.2.

**If part**: This is the same as the first case of the if part of Lemma 3.2 proof.

**Only if part**: Without loss of generality assume that the legal region is not empty but $x_{i_1} < x_{o_1} < x_{i_2}$, i.e. $o_1$ is on the top side of the die, then $R(\{i_1, i_2\}_{o_1}$ is a point if both $i_1$ and $i_2$ are on the top side as well (Figure 3.8a); it is a line otherwise (Figure 3.8b). Since $R(\{i_1, i_2\}_{o_1,o_2}) = R(\{i_1, i_2\}_{o_1}) \wedge R(\{i_1, i_2\}_{o_2})$, it is not empty iff $y_{i_1} = y_{i_2}$ and $x_{o_1} = x_{o_2}$, i.e. $o_1$ and $o_2$ are at opposite side (Figure 3.8c). There is no way to add a third output to $\{i_1, i_2\}_{o_1,o_2}$ with a non-empty legal region. Hence, it is a contradiction. ∎

By the input-output symmetric nature of legal regions, the above three lemmas apply with the role of $m$ and $n$ interchanged.

Let the condition $(\forall i \forall o\, x_i \geq x_o \wedge y_i \geq y_o) \vee (\forall i \forall o\, x_i \geq x_o \wedge y_i \leq y_o) \vee (\forall i \forall o\, x_i \leq x_o \wedge y_i \geq y_o) \vee (\forall i \forall o\, x_i \leq x_o \wedge y_i \leq y_o)$ be called the *non-overlapping* condition. Then, with these three lemmas, the legality of a node with node placement constraint $\{i_1, i_2, \ldots, i_m\}_{o_1,o_2,\ldots,o_n}$ can be checked with the following algorithm:

1. If $n$ is 1, then the node is legal.
2. If the non-overlapping condition is true, then the node is legal. This can be checked in $O(m+n)$ by first finding the largest and smallest $x$ and $y$ coordinates of both inputs and outputs and then check for the overlapping condition using these values.
3. If the node placement constraint satisfies Condition 2 of Lemma 3.2, then it is legal.
4. If none of the above are satisfied, then the node is illegal.

It is obvious that this legality checking algorithm is $O(m+n)$. Hence, it is very efficient.

**Corollary 3.4** *There exists a corner point $p_c$ of $R(\{i_1, i_2, \ldots, i_m\}_{o_1,o_2,\ldots,o_n})$ that is closest in distance to all outputs, and a corner point $p_f$ furthest from all outputs. The point $p_c$ is called the closest point of the region and $p_f$ the furthest point.*

**Lemma 3.5** *1. If $R(\{i_1, i_2, \ldots, i_m\}_{o_1,o_2,\ldots,o_n}) \cap R(\{i_k\}_{o_1,o_2,\ldots,o_n})$, where $i_k \notin \{i_1, i_2, \ldots, i_m\}$, is not empty, then it contains the closest point of $R(\{i_1, i_2, \ldots, i_m\}_{o_1,o_2,\ldots,o_n})$.*

*2. If $R(\{i_1, i_2, \ldots, i_m\}_{o_1,o_2,\ldots,o_n}) \cap R(\{i_1, i_2, \ldots, i_m\}_{o_k}$, where $o_k \notin \{o_1, o_2, \ldots, o_n\}$, is not empty, then it contains the furthest point of $R(\{i_1, i_2, \ldots, i_m\}_{o_1,o_2,\ldots,o_n})$.*

**Proof**: Assume that $m \geq 2$ and $n > 2$. The proof is similar for other cases.

1. Assume $(\forall i \forall o\, x_i > x_o \wedge y_i > y_o)$ (the proofs of the other cases are the same), then $x_k > x_o \wedge y_k > y_o$. If $x_k$ is greater than the $x$-coordinates of any other input, then $R(\{i_1, i_2, \ldots, i_m\}_{o_1,o_2,\ldots,o_n}) \cap R(\{i_k\}_{o_1,o_2,\ldots,o_n}) = R(\{i_1, i_2, \ldots, i_m\}_{o_1,o_2,\ldots,o_n})$. If $x_k$ is less than the $x$-coordinate of all other inputs, then the vertical line going through $i_k$ partitions $R(\{i_1, i_2, \ldots, i_m\}_{o_1,o_2,\ldots,o_n})$ into two regions and $R(\{i_1, i_2, \ldots, i_m\}_{o_1,o_2,\ldots,o_n}) \cap R(\{i_k\}_{o_1,o_2,\ldots,o_n})$ is the partition that includes the closest point.

2. The proof is similar to case 1.

■

Lemma 3.5 says that:

1. Adding inputs to a node placement constraint will not change the closest point of its legal region.
2. Adding outputs to a node placement constraint will not change the farthest point of its legal region.

At this point, we have defined what legal nodes are and how to check for legality of nodes. We now put the legal context into Boolean networks and discuss the implication of legality of Boolean network on placement.

**Definition 3.8** *A Boolean network is* **legal** *is every node in the network is* **legal**.

There is a nice property of a legal Boolean network as described by the following theorem.

**Theorem 3.6** *Given a legal boolean network, there exists a monotonic point placement for the network.*

**Proof**:

This is an induction proof. We traverse the Boolean network in a reverse topological order, i.e. a node is visited only after all its fanouts have been visited.

The base case is where we have all primary outputs. Let $o$ be an arbitrary primary output, then place $o$ at its pin location. For $o$, its pin location is its closest point. The induction hypothesis is that fanouts of a node $n$ are placed at their closest points and still maintaining monotonicity, i.e. the distances from their closest points to their primary outputs are their Manhattan distances. we show that $n$ can also be placed at its closest point while still maintaining monotonicity.

Let $n_f$ be an arbitrary fanout of $n$. Let $c'$ be the node placement constraint of $n_f$ with all fanins except $n$ removed. Also let the node placement constraints of $n$ and $n_f$ be $c$ and $c_f$. Then $c_f$ is derived from $c'$ by adding the primary inputs of fanins of $n_f$ other than $n$ and $c$ is derived from $c'$ by adding the primary outputs of fanouts of $n$ other than $n_f$. We know that $R(c') \neq 0$ because $c' \subseteq c$ and $R(c) \neq 0$ by the assumption that $n$ is legal. By applying Lemma 3.5 for each primary input added to $c'$ to form $c_f$, $R(c_f)$ includes the closest point of $R(c')$. Since $R(c) \subseteq R(c')$, the distance from the closest point of $R(c)$ to a primary output $o$ is the same as the sum of the distance from the closest

point of $R(c)$ to the closest point of $R(c_f)$ and the distance from the closest point of $R(c_f)$ and $o$. Hence, the monotonic property is maintained and $n$ can be placed at the closest point of $R(c)$.

■

Theorem 3.6 reduces our problem of finding a monotonic point placement of a circuit into the problem of finding a legal Boolean network. The logic synthesis transformations we use to convert an illegal Boolean network into a legal one is called *make-legal*, and it is explained below.

### 3.4.4 Make-Legal

The *make-legal* operation takes a Boolean network as its input and produces a legal Boolean network. In the effort of producing a legal Boolean network, it attempts to minimize the number of new Boolean nodes created.

The following lemma and corollary guarantee that a Boolean network can always be made legal.

**Lemma 3.7** *If $n$ is a fanin of $n_f$, and $n$ is illegal but $n_f$ is legal, then collapsing $n$ into $n_f$ will not make $n_f$ illegal.*

**Proof**: Collapsing $n$ to $n_f$ does not change the support of $n_f$, nor does it add any primary output to the transitive fanout of $n_f$. Therefore, the node placement constraint of $n_f$ does not change and hence $n_f$ stays legal. ■

By the proof of Theorem 3.6, we know that every primary output is legal. Then it is easy to see the following corollary.

**Corollary 3.8** *An illegal Boolean network can always be made legal by collapsing all nodes into the primary output nodes.*

Beside collapsing, node duplication can also legalize a node.

**Lemma 3.9** *Let nodes $n_f$ and $n_g$ be fanouts of $n$, and $n$ is illegal but both $n_f$ and $n_g$ are legal. If $n$ is duplicated into $n_1$ and $n_2$, such that $n_1$ is a fanin of $n_f$ but not $n_g$ and $n_2$ is a fanin of $n_g$ but not $n_f$, then both $n_1$ and $n_2$ are legal.*

**Proof**: The support of $n$ is a subset of both the supports of $n_f$ and $n_g$, but the output set of the node placement constraint of $n$ is a superset of the node placement constraints of both $n_f$ and $n_g$. By

duplicating $n$ into $n_1$ and $n_2$ such that $n_1$ is a fanin of $n_f$ and $n_2$ is a fanin of $n_g$, node placement constraint of $n_1$ is contained in that of $n_f$ and thus $n_1$ is legal. Similarly for $n_2$. ■

Make-legal traverses the Boolean network in a reverse topological order, i.e. a node is visited after all its fanouts have been visited. During the traversal, if it sees an illegal node, it collapses the node into its fanouts until the node becomes legal. Hence, there is a frontier moving from each primary output to primary inputs in its support where every node is legal on the side of the frontier toward the primary output. If the sum-of-product expression of the fanout, as a result of collapsing a node into one of its fanouts, exceeds $t$ literals, the node is replicated for each fanout until it becomes legal. The intuition behind this parameter is that large nodes tend to have more common sub-functions with other nodes and thus allow for sharing. However, the parameter should not be too large since it can result in explosion in memory usage.

As shown above, legality of a node can be checked efficiently, that is, it is linear in the size of the node placement constraint. Hence, the *make-legal* operation is efficient.

## 3.5 Constraint-Driven Synthesis

The constraint generation step takes a possibly illegal Boolean network and makes it legal. Theorem 3.6 guarantees that there exists a point placement for this network. However, by definition of the point placement of a circuit, nodes are assumed to be a point; hence, they have no area. In reality, nodes have area and the length of a longest path depends strongly on the size of a Boolean network. The constraint-driven synthesis step is responsible for minimizing the area of an already legal Boolean network while preserving its legality. As mentioned in Section 3.3, we use the number of literals of a Boolean network as a measure of the area of the circuit represented by the Boolean network. So this step is to optimize the network such that we get a minimum literal legal Boolean network.

We leverage the well developed algebraic transformations in the conventional logic synthesis by extending them to deal with and produce legal Boolean nodes. Each of these operations is explained below.

### 3.5.1 Fast Extract

The *fast extract* algorithm is explained in [VR90]. Given a Boolean network, the fast extract algorithm traverses the network and extract divisors while minimizing the literal count of

the network. The algorithm iterates until there is no improvement in literal count. In each iteration, the divisor that reduces the literal count the most is chosen. The network is then re-expressed using the new divisor. The divisors considered during each iteration are two-cube divisors and two-literal cubes. A two-cube divisor is a divisor that consists of only 2 cubes and is minimum with respect to single cube containment. A two-literal cube is a cube that consists of only 2 literals.

When dealing with legal Boolean network, this algorithm may result in illegal divisors. For example, assume that node $n$ with fanins $a$ and $b$ is the best divisor found and it divides nodes $x$, $y$, and $z$ as shown in Figure 3.9. Then the output set of the node placement constraint of $n$ is the union of the output sets of the node placement constraints of $x$, $y$, and $z$. From Section 3.4, we know that the legal region of $n$ may be empty and $n$ may therefore be illegal.

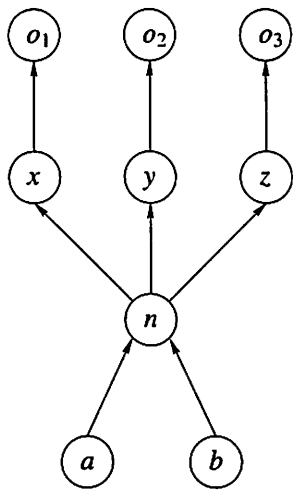



Figure 3.9: Fast extract example.

However, it may be the case that $n$ remains legal if it only divides $x$ and $y$, $x$ and $z$, or $y$ and $z$. For example, let $a$ and $b$ be primary input nodes and let the output set of $x$ be a singleton $o_1$, the output set of $y$ be a singleton $o_2$, and the output set of $z$ be a singleton $o_3$. Furthermore, let the positions of these primary inputs and outputs be as shown in Figure 3.10(a). It is clear that $n$ is not legal if it divides $x, y$, and $z$. However, if we remove $x$ from its fanout, then the legal region is shown as region $r_n$ in Figure 3.10(b).

As seen from the example in Figure 3.10, a divisor can have multiple values (which is the number of literals that can be reduced if the divisor is extracted), each associated with a subset of fanouts it can be extracted from. If node $n$ divides a set of nodes $N$, then complexity of finding

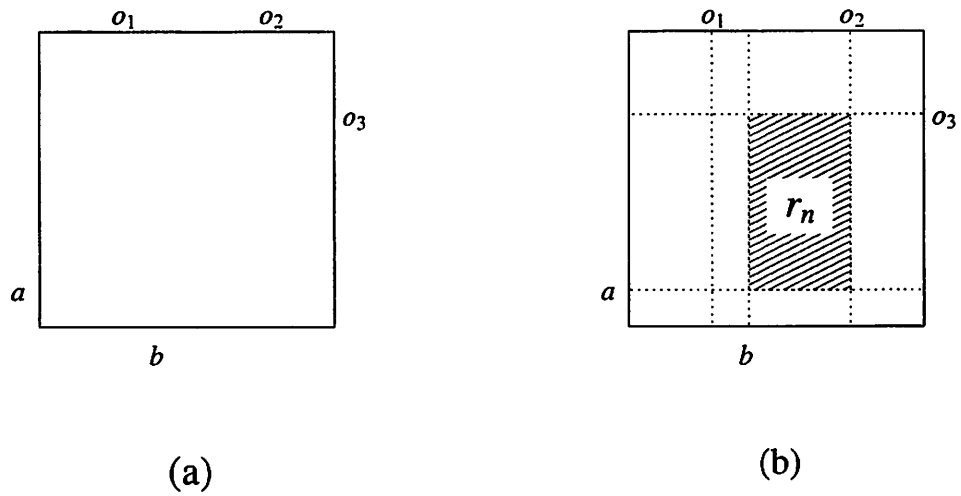



Figure 3.10: (a) Pin positions of fast extract example. (b) Legal region of node $n$.

a subset $N_l$ of $N$ which preserves the legality of $n$ and has the largest reduction in the number of literals is exponential in the size of $N$. Hence, a heuristic is used to select an optimal subset. First the nodes in $N$ are ordered in decreasing sizes of the legal regions to form a list $N_{sorted}$. Then $N_{sorted}$ is linearly traversed. Each node is added to the subset $N_l$ if the legality of $n$ is preserved. Node $n$ is used as a divisor if it reduces the number of literals in the network.

### 3.5.2 Re-substitution

In the conventional logic synthesis, a node $n$ is re-substituted into another node $x$ if $n$ divides $x$. The value of the re-substitution is the number of literals reduced by the re-substitution. The algorithm traverses the network in many iterations. In each iteration, the re-substitution with the largest value is chosen. The iteration stops when no reduction in literal count can be achieved.

If both nodes $n$ and $x$ are legal and $n$ is re-substituted into node $x$, the legality of both $n$ and $x$ may be affected. The following observation states when $n$ and $x$ can become illegal.

**Observation 3.1** *If n divides x and both n and x are legal before re-substitution, then after re-substitution*

1. *x can become illegal if its support is not the superset of that of n.*

2. *n can become illegal if its output set is not the superset of that of x.*

In other words, node $x$ can become illegal if we add a new primary input into its transitive fanin. Node $n$ can become illegal if we add a new primary output into its transitive fanout. Because algebraic operations do not add new primary inputs to any node, $x$ will remain legal if we only do algebraic re-substitution. Node $n$ can become illegal if a new primary output is added to $n$ as a result of the re-substitution.

### 3.5.3 Full_Simpfily

There are two types of *don't cares*, i.e. the observability don't cares (ODCs) and the satisfiability don't cares (SDCs). Computing the exact ODCs of a node is computationally expensive. In practice, a subset of the ODCs called the compatible ODCs (CODCs) are computed. These CODCs are expressed in terms of the primary inputs. Then together with the external don't cares (XDCs) of the primary outputs, a don't care set in terms of the immediate fanins is computed using an image computation. In computing the SDCs, a support filter is used. A node is included in the SDCs if its support set intersects the support set of the node being considered. Employing SDCs in the minimization procedure can result in boolean re-substitutions. The support filter procedure can also be used in the image computation of the CODCs and XDCs. Once the SDCs are computed and the XDCs and CODCs are expressed in terms of immediate fanins, a two-level minimization algorithm is invoked to find an optimized expression. This is simply a brief description of the *full_simplify*. For a more detail explanation, we refer the readers to [Sav92].

**Lemma 3.10** *Throughout full_simplify computation, the only steps that can introduce illegality into the network are the image computation and the SDC computation.*

**Proof**: Let node $n$ be the node we are computing don't cares for. Legality of the Boolean network can only change if an edge is added to the network. During the whole full_simplify process, only the fanin edges of $n$ can be added. Edges of fanins of other nodes can not change. Adding a fanin edge to $n$ means that a re-substitution happens and Observation 3.1 applies. Potential new fanin edges of $n$ are added only during the image computation and SDC computation through the support filter, which basically says that a node $x$ is a potential divisor of $n$ if the support of $x$ intersect the support of $n$. ■

We therefore constrain this operation by allowing a node $x$ to be in the support filter when computing full_simplify for node $n$ if the inclusion of node $x$ preserves the legality of the network according to Observation 3.1.

### 3.5.4 Synthesis Flow

With all the above basic operations, a synthesis flow is then a script similar to the rugged script, *script.rugged*, in SIS. An empirical study needs to be conducted to derive an optimal script.

## 3.6 Experimental Results

To see the effect of the proposed approach, we have implemented the basic operations described in Section 3.5. An optimization script has been created and we call it *script.wire*, which consists of:

```
make_legal
eliminate 5
sweep; eliminate -1
simplify -m nocomp
eliminate -1
sweep; eliminate 5
simplify -m nocomp
resub -a
fx
resub -a; sweep
eliminate -1; sweep
full_simplify -m nocomp
```

Our experiment uses SIS and Ritual version 3.4, a timing-driven standard cell placer [SCK92]. The input blif file and a randomly generated pad assignment file is read into SIS. The *script.wire* optimization script is run in SIS to generate an optimized logic netlist. The optimized netlist is mapped to the standard cell technology library *stdcell2_2.genlib* of SIS. The mapped netlist is then placed by Ritual with a fixed pad assignment. We measure the length of the longest path and the delay of the Ritual output. The distance of two cells is measured as the Manhattan distance from the center of both cells. The length of a path is the sum of all distances between consecutive cells along the path.

Table 3.1 shows the results for four circuits. The circuit *bbaraComb* is obtained from the sequential circuit *bbara* by removing all latches and treating the outputs of the latches as primary inputs and the inputs to the latches as primary outputs of the network. Column 2, 3, and 4 show the number of literals in factored forms of the scripts script.rugged, script.delay, and script.wire respectively. Columns 5, 6, and 7 list the length of the longest path for each script. The experiments

were run on a DEC AlphaServer 8400 with 2GB of memory. The runtime is for the technology independent step.

As shown in this table, although the number of literals in *script.wire* approach is more than that of *script.rugged*; the length of its longest path is the same for *rd53* and better in other circuits. The longest paths are much shorter than *script.delay* results.

Table 3.2 shows the run time in seconds. As seen from this table, the runtime is comparable. This is expected since the legality checking is linear in the size of the node placement constraints and hence its runtime is a minor part of the total runtime.

Table 3.3 shows the delay computed by Ritual for the four circuits. Columns 2, 3, and 4 show the wire delay for each script The total delay is listed in columns 5, 6, and 7. Except for the total delay of *z4ml* running *script.delay*, the total delay of all circuits is the best using *script.wire*.

Table 3.1: Path length comparison of *script.rugged*, *script.delay*, and *script.wire*.

| | Number of Literals | | | Length of Longest Path | | |
|---|---|---|---|---|---|---|
| Name | sc.rugged | sc.delay | sc.wire | sc.rugged | sc.delay | sc.wire |
| z4ml | 41 | 84 | 49 | 1324 | 1342 | 1025 |
| rd53 | 42 | 62 | 50 | 1122 | 1624 | 1122 |
| rd73 | 74 | 178 | 87 | 1689 | 2457 | 1680 |
| bbaraComb | 69 | 79 | 109 | 2021 | 1573 | 1464 |

Table 3.2: CPU time comparison of *script.rugged*, *script.delay*, and *script.wire*.

| | CPU time (secs) | | |
|---|---|---|---|
| Name | sc.rugged | sc.delay | sc.wire |
| z4ml | 0.2 | 0.3 | 0.3 |
| rd53 | 0.1 | 0.3 | 0.2 |
| rd73 | 0.8 | 1.8 | 1.2 |
| bbaraComb | 0.5 | 0.5 | 0.3 |

Table 3.3: Delay comparison of *script.rugged*, *script.delay*, and *script.wire*.

| | Wire Delay | | | Total Delay | | |
|---|---|---|---|---|---|---|
| Name | sc.rugged | sc.delay | sc.wire | sc.rugged | sc.delay | sc.wire |
| z4ml | 0.93 | 1.03 | 0.97 | 6.59 | 6.31 | 5.75 |
| rd53 | 1.67 | 2.13 | 1.42 | 11.40 | 9.50 | 7.36 |
| rd73 | 1.37 | 0.88 | 0.86 | 8.38 | 5.97 | 6.45 |
| bbaraComb | 2.19 | 1.72 | 1.08 | 10.37 | 7.94 | 5.98 |

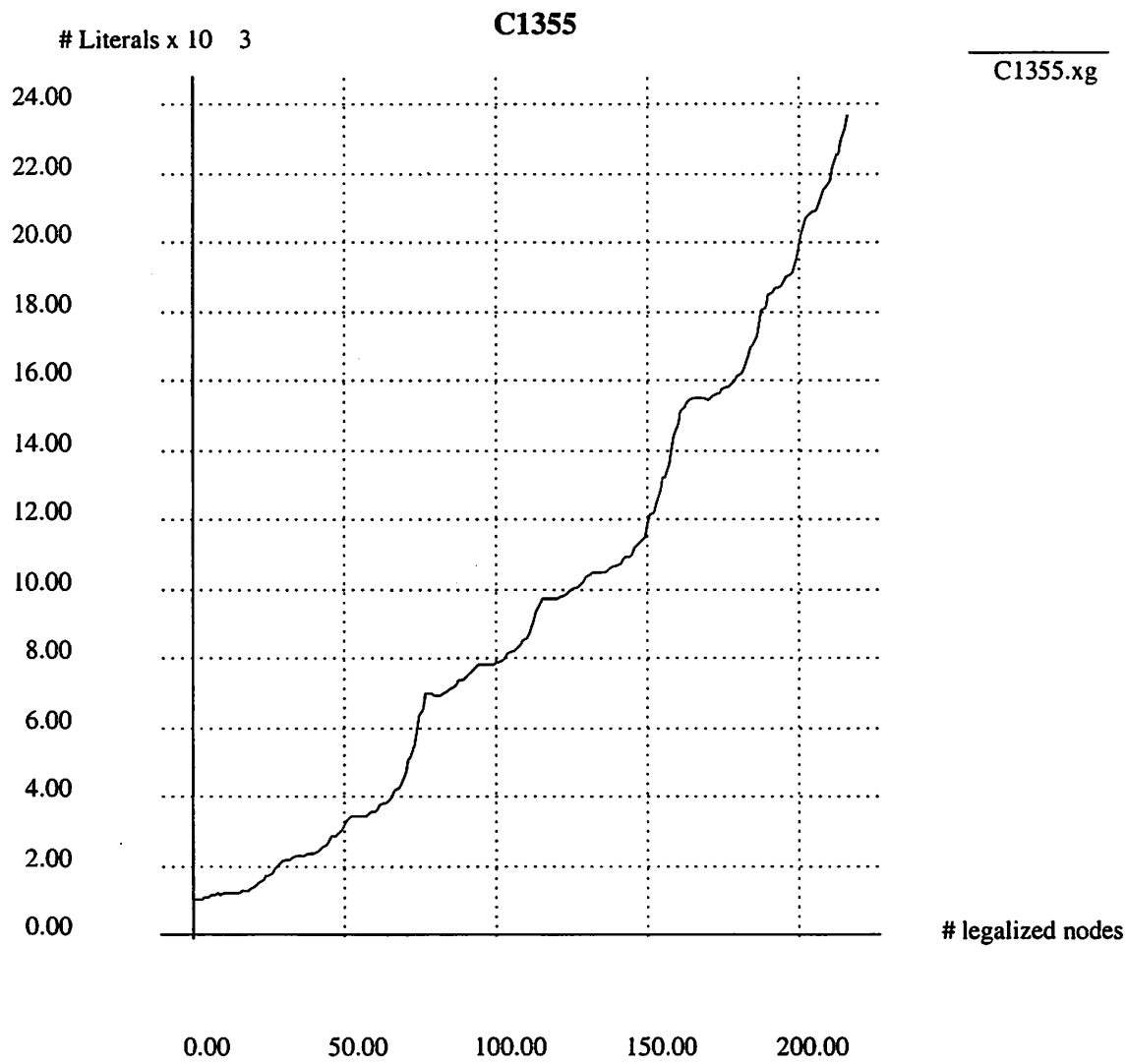



Figure 3.11: Number of literals vs number of nodes legalized for C1355.

### 3.6.1 Discussion of Results

Though the results in the previous section shows that the approach performs satisfactorily, these circuits are fairly small. For bigger circuits, the number of nodes in a legal network can be large and optimizing such large networks using operations like *fast_extract* and *full_simplify* can be very expensive. To illustrate this, we plot the number of literals versus the number of nodes in the constraint generation step for C1355 as shown in Figure 3.11. On the $x$-axis is the number of illegal nodes that are legalized. On the $y$-axis is the number of literals in the Boolean network. The network increases from 1032 literals to 23709 literals after 216 nodes have been legalized out of a total of 514 nodes in the network.

Although the wire-planning approach only works for small circuits, the theory of when a circuit will generate long wires has led us to devise heuristics in our practical approach, which is integrating logic synthesis and placement. The integrated approach will be described in the next chapter.

## 3.7 Conclusions

We have proposed a novel approach to deal with the increasingly importance of wire delays in deep sub-micron technologies. It is based on the fact that the shortest path between any two points in a circuit is the Manhattan distance between them. We showed an example of why conventional logic synthesis may produce circuits where the minimum distance can not be achieved.

The proposed approach decouples logic synthesis phase and place & route phase. It consists of a constraint generation step which produces a legal Boolean network, which can be placed such that every path is monotonic, and a constraint-driven synthesis step which minimizes the legal Boolean network while preserving legality.

The wire-planning approach is theoretical in nature and computational expensive for large circuits. However, it is the first approach that characterizes circuits with long wires without performing layout. We have implemented heuristics based on the wire-planning approach in our integrated logic synthesis and placement scheme.

# Chapter 4

# Integrating Logic Synthesis and Placement

In Chapter 3, we introduced a logic synthesis flow which guarantees a monotonically place-able layout. Logic synthesis and layout synthesis were decoupled in this approach. In this chapter, we integrate logic synthesis and placement to specifically address the *timing closure* problem. As we discussed in Section 1.2.3, timing closure problem is caused by the non-convergence of the timing estimates obtained during logic synthesis and after physical synthesis. This potentially results in a large number of iterations of logic synthesis and physical synthesis.

The timing closure problem occurs mainly due to the difficulty of predicting wire lengths in logic synthesis. Our solution integrates logic synthesis and placement in a tightly coupled fashion.

## 4.1 Design Flow

A typical application specific integrated circuit (ASIC) design flow is shown in Figure 4.1. A design described in a high level language, is passed to the logic synthesis tools. Within the logic synthesis tools, the design is synthesized into a gate level description while minimizing area and satisfying the design constraints. Although the constraints can be in terms of delay, power consumption, etc, we only focus on delay in this thesis. The optimized logic circuit is then placed by a global placement tool, again minimizing area and satisfying delay constraints. After this step, the placement is usually not *legal*[1], meaning that cells can overlap (Although the amount of overlap is

[1] Note that this notion of legality is not the same as that of Chapter 3

usually minimum). The illegal placement is then detailed placed. The detailed placement tool legalizes the placement by perturbing the location of cells minimally so that the optimization performed in the global placement step is not nullified. The optimization done during detailed placement usually involves swapping neighboring cells while minimizing area or delay. The wiring between cells is then performed during routing, which consists of global routing and detailed routing steps. In global routing, the circuit area is typically partitioned into grids and the routes of nets are planned with respect to these grids so as to minimize wire length and delay. In detailed routing, all nets are routed according to the routing plan obtained from the global routing step. It is often the case that not all nets can be routed in one iteration of global and detailed routing. In such a case, multiple iterations are needed, which usually involve ripping up and rerouting non-critical nets. After detailed routing, the design is complete. The parasitics of the circuit are extracted and a timing analysis tool is used to check if the design satisfies the delay constraints. If so, the design is finished. Otherwise, the design steps (logic synthesis, placement, and routing) need to be iterated.

As seen from Figure 4.1, there are many choices of iteration points. If there is a need to iterate, the choice of which preceding step to jump to is determined by how much the design needs to be improved in order to satisfy the delay requirement. An iteration that jumps to the logic synthesis step is usually called a *big loop*; while others are called *small loops*.

Analyzing the flow, we notice that from the global placement step onwards, an estimate of a net's wire length exists because the positions of all cells are known. This is not true at the logic synthesis step and therefore the *wire-load* models are used to estimate net lengths in a conventional flow. As we discussed in Section 1.2.3, wire-load models are inaccurate especially for larger design. The use of wire-load models is the main cause of the timing closure problem, which limits the size of a circuit that can be practically synthesized.

In an attempt to introduce more accurate wire length estimates during logic synthesis, we propose a different flow. In our flow, the logic synthesis step and the global placement step are integrated, as seen in Figure 4.2. In other words, while performing logic optimization, we will be *placing* Boolean nodes as well. Then unlike conventional logic synthesis, each Boolean node will have a position, or an $(x, y)$ coordinate. With the integrated flow, we eliminate the big loop and show results to support this.

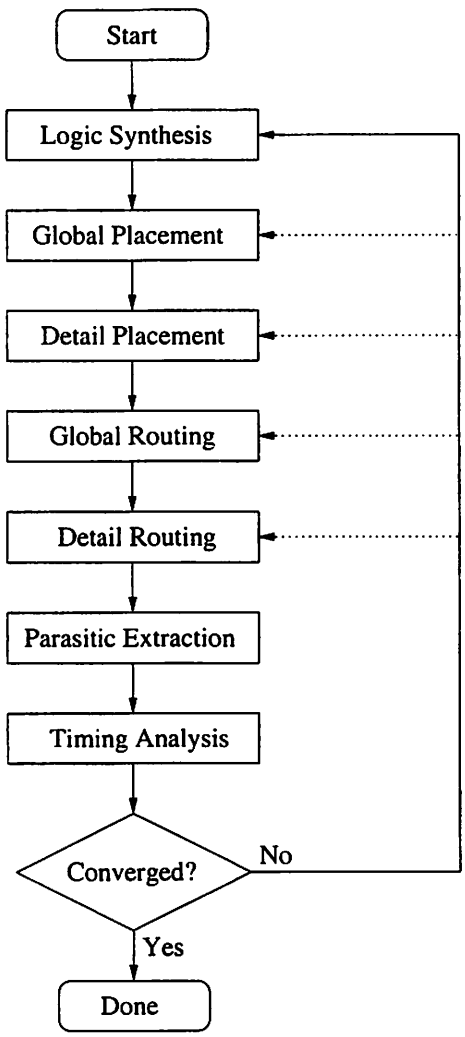



Figure 4.1: Conventional design flow.

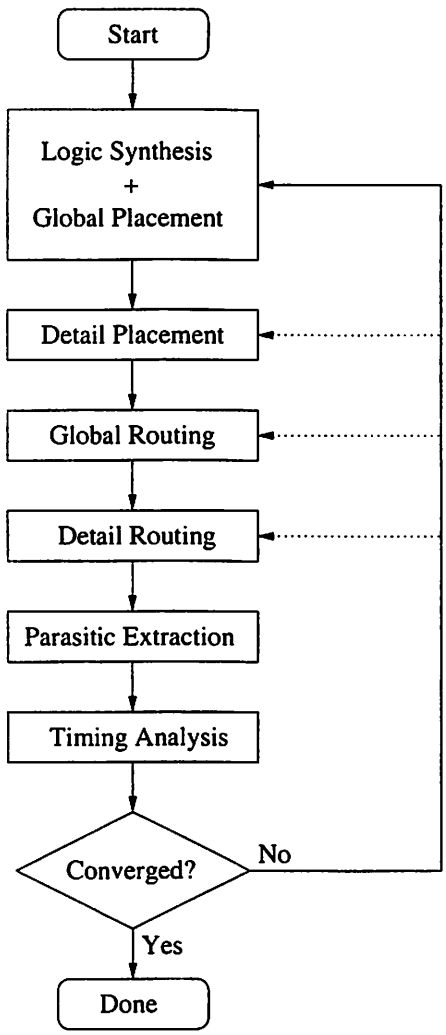



Figure 4.2: Integrated logic synthesis and placement design flow.

## 4.2 Previous Work

There have been many related efforts addressing interconnect length and interconnect delay. These efforts can be broadly divided into three categories:

- pre-layout interconnect estimation
- post-layout optimization
- integrated logic synthesis and layout

In pre-layout interconnect estimation, logic optimizations are guided by heuristics that measure the interconnect length. In post-layout optimization, logic operations are performed on a placed or placed and routed circuit to minimize the interconnect delay. The layout is minimally perturbed and updated incrementally. In integrated logic synthesis and layout approaches, placement is performed along with logic optimizations, and the positions of Boolean nodes are used to compute the cost function for logic optimizations. The majority of the previous efforts fall into the post-layout optimization category.

### 4.2.1 Pre-Layout Interconnect Estimation

Examples of interconnect estimation include the work in [KP89]. In this work, the authors propose a probabilistic model for area estimation of VLSI layouts. Based on Rent's rule, a geometric distribution for the wire-lengths is assumed. A model is constructed for the standard cell design style and analytical expressions are derived to estimate the layout area. In [VP93] a fanout optimization algorithm is proposed which maintains the order of the fanouts to simplify routing. A similar idea is used in [ASSP90] to minimize routing factor during logic synthesis. The approach is based on lexicographical expression of Boolean function controlling input dependency. In a lexicographic expression, for a given sum of products form, all the literals respect the same order in each product term. Maintaining this order in all expressions results in a simpler layout which takes less area. In [VP95] a heuristic to minimize the layout cost is proposed which doesn't employ a companion placement solution. Although impressive gains are reported for some cases, the experiments designed to obtain these are not very realistic. All the input pins are located on one side of the chip and all the output pins are located on the other side of the chip. Their method is based on minimizing the average fanout range and evenly distributing the fanout range through the chip.

### 4.2.2 Post-Layout Optimization

In the category of post-layout optimization approaches, re-wiring idea is used in [SRRJ97b] to restructure logic after detailed placement followed by an incremental detail placement. The re-wiring and incremental detailed placement optimizations are iterated until no improvement in delay is achieved. A gain of about 13% is reported. The work in [LCP99] collapses a group of cells along critical paths and re-maps them to satisfy delay requirement. Also in this category are the buffer insertion algorithm presented in [vG90], and the simultaneous driver and wire sizing algorithm found in [CK94].

### 4.2.3 Integrated Logic Synthesis and Layout

In the category of integrated approach, there have been attempts to address the interconnect delay problem. However, except for the work in [SID+99], the timing closure problem has not been addressed. A step between logic synthesis and global placement is introduced in [SID+99]. In this step, the design is iteratively improved by eliminating the maximum capacitance violations. By eliminating these violations, the authors claim to minimize the timing closure problem. The work in this thesis is orthogonal to their work. In [HV97] a re-synthesis algorithm is presented which is invoked after each minimum cut placement iteration. The re-synthesis algorithm is restricted to gate and fanout optimization. The number of electrical violations are reduced and slacks of the critical paths are increased and timing convergence is improved. A library-less technology mapping algorithm integrated with placement is presented in [JS99]. The Boolean network is first decomposed into 2-input NAND gates and inverters, and placed. Gates that are close together are then collapsed and the cell is generated on the fly using combined pass transistor logic and CMOS. An algorithm integrating technology mapping and linear placement is shown in [LSP97]. In this work, an optimum simultaneous technology mapping and linear placement is used to approximate the two-dimensional technology mapping and placement. A layout driven technology independent optimization procedure was introduced in [PB91a]. A companion placement is maintained and the positions of Boolean nodes are used to drive kernel extraction and elimination algorithms. A layout driven technology mapping approach is described in [PB91b]. While performing technology mapping, a companion placement is maintained and the positions of nodes are used to estimate the wire-lengths of nets. The estimated length is then used as part of the cost function in the technology mapping step.

The work described in this thesis is similar to the work in [PB91a] and [PB91b] as far

as integrating logic synthesis and quadratic placement. However it is very different in terms of goals and approaches. The work of [PB91a] and [PB91b] is geared towards minimizing circuit delay, whereas the goal of our work is to solve the Timing Closure problem. We believe the key to successfully integrate logic synthesis and placement lies in the ability to incrementally perform logic operations and placement in an integrated manner. One significant difference between our work and [PB91a] [PB91b] is that our placement tool is incremental. It is not clear how the placement tool in their work, GORDIAN [KSJA91], behaves from iteration to iteration because GORDIAN is a mixed quadratic and min-cut placement tool. While our placement algorithm is *integrated* with our logic synthesis tool, GORDIAN is externally invoked in their work. While our placement algorithm is run incrementally given an initial placement, GORDIAN is invoked from the beginning repeatedly in their work. Another significant difference is that in [PB91a] and [PB91b] different net models are used for the cost computation algorithms and GORDIAN while we use the same net models in order to minimize the perturbation of the existing placement.

## 4.3 Net Topology and Interconnect Delay Model

In our integrated logic synthesis and placement flow, a Boolean node has an $(x,y)$ coordinate. If the node is mapped to a library cell, then it also has a width and a height and the $(x,y)$ coordinate is the coordinate of the center of the node. If the node is not mapped, then it is treated as a point. Since we do not perform any routing in our algorithms, the topologies of nets are not known and need to be estimated.

### 4.3.1 Semi-Perimeter Estimate

In our algorithms, we need to estimate the lengths of nets and to compute the delay of the circuit. In our area and wire-length minimization algorithms, we use the semi-perimeter of the bounding box of a net to estimate the length of a net. This is a very simple estimate and can be efficiently computed. For example, a net with driver $d$ driving receivers $r_1$, $r_2$, and $r_3$ as in Figure 4.3. The semi-perimeter estimate is simply $w+h$.

### 4.3.2 Steiner Tree Estimate

For delay minimization and delay computation, the semi-perimeter estimate is not sufficiently accurate because all receivers connected to the driver of a net have the same delay under this

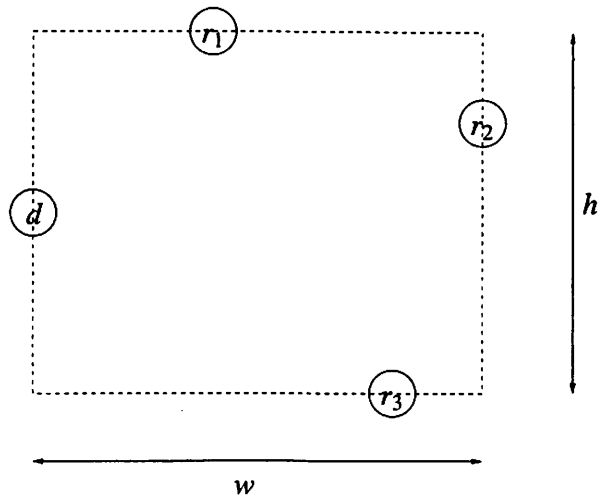



Figure 4.3: Semi-perimeter estimate of a net.

estimate. This is inaccurate since the delay of a receiver close to the driver is over-estimated and that of another receiver far from the driver is under-estimated. In order to avoid such inaccuracies, we use a Steiner tree topology. For efficiency reasons, the center of gravity of all the cells connected to the net is estimated to be the Steiner point. An example is shown in Figure 4.4. In this figure, $p$ is the Steiner point. The length of the net is estimated to be the sum of all the net segments in Figure 4.4. Each segment is modeled as a $\pi$-circuit, as shown in Figure 4.5. With this topology, the Elmore delay [Elm48] is used to estimate the delay of the net. For example, the delay from the input of $d$ to the input of $r_1$ is

$$\begin{aligned} t_{r_1} &= R_d\left(C_0+C_1+C_2+C_3+C_{L_1}+C_{L_2}+C_{L_3}\right) \\ &+ R_0\left(\tfrac{C_0}{2}+C_1+C_2+C_3+C_{L_1}+CL_2+C_{L_3}\right) \\ &+ R_1\left(\tfrac{C_1}{2}+C_{L_1}\right) \end{aligned}$$

where $R_d$ is the output resistance of $d$ and $C_{L_1}$, $C_{L_2}$, and $C_{L_3}$ are the gate capacitances of $r_1$, $r_2$, and $r_3$ respectively. In this example, we have ignored the diffusion capacitance of $d$.

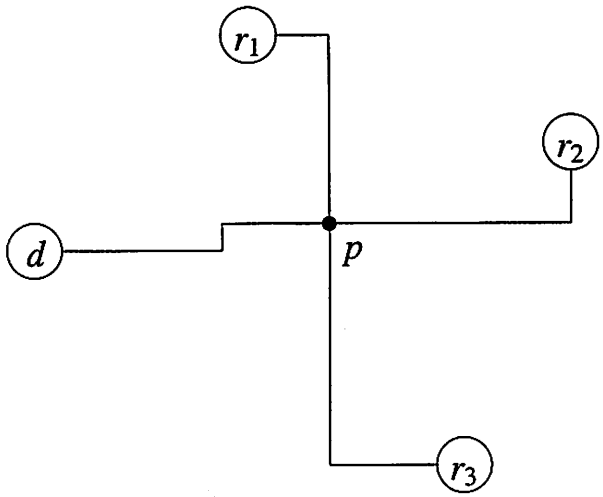



Figure 4.4: Topology model of a net.

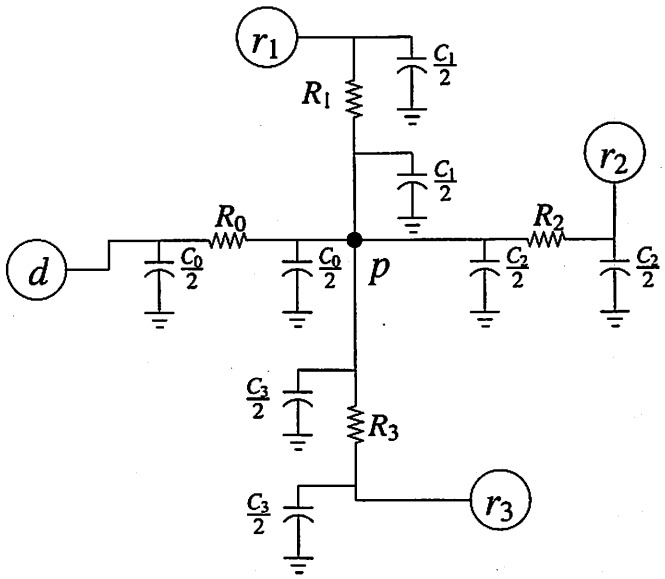



Figure 4.5: Delay model of a net.

# Chapter 5

# Global Placement

The placement problem is one of the first CAD problems to receive the attention of the research community. There has been a lot of research on this topic. The approaches to solve the placement problem can be classified into 3 categories:

- Partitioning based approaches
- Mathematical programming approaches
- Stochastic approaches

Examples of partitioning based placement approaches are [DK85] and [HK97]. The first work on mathematical programming approaches is by Cheng and Kuh [CK84], and the most successful stochastic placement is TimberWolf [SSV84].

Although these approaches all generate reasonably good results, we require that the placement approach in our integrated algorithm be *incremental*. This means that given an initial placement, the algorithm should only minimally perturb it while finding a new placement when new placement constraints are imposed. During logic operations in our integrated approach, new Boolean nodes are created in some cases while existing nodes are deleted in other cases. Therefore we additionally require that the placement algorithm find good positions for newly created Boolean nodes and fill in the voided space when nodes are deleted in the algorithm. Stochastic approaches use random moves within the algorithm and therefore not incremental. Partitioning based approaches are incremental but the placement quality is usually inferior to that of mathematical programming approaches. Moreover, partitioning based approaches are recursive in nature which means that it is difficult to accept an initial placement without executing the whole algorithm. A widely popular

placement tool which produces good results is GORDIAN [KSJA91]. Although quadratic programming is used extensively within the algorithm, GORDIAN is a mixed quadratic programming and partitioning based approach. Hence, it does not satisfy our requirements. An interesting placement algorithm developed in [EJ98] is called "Kraftwerk". It is a quadratic programming approach combined with an approach that mimics the behavior of a vector field. Kraftwerk matches all the requirements above.

We describe the Kraftwerk algorithm and our implementation of this tool in this chapter. We start by describing the formulation of placement as a quadratic programming problem in Section 5.1. In Section 5.2, we describe the formulation of Kraftwerk algorithm. The details of our implementation are described in Section 5.3.

## 5.1 Quadratic Placement

Let $C$ be the set of cells and $N$ be the set of nets in a circuit. The circuit is modeled as an undirected graph, called the *placement* graph $G$. The placement graph has a vertex for each cell. A net connecting $k$ cells is modeled as a clique of size $k$ in the graph. Let $C_m \subset C$ be the set of movable cells, and $C_f \subset C$ be the set of fixed cells, where $C_m \cup C_f = C$ and $C_m \cap C_f = \emptyset$. Let $(x_i, y_i)$ be the coordinate of the center of cell $c_i$. The cost of an edge $(c_i, c_j)$ of the placement graph $G$ is the squared Euclidean distance between $c_i$ and $c_j$, i.e. $(x_i - x_j)^2 + (y_i - y_j)^2$. The cost $f$ of a placement of the circuit is sum of the cost of all edges

$$f = \sum_{(c_i,c_j)\in G} w_{ij}(x_i - x_j)^2 + w_{ij}(y_i - y_j)^2$$

where $w_{ij}$ is the weight of the edge and is often used as a measure of criticality of the net. For ease of explanation, let us look at only the $x$ component of the cost function $f$ and denote it as $f_x$. Hence

$$f_x = \sum_{(c_i,c_j)\in G} w_{ij}(x_i - x_j)^2 = \sum_{c_i,c_j\in C} w_{ij}(x_i^2 - 2x_i x_j + x_j^2)$$

Rewriting $f_x$ in terms of movable cells and fixed cells, we get

$$f_x = \sum_{(c_i,c_j)\in G \wedge c_i,c_j\in C_m} w_{ij}(x_i^2 - 2x_i x_j + x_j^2) + \sum_{(c_i,c_k)\in G \wedge c_i\in C_m \wedge c_k\in C_f} w_{ik}(x_i^2 - 2x_i x_k + x_k^2) \qquad (5.1)$$

This cost function can be written in matrix form

$$f_x = \mathbf{x}^T\mathbf{A}\mathbf{x} - 2\mathbf{b}\mathbf{x} + const$$

where $\mathbf{x}$ is the vector of $x_i$ and $y_i$ (the size of $\mathbf{x}$ is $2|C_m|$, $|C_m|$ $x_i$ elements and $|C_m|$ $y_i$ elements) for each movable cell $c_i \in C_m$. The terms $w_{ij}x_i^2$ and $w_{ij}x_j^2$ of Equation 5.1 contribute $w_{ij}$ to the diagonal entry $a_{ii}$ and $a_{jj}$ of $\mathbf{A}$ respectively, the term $-2w_{ij}x_ix_j$ contributes $-w_{ij}$ to entries $a_{ij}$ and $a_{ji}$. of $\mathbf{A}$, the term $w_{ik}x_i^2$ contributes $w_{ik}$ to the diagonal entry $a_{ii}$ of $\mathbf{A}$, the term $-2w_{ik}x_ix_k$ contributes $w_{ik}x_k$ to row $i$ of $\mathbf{b}$ and finally the term $w_{ik}x_k^2$ is a constant.

Multiplying $f_x$ by $\frac{1}{2}$ generates the familiar quadratic programming equation

$$\frac{1}{2}f_x = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}\mathbf{x} + const \tag{5.2}$$

which is minimized by solving the linear equation system

$$\mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{0}$$

There are two important advantages of formulating the placement problem as Equation 5.2. First, the matrix $A$ is sparse, hence the storage requirement is linear in the size of vector $\mathbf{x}$. Second, the matrix $A$ is symmetric positive semi-definite, hence rather than using direct methods like LU decomposition, we can use *conjugate gradient* [GvL96] method to solve it. The conjugate gradient method is very efficient in both time and space. The most time consuming operation in this case is a multiplication of a sparse matrix and a vector which is linear in the number of edges in the placement graph $G$. Unlike LU decomposition, conjugate gradient does not introduce *fill-ins* which destroy the sparsity of the matrix and increase storage and run time.

The above formulation requires that there be some fixed cells. If there are no fixed cells, then $\mathbf{b} = \mathbf{0}$ and $\mathbf{x} = \mathbf{0}$ is a solution. For this reason, the positions of pins need to be known before the placement problem can be formulated as a quadratic programming problem.

Unfortunately, since cells that are not connected can overlap in the quadratic programming formulation, the solution tends to cluster all cells at the center of the placement area. The different quadratic placement algorithms in the literature usually differ in the technique they use to spread overlapping cells apart evenly in the placement area. For example, GORDIAN starts with the placement area as a single partition, and repulsive forces are introduced from the center of gravity of the partition to all cells in order to spread cells apart. It iteratively partitions each partition into two smaller partitions. The iteration stops when a minimum sized partition is reached.

## 5.2 Kraftwerk Algorithm

Kraftwerk [EJ98] is a quadratic placement algorithm with an interesting algorithm to spread overlapping cells apart from the center of the placement area. For completeness, we briefly

describe the algorithm. Kraftwerk introduces a force from each point in the placement area to every cell in the circuit. From the set of all such forces acting on a cell, the resultant force acting on the cell is determined and added to the vector **b** in Equation 5.2, written as a new vector **e**

$$\frac{1}{2}f_x = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - (\mathbf{b}+\mathbf{e})\mathbf{x} + const \tag{5.3}$$

Let the additional force $\vec{f}_i$ at cell $c_i$ with location $(x_i, y_i)$ be $\vec{f}_i = \vec{f}(x,y)|_{x=x_i, y=y_i}$. Let the rectangle function $R(z)$ be

$$R(z) = \begin{cases} 1 & \text{if } -\frac{1}{2} < z < \frac{1}{2} \\ 0 & \text{otherwise} \end{cases}$$

Let the width and height of $c_i$ be $w_i$ and $h_i$ respectively, then $a_i(x,y)$ is defined as an indicator function which is one if the point $(x,y)$ is covered by cell $c_i$, and zero otherwise. Using the rectangle function defined above, $a_i(x,y)$ can be written as

$$a_i(x,y) = R\left(\frac{x-x_i}{w_i}\right) R\left(\frac{y-y_i}{h_i}\right)$$

Similarly, an indicator function $A(x,y)$ is defined which is one if the point $(x,y)$ is within the placement area and zero otherwise. If the center of the placement area is $(x_A, y_A)$, and the width and height of the placement area are W and H respectively, then $A(x,y)$ can be written in terms of the rectangle function as

$$A(x,y) = R\left(\frac{x-x_A}{W}\right) R\left(\frac{y-y_A}{H}\right)$$

For an evenly distributed placement, the density of each point should be

$$s = \frac{\sum_{c_i \in C} w_i h_i}{WH}$$

For a particular placement, the density of each point $D(x,y)$ can be computed by

$$D(x,y) = \sum_{c_i \in C} a_i(x,y) - sA(x,y)$$

which means that the density of a point $(x,y)$ is the number of cells covering the point minus the desired density $s$. Hence, $D(x,y)$ is positive if there are fewer than $s$ cells covering the point and negative otherwise. For a typical placement problem, $s < 1$ because there is not enough area to place all cells with no overlaps.

With the above definitions of densities, the purpose is to move cells away from higher density regions to lower density regions. With proportional constant $k$, the force at point $(x,y)$ can be described as

$$\vec{\nabla}\vec{f}(x,y) = k \cdot D(x,y) \tag{5.4}$$

Since the purpose of adding forces is to evenly distribute the cells in the placement area, forces are required not to form circles. In other words, $\vec{f}(x,y)$ is conservative, i.e. there exists a scalar function $\Phi(x,y)$ with

$$\vec{\nabla}\Phi(x,y) = \vec{f}(x,y) \tag{5.5}$$

Combining 5.4 and 5.5 results in the Poisson's equation

$$\Delta\Phi(x,y) = k \cdot D(x,y) \tag{5.6}$$

with boundary conditions

$$\lim_{|\vec{r}|\to\infty} |\vec{\nabla}\Phi(x,y)| = 0, \text{with } \vec{r} = (x,y)^T$$

which has a unique solution for $\vec{f}(x,y)$

$$\vec{f}(x,y) = \frac{k}{2\pi}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} D(x',y')\frac{\vec{r}-\vec{r'}}{|\vec{r}-\vec{r'}|^2}dx'dy'$$

## 5.3 Implementation of Kraftwerk Algorithm

In this section, we describe our implementation of the Kraftwerk algorithm described above. The pseudo-code of the algorithm is as follows:

**Kraftwerk**()
1 build matrix **A** and vector **b**
2 solve $\mathbf{Ax} - \mathbf{b} = 0$ for **x** using the Conjugate Gradient method
3 **while** (*stopping criterion* is not met) **do**
4 compute **e**
5 update **A** and **b**
6 solve $\mathbf{Ax} - \mathbf{b} = 0$ for **x** using the Conjugate Gradient method
6 **end while**

The detail of the algorithm is explained below.

### 5.3.1 Conjugate Gradient

At the core of the Kraftwerk algorithm is the quadratic programming problem. As described in Section 5.1, the quadratic programming problem can be solved using conjugate gradient method. Given matrix **A**, vector **b**, and initial guess of the solution $\mathbf{x}_0$, the conjugate gradient algorithm computes the solution **x** as follows:

**ConjugateGradient(A, b, $\mathbf{x}_0$)**
1 $k = 0$
2 $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$
3 **while** $\mathbf{r}_k \neq 0$ **do**
4 $k = k + 1$
5 **if** $k = 1$ **then**
6 $\mathbf{p}_1 = \mathbf{r}_0$
7 **else**
8 $\beta_k = \mathbf{r}_{k-1}^T \mathbf{r}_{k-1} / \mathbf{r}_{k-2}^T \mathbf{r}_{k-2}$
9 $\mathbf{p}_k = \mathbf{r}_{k-1} + \beta_k \mathbf{p}_{k-1}$
10 **end if**
11 $\alpha_k = \mathbf{r}_{k-1}^T \mathbf{r}_{k-1} / \mathbf{p}_k^T \mathbf{A}\mathbf{p}_k$
12 $\mathbf{x}_k = \mathbf{x}_{k-1} + \alpha_k \mathbf{p}_k$
13 $\mathbf{r}_k = \mathbf{r}_{k-1} + \alpha_k \mathbf{A}\mathbf{p}_k$
14 **end while**
15 $\mathbf{x} = \mathbf{x}_k$
16 return **x**

### 5.3.2 Net Weights

Net weights may be used to minimize the lengths critical nets. This is often done by timing-driven placement algorithms like Ritual [SCK92]. Weights are increased for critical nets to increase their contribution to the cost function which results in shorter critical net lengths. In our implementation, we keep all net weights equal to unity. This means that the weight of each edge in the placement graph $G$ of a net of size $n$ (i.e. $n-1$ fanouts) is given by $w_{ij} = \frac{1}{n(n-1)}$. This is because each net is modeled as a clique as described in the construction of the placement graph $G$ in Section 5.1.

### 5.3.3 Discretization

The formulation of the Kraftwerk algorithm applies on an infinite space and for infinite number of points in the placement area. In our implementation, the space is approximated by $2W \times 2H$, where $W$ and $H$ are the width and height of the placement area. Let this space be called

the *working area*. The working area is then divided into grids. Then quantities like densities in the algorithm are computed only for these grid points. Quantities for points other than the grid points are interpolated from their values for the surrounding grid points. Figure 5.1 shows the working area and the grid points.

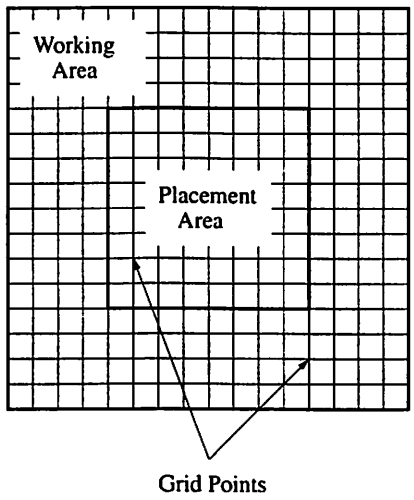



Figure 5.1: Working area, placement area, and grid points.

The number of grid points is determined by the smallest cell in the circuit. Let $w_c$ be the width of a narrowest cell and $h_c$ be the height of a shortest cell. The number of grid points is $g_x \times g_y$, where $g_x$ is the smallest multiple of 2 that is larger than $2W/w_c$ and $g_y$ is the smallest multiple of 2 that is larger than $2H/w_h$. We require that $g_x$ and $g_y$ be multiples of two for the Fast Fourier Transform computation purposes (described in the next section).

With such a discretization scheme, we can compute $D(x,y)$ for all the grid points. We then solve the Poisson equation shown in Equation 5.6.

### 5.3.4 Poisson Equation

The Poisson equation can be solved by convoluting the right hand side of Equation 5.6 with Green's function

$$G(x,y) = \ln\left(\sqrt{x^2+y^2}\right)$$

to get

$$\Phi(x,y) = k'D(x,y) \star G(x,y)$$

If we take the Fourier transform of this equation, we get

$$\Phi^{\mathcal{F}}(\omega,\tau) = \mathcal{D}(\omega,\tau) \cdot \mathcal{G}(\omega,\tau)$$

where $\Phi^{\mathcal{F}}(\omega,\tau)$, $\mathcal{D}(\omega,\tau)$, and $\mathcal{G}(\omega,\tau)$ are the functions $\Phi(x,y)$, $D(x,y)$, and $G(x,y)$ in Fourier space.

The Green's function $G(x,y)$ only depends on the $x$ and $y$ values for each grid point and thus can be easily precomputed. To compute the Fourier transform, we use the Fast Fourier Transform package *FFTW* [FJ98].

Finally, $\vec{f}(x,y)$ can be computed from Equation 5.5.

### 5.3.5 Dimensionless Cells

After the forces are computed, they need to be added into the quadratic programming formulation. This is handled by introducing a fixed cell for each cell in the circuit and connect them through a net. These fixed cells have no dimension and are called *dimensionless* cells. The positions of the fixed cells are determined by the additional forces acting on their corresponding cells as follows. First, a scaling factor $S$ is computed. From all the forces found for the grid points, let the force with the largest magnitude be $\vec{f}_{\max}$. The scaling factor $S$ is found by dividing the quarter perimeter of the placement area by the magnitude of $\vec{f}_{\max}$

$$S = \frac{W+H}{2|\vec{f}_{\max}|}$$

Then the positions of all dimensionless cells are found by scaling the additional forces acted on their corresponding cells by $S$, i.e. for cell $c_i$ with location $(x_i,y_i)$ and the additional force $\vec{f}(x_i,y_i) = f_x\mathbf{i} + f_y\mathbf{j}$, which is interpolated if $(x_i,y_i)$ is not one of the grid points, the location of the dimensionless cell is $(x_i + f_x \times S, y_i + f_y \times S)$.

### 5.3.6 Iteration Control

Let $d_i$ be the dimensionless cell of cell $c_i$. Let the net connecting cell $c_i$ to $d_i$ be $n_i^d$. While the weight of each net in the circuit is 1, the weight $w_i^d$ of net $n_i^d$ is changed from iteration to iteration. In the first iteration, $d_i$ does not exist since we have not introduced any additional force. In the subsequent iterations, the weight $w_i^d$ is scheduled according to the function

$$w_i^d(iter) = \begin{cases} w_i^d(iter-1) + a \times iter^2 + b \times iter & \text{if } 1 \le iter \le 50 \\ w_i^d(iter-1) + a \times 50^2 + b \times 50 & \text{if } iter > 50 \end{cases}$$

where *iter* is the iteration count and $w_i^d(0) = 0$, $a = 4 \times 10^{-7}$, and $b = 2 \times 10^{-5}$. This function is shown in Figure 5.2. In the initial iterations, the weight of $n_i^d$ is kept to be small to allow the connectivity of the circuit to determine the relative positions of cells. The weight increases as objective of later iterations is to evenly spread the cells over the placement area.

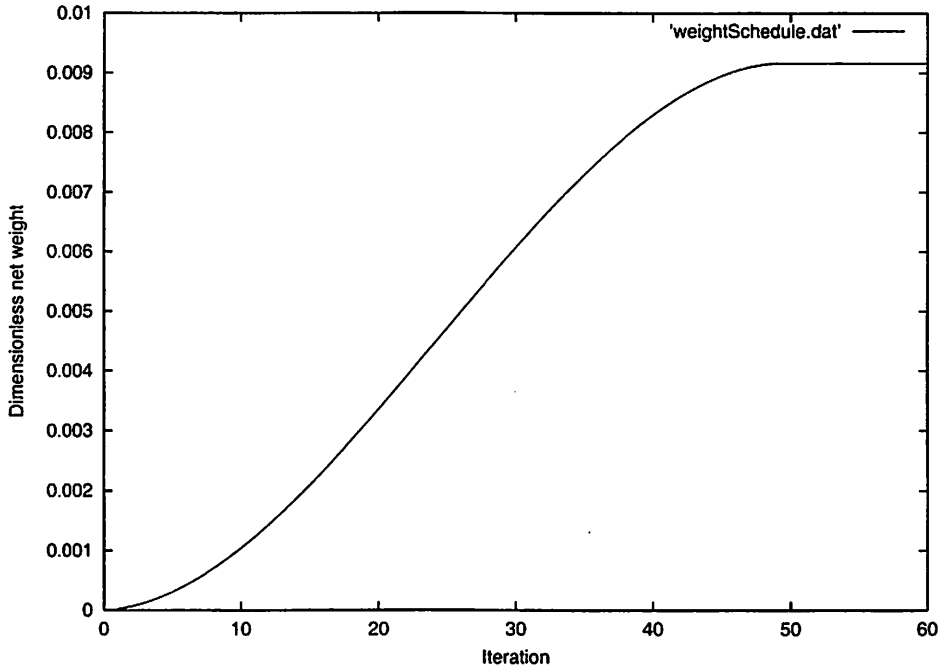



Figure 5.2: Weight schedule.

To better understand the spreading algorithm, we illustrate the procedure in Figure 5.3. After the first iteration, the additional force $\vec{f}_i(1)$ is computed for cell $c_i$ as shown in Figure 5.3(a). Then the position of the dimensionless cell $d_i$ is computed. The weight of the net connecting $c_i$ to $d_i$ is computed according to the above weight equation and it is shown in Figure 5.3(b). Figure 5.3(c) and Figure 5.3(d) show the additional force $\vec{f}_i(2)$ and position of $d_i$ in the second iteration, and Figure 5.3(e) and Figure 5.3(f) show the additional force $\vec{f}_i(3)$ and position of $d_i$ in the third iteration.

An iteration by iteration progess of the Kraftwerk algorithm is illustrated in Figure 5.4. In this figure, a picture of the placement is drawn after every 6 iterations.

### 5.3.7 Incremental Kraftwerk Algorithm

We noted above that the advantage of Kraftwerk over other algorithms is its ability to run in incremental mode. Given an initial placement, i.e. $\mathbf{x}$ vector, we can compute the $\mathbf{b} + \mathbf{e}$ vector

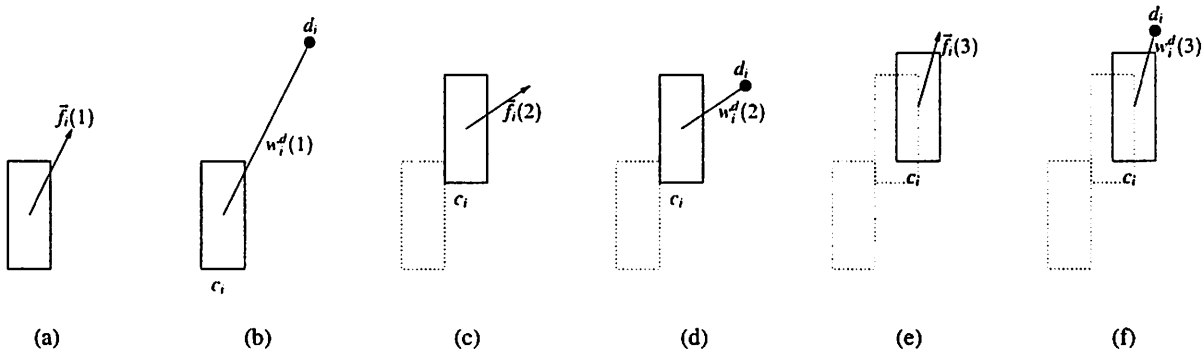



Figure 5.3: Illustration of spreading phase of Kraftwerk.

which is $\mathbf{Ax}$. The $\mathbf{e}$ vector can be easily computed since $\mathbf{b}$ depends only on the connectivity of the circuit. After $\mathbf{e}$ is found, the algorithm continues with adding additional forces until the stopping criterion is reached.

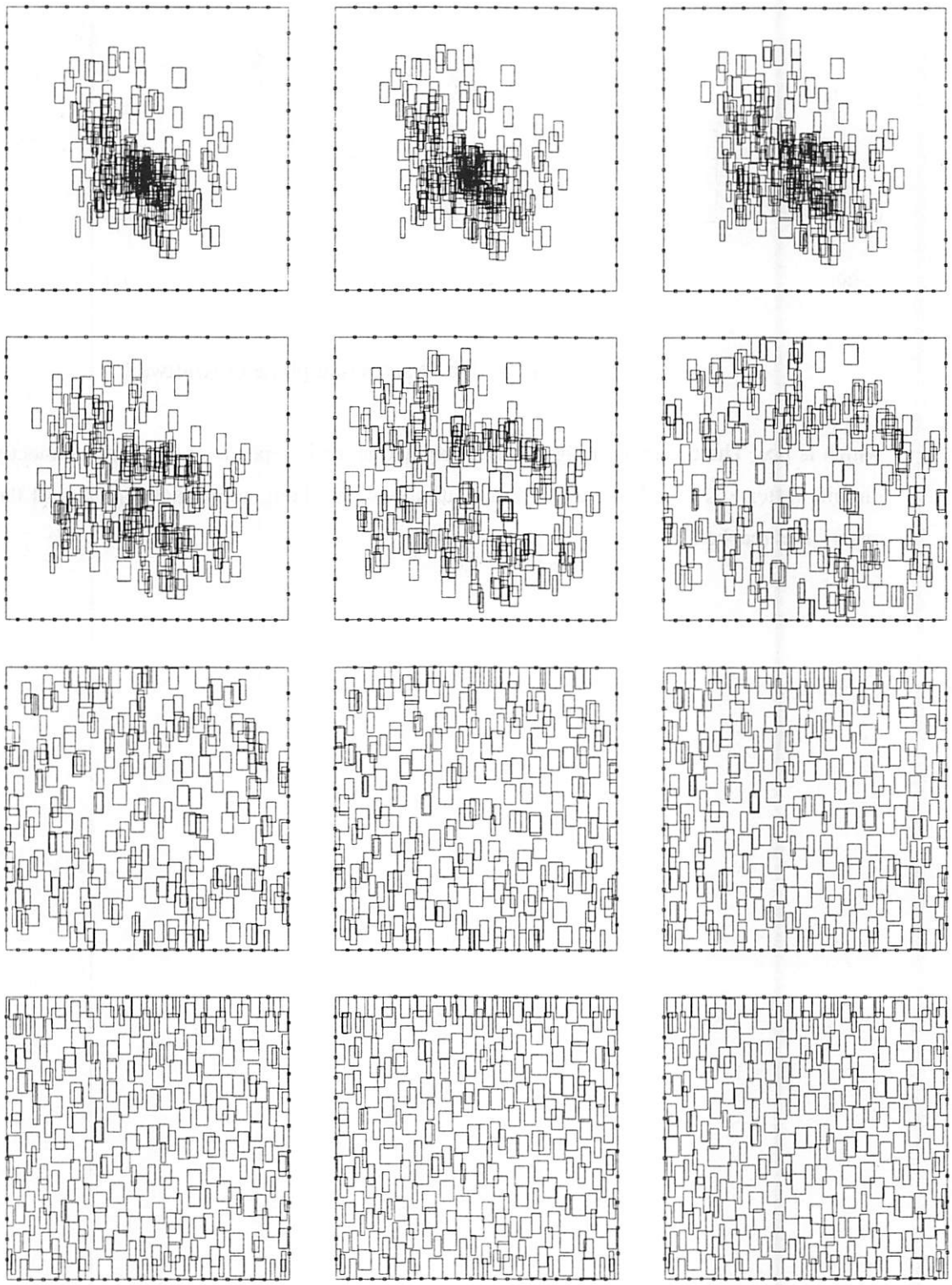


Figure 5.4: Execution of Kraftwerk.

# Chapter 6

# Technology Dependent Optimization

Logic synthesis is a process of reading a *high-level* description of a circuit to generate a *gate-level* description of the circuit while minimizing some cost function like area and delay. It is typically divided into two optimization steps, technology independent optimization and technology dependent optimization [SSL+92]. In technology independent optimization, the high-level description of the circuit is transformed into a Boolean network of logic functions, which is a directed acyclic graph (DAG) where each node of the graph represent a logic function. The cost function is typically modeled by literal count of the Boolean network. Logic operations are then performed on the network to minimize its literal count. In technology dependent optimization, the optimized Boolean network is mapped into a library of gates. In this step, the cost function can be directly estimated because library gates are used. The Boolean network is mapped using mapping algorithms that minimizes the cost function. In the proposed integrated logic synthesis and placement approach, the cost function can be computed more accurately in the technology dependent step because library gates are used and the circuit is closer to the final circuit as compared to the technology independent step. For this reason, we address the technology dependent step in this chapter and the technology independent step in the next chapter.

The technology dependent optimization process is divided further into technology decomposition and technology mapping. Technology decomposition is the process of decomposing a Boolean network (representing the circuit to be implemented) into primitive gates, e.g. 2-input NOR gates and inverters. During technology mapping, the decomposed Boolean network (which consists of only primitive gates) is mapped into library gates.

## 6.1 Local Placement

In practice, we cannot repeat the placement of all nodes in the Boolean network for every logic operation and cost function while our algorithm performs its computation. This would result in excessive run time. For cost computation, and when the Boolean network is minimally perturbed during logic operations, we *locally* place the affected nodes. We require that the local placement results and the final placement result are similar. To achieve this, the net model and algorithm used for the local placement must be the same as those used in the global placement algorithm. Since we use a quadratic global placement tool, we locally place nodes by formulating the local placement problem as a quadratic programming problem as well. Because the net model used in the quadratic global placement tool is the *clique model*, we use the same clique model in our local placement algorithm. The clique model is illustrated in Figure 6.1. Let node $n$ shown in Figure 6.1(a) be a new node, generated during logic synthesis optimizations. Let the positions of all fanout nodes, fanin nodes, and the fanouts of the fanin nodes (other that $n$ itself) be known. We first model all fanin and fanout nets of $n$ as cliques. The corresponding placement graph is shown in Figure 6.1(b). The quadratic programming problem is then formulated on this local placement graph. Since node $n$ is the only node without a position, the problem can be solved in constant time.

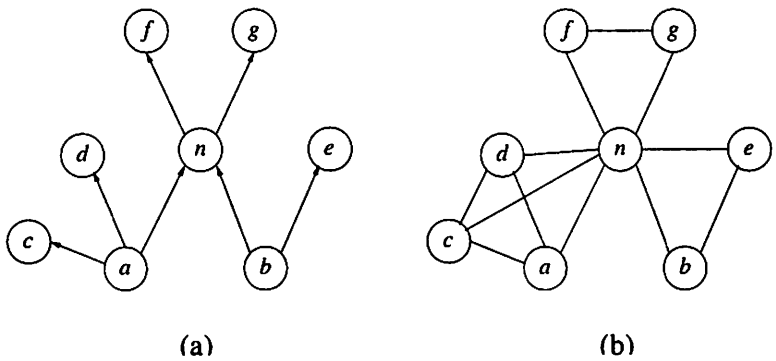



Figure 6.1: Incremental placement.

The idea of performing local placement with the same quadratic placement formulation and same net model as the global placement algorithm is one of the contributions of this thesis. Our results show that this minimizes the timing closure problem while resulting in a significant reduction in interconnect delay.

## 6.2 Technology Decomposition

As mentioned above, technology decomposition is a process of decomposing an optimized Boolean network whose nodes are usually complex logic functions into another Boolean network whose nodes are primitive gates (either 2-input NOR gates and inverters or 2-input NAND gates and inverters). This step is introduced in logic synthesis to reduce the complexity of the mapping of Boolean nodes into library gates. After this step, mapping a set of Boolean nodes into a library gate reduces to a graph isomorphism problem.

Our placement-aware technology decomposition algorithm decomposes a Boolean network using primitive gates in a manner that minimizes wire-lengths in the decomposed network. The algorithm consists of the steps outlined below. Here we describe the algorithm where each node is decomposed into 2-input NOR gates and inverters. The algorithm where each node is decomposed into 2-input NAND gates and inverters is similar.

1. We first invoke the global placement algorithm to find the positions of all nodes in the original (optimized) Boolean network.

2. Next we decompose every Boolean node $N$ of the original network into AND nodes (corresponding to each cube of the Boolean node $N$) and an OR node with all the AND nodes as its fanins. After all the nodes of the original network have been decomposed in this manner, we compute the positions of the new AND and OR nodes by invoking the global placement algorithm. Let the AND nodes be called $N_0, N_1, \cdots, N_{m-1}$, and let the OR node be called $N_m$. Here, the cardinality of the sum-of-products cover representing the logic function of $N$ is $m$.

3. For each AND or OR node $n \in \{N_0, N_1, \cdots, N_m\}$ with fanins $FI = \{f_1, f_2, \ldots, f_k\}$, we decompose $n$ into $n'$ with fanins $NI = \left\{n_1, n_2, \ldots, n_{\lceil \frac{k}{2} \rceil}\right\}$, where $f_i \in FI$. After such a decomposition step, each node $n_j \in NI$ is a 2-input AND or OR node with a pair of nodes in $FI$ as its fanins. We call this decomposition a *two-step decomposition*.

   The objective of the two-step decomposition process is to choose a pair of fanins from $FI$ for each $n_j$, so as to minimize the total wire-length of all the nets connected to the outputs of $NI$ and $FI$, where the positions of nodes $NI$ are computed utilizing the *local* placement algorithm. We call this problem the *fanin ordering* problem.

   Figure 6.2 illustrates this process. In this figure, all nodes are drawn to scale. Figure 6.2(a) shows a node $N$ in the original Boolean network being decomposed. Figure 6.2(b) shows

the decomposition of $N$ into AND and OR nodes as described in step 2. The logic function computed by the node $N$ is $f_1f_2f_3f_4f_5f_6f_7f_8 + f_9f_{10}$. Figure 6.2(c), shows node $N_0$ before two-step decomposition. For consistency in notation, $n$ is also used to refer to $N_0$ in Figure 6.2(c). The result of two step decomposition is shown in Figure 6.2(d). The fanin ordering problem essentially attempts to find a two-step decomposition of the nodes in Figure 6.2(c) such that the sum of wire-lengths of all wires in the resulting decomposition (Figure 6.2(d)) is minimized.

At the end of this step, not all nodes are 2-input nodes. For example, node $n'$ in Figure 6.2(d) has 4 fanins.

4. After all AND and OR nodes in the network have been decomposed using the two-step decomposition method, we run the global placement algorithm in incremental mode to update all node positions.

5. We then iterate steps 3 and 4 over all Boolean nodes of the network until all nodes have at most two fanins.

6. Finally, we run global placement on the resulting network of primitive gates. The reason for running the global placement at this stage is to minimize the overlaps between cells. Since the core algorithm and net model of both the local algorithm and the global placement algorithm are the same, the resulting placement is minimally perturbed.

The algorithm above decomposes a complex node into a balanced tree of primitive gates. Such decomposition ensures that no path becomes excessively long and increases the delay of the circuit. However, if delay information is available or different cost functions are being optimized, other decomposition algorithms can be performed. One such algorithm [Bra00] selects two nodes from the set of leaf nodes to pair to form a new node. The two selected nodes are removed from the set of leaf nodes and the new node is added to it. The procedure is repeated until all leaf nodes have been paired.

In addition to the invocations of the global placement algorithm described above, we additionally invoke global placement at most $\beta$ times (where $\beta$ is a user defined variable) during the technology decomposition algorithm. This is to ensure that our local placement runs utilize accurate node placement information at all times. We experimented with several values of $\beta$, and found that $\beta = 10$ resulted a good trade-off between run-time and circuit optimality.

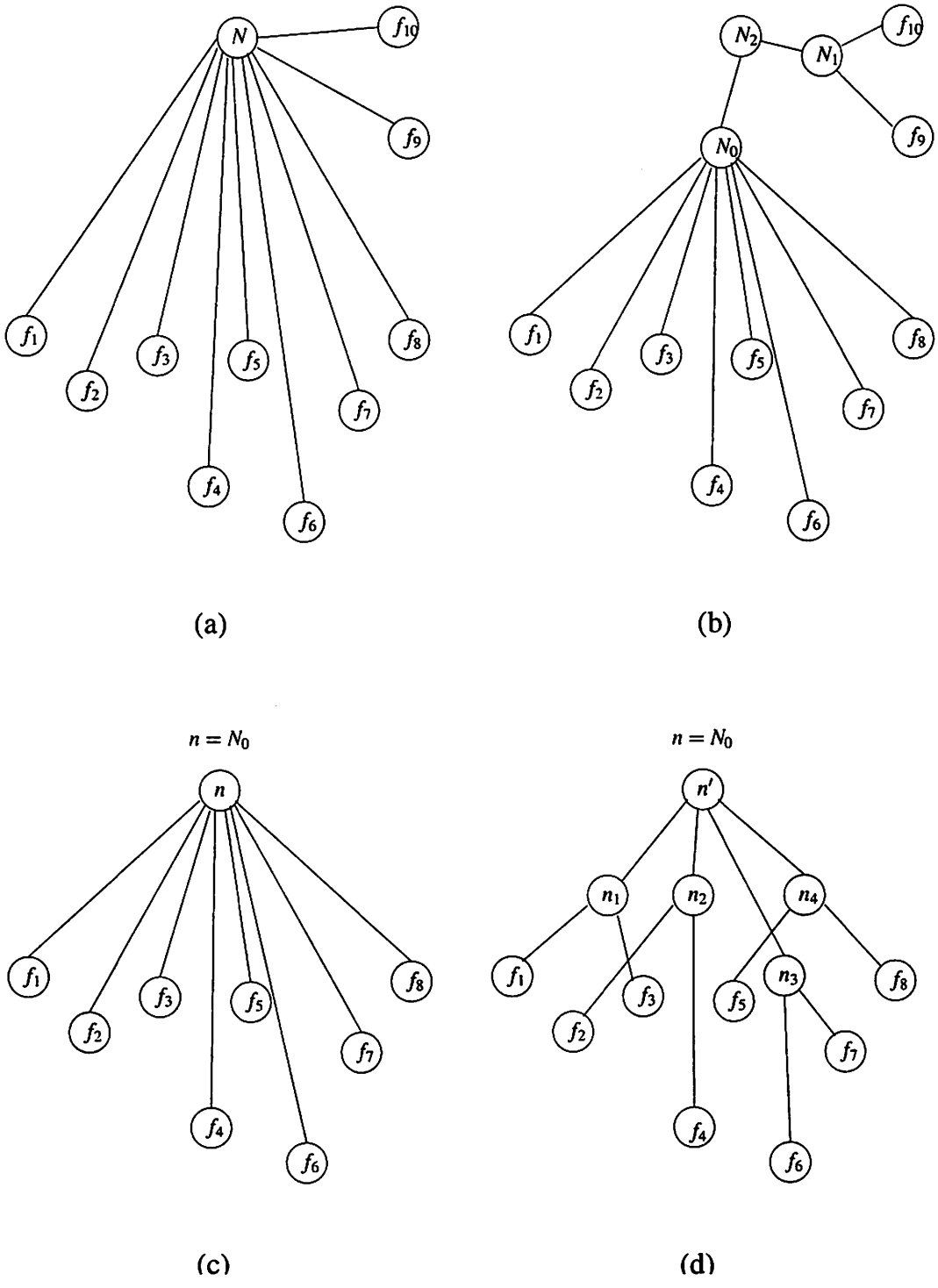



Figure 6.2: Technology Decomposition and Fanin Ordering Problem.

When the global and local placement are invoked during technology decomposition, the Boolean nodes are either 2-input NORs, inverters, or complex gates (i.e. NOR gates with more than two inputs). Each node needs to have a finite size for our global placement tool to execute. Since this is the early stage of the technology dependent optimization, we treat all Boolean nodes as points, which is handled by treating them as cells of equal sizes for placement. Since many nodes are created during technology decomposition, these cells are scaled while maintaining their aspect ratio. The size of a cell is scaled according to the number of nodes in the network such that the total area matches the available placement area. This results in minimum overlap throughout technology decomposition.

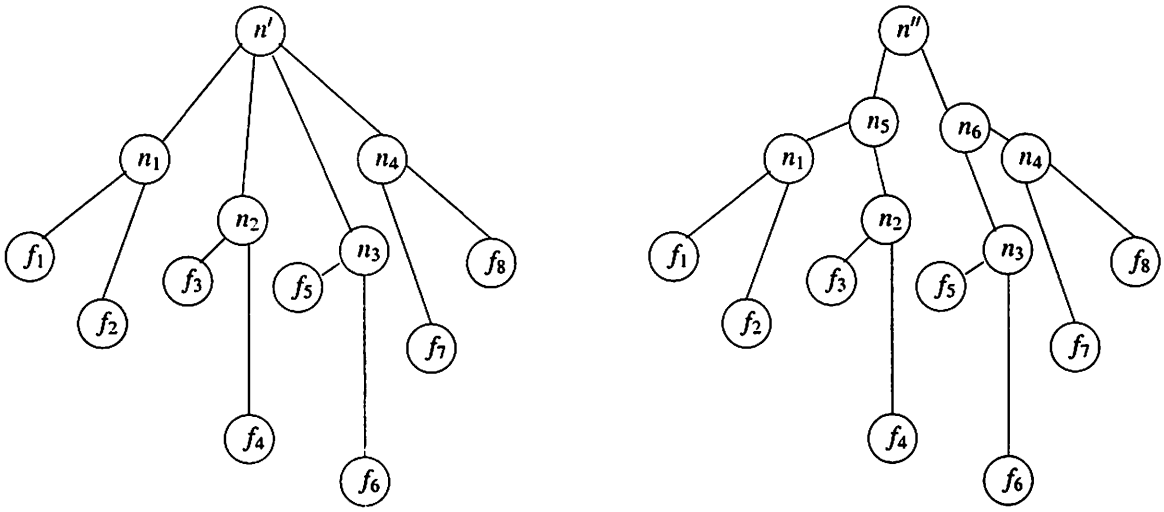
Rather than treating each Boolean node as points as what we have done in this thesis, the area of each Boolean node can be estimated according to the number of literals in the Boolean node. The area can be used to estimate a rectangle representing the Boolean node. All these rectangles can then be placed by the global placement tool. However, the placement of such a scheme is not likely to be better than the placement which treats each Boolean node as a point. The reason is that a complex Boolean node will be decomposed into small gates and the placement of these small gates will not be a rectangle. In fact, these small gates tend to spread across the placement area. Hence, the placement of big rectangles may be very different from the placement of the final circuit.

**Theorem 6.1** *The fanin ordering decision problem is NP-complete.*

**Proof**: In the decision problem, we ask the question if a decomposition whose total wire length is less than a constant $B$ exists. The problem is clearly in **P** because we can compute the total wire length given a decomposition and check if it is less than $B$. Let $n_{ij}$ be the new node obtained by pairing $f_i \in FI$ and $f_j \in FI$. The position of $n_{ij}$ can be computed using incremental placement as described above. Let $d(f_i, f_j)$ be the cost of pairing $f_i$ and $f_j$. The cost $d(f_i, f_j)$ is the sum of the length of nets $f_i$, $f_j$, and $n_{ij}$. We perform reduction from the *clustering* problem [GJ79]. Let the finite set $X$ of the clustering problem be the set of nodes $FI$. Let the distance between any pair $(x_i, x_j)$ of $X$ be $d(f_i, f_j)$ as described above. Then it is easy to see that there exists a partition of $X$ into $\lceil \frac{k}{2} \rceil$ disjoint sets such that the total distance is $\leq B$ if and only if there is a decomposition whose total wire length is $\leq B$. ∎

Since the fanin ordering problem is a hard problem, we utilize heuristics to solve it. The two heuristics that we use are *angle* ordering and *furthest-pair* ordering. In both heuristics, we look for a linear order $FI_s = (f_{s_1}, f_{s_2}, \ldots, f_{s_k})$ of $FI$. Then nodes $NI$ are created by pairing nodes in $FI_s$ in this linear order, i.e. by paring $f_{s_1}$ with $f_{s_2}$, $f_{s_3}$ with $f_{s_4}$, and so on.

In *angle ordering*, we traverse the fanins of node $n$ in a counter clockwise direction to form a circular order. The two consecutive fanins in this traversal that are furthest apart in terms of linear distance form the end points of the final linear order. The remaining nodes are ordered in the counter clockwise manner. For the example shown in Figure 6.2(c), the counter clockwise traversal gives the following circular order: $(f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_1, ...)$. The two consecutive fanins that are furthest apart in terms of linear distance are $f_1$ and $f_8$. Hence these two nodes form the two end points of the linear order, and this linear order is therefore $(f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8)$. A node is created and becomes the parent of each pair of nodes in this order. For this example, the new nodes are $\{n_1, n_2, n_3, n_4\}$ as shown in Figure 6.3(a). These four new nodes become the leaves of a new ordering problem for node $n'$ and are ordered in the next iteration in the same way. The angle order is $(n_1, n_2, n_3, n_4)$. The new nodes formed after pairing these nodes are $\{n_5, n_6\}$ as shown in Figure 6.3(b).




(a) (b)

Figure 6.3: Angle Ordering Solution.

In *furthest pair* ordering, we iteratively pair fanin nodes (i.e. nodes in $FI$) until there are no more nodes to pair. In each iteration, we first find the fanin $f_f \in FI$ that is furthest away from node $n$ in terms of linear distance. We then pair node $f_f$ with the unpaired fanin node $f_g \in FI$ that is closest to $f_f$ in terms of linear distance. Nodes $f_f$ and $f_g$ form the inputs to a new 2-input node in $NI$. The result of furthest pair ordering for the example given in Figure 6.2(c) is shown in Figure 6.4(a).

In this example, we first pair $f_6$ and $f_4$, then $f_1$ and $f_8$, then $f_2$ and $f_1$, and finally $f_3$ and $f_5$. The new nodes created are $\{n_1, n_2, n_3, n_4\}$. These nodes are ordered in the next iteration. The order of pairing is $n_3$ and $n_4$, and $n_1$ and $n_2$. The new nodes are $n_5$ and $n_6$ as shown in Figure 6.4(b).

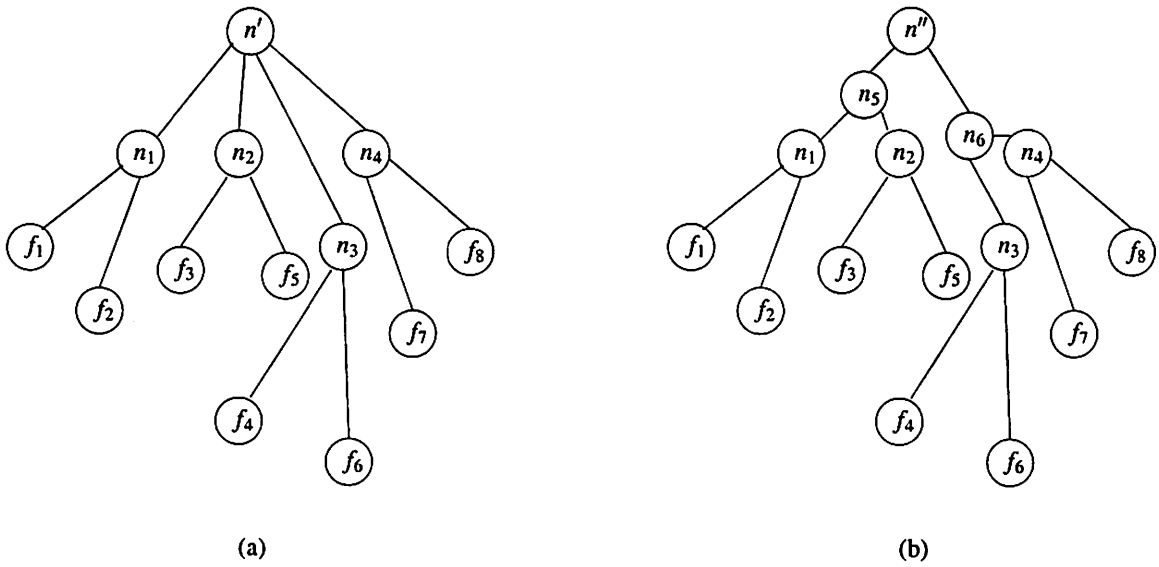



Figure 6.4: Furthest Pair Solution.

For each node, we perform technology decomposition using both the angle order and the furthest pair order and compute the cost of each order in terms of total net lengths. The cost of an order is the sum of the length of the fanin nets and the new nets (i.e. the total length of the nets in Figures 6.4 and 6.3).

## 6.3 Technology Mapping

After technology decomposition, the original Boolean network is transformed into a network consisting exclusively of primitive gates, i.e. 2-input NOR/NAND gates and inverters. This network is called the *subject graph*. The gates in the library are also decomposed using the same primitive gates, and each such decomposed gate is called a *pattern graph*. Technology mapping is the process of covering the subject graph with the pattern graphs while minimizing an objective function. For area minimization, the objective function is the total area of the mapped circuit. For delay minimization, the objective function is the total delay of the mapped circuit. If the subject graph is a tree, technology mapping with the objective function of area minimization can be solved

optimally using dynamic programming [?].

We describe our area and wire-length, and delay minimization algorithms below.

### 6.3.1 Area and Wire-Length Minimization

In our approach, we decompose the subject graph into trees and use dynamic programming to solve it. The dynamic programming algorithm consists of two steps: the forward propagation step to compute the cost of a best match and to store it on the node, and the backward tracing step to construct the match. The pseudo-code technology mapping algorithm implemented in this thesis is shown as the **AreaTreeMap()** procedure below. A match $m$ at node $n$ with gate $g$ is denoted as $m(n,g)$. The forward propagation step consists of lines 1 through 25 of the procedure. The remaining lines are pseudo-code of the backward tracing step, which include an invocation of the **TreeMapBuildNetwork()** procedure.

```
AreaTreeMap(𝒩, β)
1   N ← DfsFromPrimaryOutputs(𝒩)
2   maxCount ← |IN(𝒩)|/β
3   count ← 0
4   foreach node n ∈ N do
5     count ← count +1
6     if count = maxCount then
7       UpdateGlobalPlacement(𝒩)
8       count ← 0
9     end if
10    n.bestMatch ← ∅
11    n.bestCost ← ∞
12    foreach gate g do
13      foreach pattern graph G_p of gate g do
14        m(n,g) ← Match(G_p,n)
15        if m(n,g) ≠ ∅ then
16          (x_m,y_m) = LocalPlacement(m(n,g))
17          costOfMatch ← MapCost(m(n,g), x_m, y_m)
18          if (costOfMatch < n.bestCost) then
19            n.bestMatch ← m
20            n.bestCost ← costOfMatch
21          end if
22        end if
23      end for
24    end for
25  end for
```

```
26  N_mapped ← ∅
27  foreach primary output p ∈ PO(N) do
28    TreeMapBuildNetwork(N_mapped, p)
```

```
TreeMapBuildNetwork(N_mapped, n)
1   newNode ← createMappedNode(N_mapped, n.bestMatch)
2   n.mapped ← TRUE
3   foreach fanin f of n.bestMatch do
4     if f.mapped = FALSE then
5       TreeMapBuildNetwork(N_mapped, f)
6     end if
7   end for
```

The forward propagation step traverses the trees in topological order from primary inputs to primary outputs such that optimum matches for all fanins of a node $n$ are found before a match for $n$ is found. Let $n_m$ be the new node for match $m(n,g)$. Let $FI = \{f_1, f_2, \ldots, f_k\}$ be the fanins of $n_m$. We recursively define the *fanin wire cost* $w_I(n_m)$ of match $m(n,g)$ as the total length of the fanin nets of $n_m$ plus the fanin wire cost of all its fanins. Formally,

$$w_I(n_m) = \sum_{f_i \in FI} (w_I(n_{f_i}) + l(n_{f_i}))$$

where $n_{f_i}$ is the node of the best match at $f_i$ and $l(n_{f_i})$ is the length of its net, as illustrated in Figure 6.5. In this figure, node $n_m$ of match $m(n,g)$ has three fanins, which are $n_{f_1}$, $n_{f_2}$, and $n_{f_3}$ (Node $n_{f_1}$, $n_{f_2}$, and $n_{f_3}$ are the mapped nodes of the best matches at $f_1$, $f_2$, and $f_3$ respectively). Note that net $n_{f_i}$ is a two-terminal net, since the network being mapped consists of primitive gates which are 2-input gates, and we operate on tree decompositions of this network. Essentially, $w_I(n_m)$ is the total wire length of all nets in the mapped circuit rooted at node $n_m$.

Similarly, we recursively define the *area cost* $a(m)$ of match $m(n,g)$ as the area of $g$ plus the total area of all its fanin matches. The total cost $c(m)$ of match $m(n,g)$ is then the weighted sum of the area cost of $m(n,g)$ and the sum of the fanin wire cost of $m(n,g)$ plus the length of net $n$, or $c(m) = a(m) + \alpha(w_I(m) + l(n_m))$, where $\alpha$ is a user-defined weighting variable. The function **MapCost()** computes this total cost $c(m)$.

In the example of Figure 6.5, the fanin wire cost of match $m(n,g)$ is the sum of the lengths of nets $n_{f_1}, n_{f_2}$, and $n_{f_3}$, and the fanin wire costs of nodes $n_{f_1}$, $n_{f_2}$, and $n_{f_3}$. The area cost of match $m(n,g)$ is the sum of the area of $g$ and the area costs of nodes $n_{f_1}$, $n_{f_2}$, $n_{f_3}$.

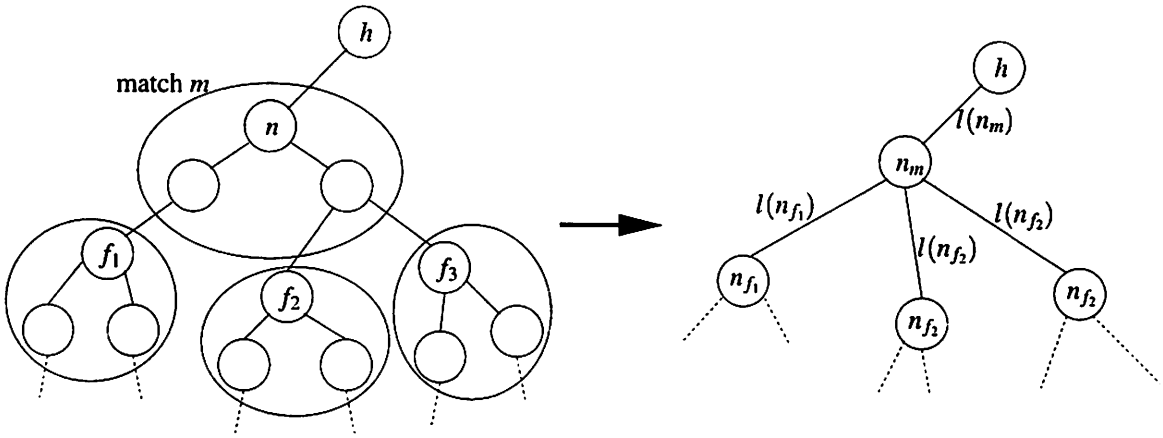



Figure 6.5: Definition of Cost Elements.

In order to compute the cost of a match, the position of the new node $n_m$ corresponding to the match needs to be computed (i.e. the node which is shown in the ellipse containing $n$ in Figure 6.5). As in technology decomposition, we could re-run global placement on all nodes in the design (including the new node). However, this is too time consuming and so we use the local placement algorithm described in Section 6.1 to estimate the position of the new node.

Even though the local placement algorithm uses the same net model and the same quadratic placement based formulation as the global placement algorithm, after a certain number of nodes have been matched, the global placement algorithm is run on the entire circuit. Just as in technology decomposition, the user defined parameter $\beta$ is used here. During the whole technology mapping algorithm, the global placement algorithm is called at most $\beta$ times. All other executions of the placement algorithm during technology mapping are in incremental mode.

While technology mapping is being performed, not all nodes are mapped in general when the global placement algorithm is invoked. The unmapped nodes are primitive nodes: NAND/NOR gates and inverters. Before calling global placement, NAND/NOR and inverter nodes are temporarily mapped into NAND/NOR and inverter gates in the library. This temporarily mapped network is then placed. As in technology decomposition, all cells are scaled to match available placement area to minimize overlap.

### 6.3.2 Updating Placement

As mention above, the global placement algorithm is invoked at most $\beta$ times during technology dependent optimization. The positions of all cells are represented by the **x** vector as in Equation 5.2 in each placement invocation and solved using the Kraftwerk algorithm described in Chapter 5. After each invocation, the positions of all cells stored in **x** are copied back to the Boolean network. The procedure described in this section is computed by **UpdatePlacement**() function in the pseudo-code above.

In technology decomposition, the positions of all nodes are used as initial solution. Each Boolean node has an entry in **x** and the final solution of the placement algorithm is updated to the Boolean nodes accordingly. However, the update procedure is more complex as described below.

To increase the effectiveness of the technology mapping algorithm, inverters are added to the decomposed Boolean network before the mapping algorithm begins without changing the functionality of the Boolean network. Two inverters are connected in series to each node. The mapping algorithm however removes extra inverters implicitly by mapping two extra inverters in series to a *wire*. The number of inverters that are added is about twice the number of nodes in the Boolean network. Hence, extra inverters need to be removed before invoking global placement during mapping, in order for the placement result to mimic the placement of the circuit after mapping is done.

When the global placement is invoked during technology mapping, there are mapped as well as unmapped nodes in the network. Let the Boolean network being mapped be $\mathcal{N}$. For placement purposes, a mapped network $\mathcal{N}_p$ is created using the existing matches. Network $\mathcal{N}_p$ has the same primary inputs and primary outputs as $\mathcal{N}$ does. An unmapped node in $\mathcal{N}$ is either a 2-input NOR gate or an inverter. These unmapped nodes are mapped to the smallest corresponding gates in the library and new nodes associated with them are created in $\mathcal{N}_p$. For example, let node $n$ be a mapped node in $\mathcal{N}$ whose fanouts are not mapped. If the best match of $n$ is a gate $g$, then a node with gate $g$ is created in $\mathcal{N}_p$. For a node $n$ in $\mathcal{N}$, we denote the node associated with $n$ in $\mathcal{N}_p$ as $\mathcal{N}_p(n)$. Similarly, the node associated with a node $n$ in $\mathcal{N}_p$, the associated node is $\mathcal{N}(n)$. Hence, $\mathcal{N}(\mathcal{N}_p(n)) = n$. We also denote the fanins of the match $m(n,g)$ as $FI_m$. Since the network is decomposed into trees before mapping, the nodes that are matched by $m(n,g)$ can be computed from $FI_m$ by recursively traversing the fanins of $n$ as long as the fanin is not a node in $FI_m$. The association of $\mathcal{N}$ and $\mathcal{N}_p$ is shown in Figure 6.6. In this figure, inverters $n_1$ and $n_2$ are matched by a wire. Nodes $n_3$, $n_4$, and $n_5$ are matched by $m_3$ with a 2-input AND gate. The association of $n_3$ and $n_3'$ is as shown. The fanins of match $m_3$ or $FI_{m_3}$ contains $n_6$ and $n_7$.

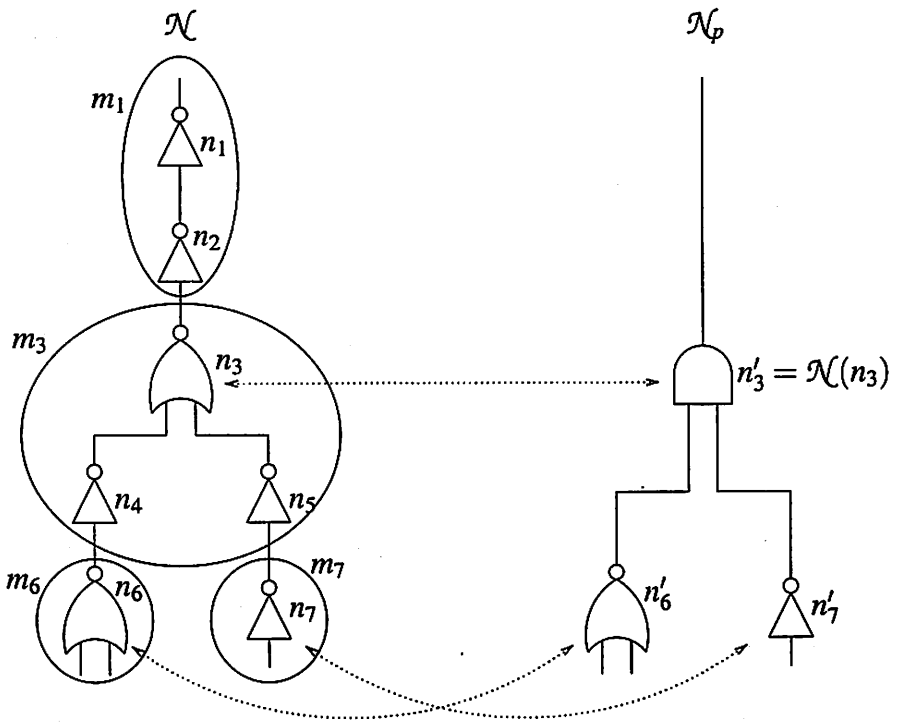



Figure 6.6: Creation of Boolean network for placement.

The mapped network $\mathcal{N}_p$ is the network placed by the global placement algorithm after extra inverters have been removed. Beside removing extra inverters in series, the inverter removal algorithm also removes extra inverters connected in parallel. An example of this is shown in Figure 6.7. In this picture, inverters $n_1$ and $n_3$ are removed.

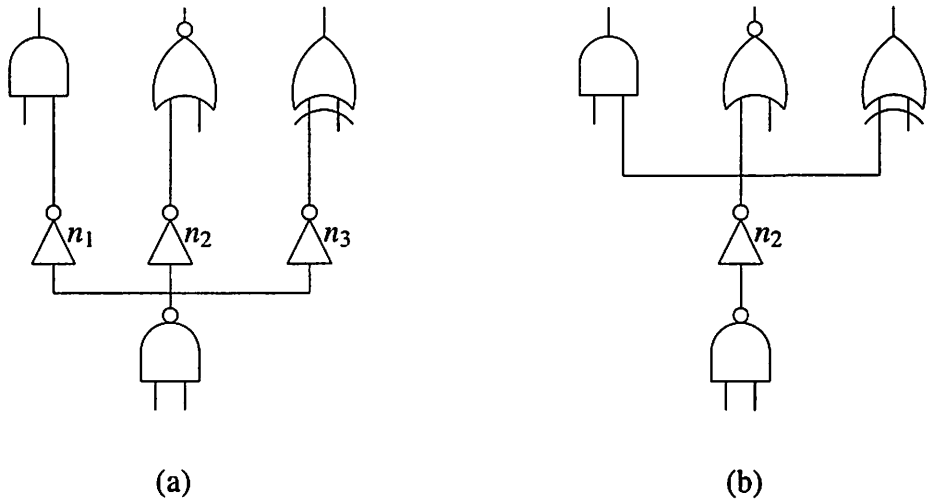



Figure 6.7: Removing parallel inverters.

After network $\mathcal{N}_p$ has been placed, the coordinates of all nodes are copied back to network $\mathcal{N}$. As seen from Figure 6.6, not all nodes in $\mathcal{N}$ has an associated node in $\mathcal{N}_p$. The algorithm to back annotate the coordinate for a node $n$ in $\mathcal{N}$ is as follows:

1. If $n$ has an associated node in $\mathcal{N}_p$, update the coordinate of $n$ with the coordinate of $\mathcal{N}_p(n)$ and the algorithm terminates.

2. If $n$ is part of a match $m_i$ of node $n_i$, update the coordinate of $n$ with the coordinate of $\mathcal{N}_p(n_i)$ and it terminates.

3. If $n$ is an inverter and there is a parallel inverter $n_i$ which has an association node $\mathcal{N}_p(n_i)$, then update the coordinate of $n$ with the coordinate of $\mathcal{N}_p(n_i)$.

4. Interleave the traversal of the fanin and fanout recursively until node $n_i$ which has an association node $\mathcal{N}_p(n_i)$ is found, then update the coordinate of $n$ with the coordinate of $\mathcal{N}_p(n_i)$.

After the algorithm terminates, all nodes in $\mathcal{N}$ have updated coordinates.

### 6.3.3 Delay Optimization

The objective of the delay optimized technology mapping is to minimize the largest arrival time among all primary outputs of the circuit being mapped.

As seen from the delay model shown in Section 4.3, the delay of a gate driving a wire and a fixed capacitive load depends not only on the intrinsic delay of the gate but also on the load seen at the output of the gate, i.e. the wire capacitance and the load. The interconnect delay depends on the length of the wire. If the load seen by any gate in a library consists of only a small number of distinct load values and there is no interconnect delay, dynamic programming can be used to find an optimum solution when the subject graph is a tree [Rud89]. The solution is to put the distinct load values in bins and to compute the optimum solution for each load value. Unfortunately, since the interconnect load of a gate varies depending on the placement of cells in a circuit, the load seen by a cell cannot be put in bins of distinct values. Hence, delay optimized dynamic programming is only an approximation to the optimum solution. Nonetheless, it has generated good results [Rud89] [Tou90].

Our delay optimized technology mapping used here is dynamic programming based. As explained in Section 6.3.1, the forward propagation step of the dynamic programming algorithm traverses the network such that a node $n$ is visited in a topological order. At each node $n$, all matches of $n$ are evaluated. The match that results in the earliest arrival time at $n$ is the considered the best match for that node. However, when computing the arrival time of $n$, its load is not known because nodes in its fanout have not been visited. Depending on how the arrival time is computed and how the load is estimated, three different methods are described in this thesis, which are called *fixed load* method, *single match* method, and *multiple match* method.

When a match $m(n,g)$ of a node $n$ is being considered, the location of the match is computed using the local placement algorithm described above. The location of the match $m(n,g)$ is denoted as $(x_m, y_m)$.

The pseudo-code of the technology mapping algorithm is shown below. Depending on the methods used, line 17 is replaced by a call to **FixedLoadCost()**, **SingleMatchCost()**, and **MultipleMatchCost()**. For the multiple match method, line 24 is replaced by **MultMatchBuildNetwork()**, which will be described later.

**DelayTreeMap**($\mathcal{N}$, $\beta$)
1 $N \leftarrow$ **DfsFromPrimaryOutputs**($\mathcal{N}$)
2 $maxCount \leftarrow |IN(\mathcal{N})|/\beta$

```
3   count ← 0
4   foreach node n ∈ N do
5     count ← count +1
6     if count = maxCount then
7       UpdateGlobalPlacement(𝒩)
8       count ← 0
9     end if
10    n.bestMatch ← ∅
11    n.bestDelay ← ∞
12    foreach gate g do
13      foreach pattern graph G_p of gate g do
14        m ← Match(G_p, n)
15        if m ≠ ∅ then
16          (x_m, y_m) = LocalPlacement(m)
17          compute match cost and store match information
18        end if
19      end for
20    end for
21  end for
22  𝒩_mapped ← ∅
23  foreach primary output p ∈ PO(𝒩) do
24    TreeMapBuildNetwork(𝒩_mapped, p)
```

#### 6.3.3.1 Fixed Load Method

The fixed-load cost computation algorithm is shown below. In the pseudo-code, $\alpha(i)$ and $\beta(i)$ denotes the intercept and slope of input pin $i$ of the gate being processed. For clarity purposes, the arrival time at a node is shown as a scalar. In reality, it consists of rise and fall values. The function **ComputePinLoad**() computes the sum of the total pin capacitances at the output of a node $n$. The function **ComputeWireLoad**() computes the capacitance of a net $n$.

```
FixedLoadCost(𝒩, m(n,g), x_m, y_m)
1   n.load ← ComputePinLoad(n) + ComputeWireLoad(m(n,g), x_m, y_m)
2   n.arrival ← 0
3   foreach fanin f_i of m(n,g) do
4     delay ← α(i) + β(i) × n.load
5     if (f_i.arrival + delay) > n.arrival then
6       n.arrival ← f_i.arrival + delay
7       n.bestMatch ← m(n,g)
8     end if
9   end for
```

During mapping, nodes that have not been visited are either 2-input NOR/NAND gates or inverters. In the *fixed load* method, the arrival time of node $n$ for match $m(n,g)$ is computed assuming that its fanouts are mapped to the smallest gates (i.e. smallest NOR/NAND gates or inverters) in the library that match them. In other words, if one of the fanouts of $n$ is a 2-input NOR gate, then its load due to that fanout is the load of one of the input pins of the smallest 2-input NOR gate in the library. The choice of which input pin is determined at random. Using $(x_m, y_m)$ and the locations of all its fanouts, the wire length of interconnect seen by $n$ is computed using the semi-perimeter model as described in Section 4.3.1. The interconnect delay due to the resistance of the wire is neglected. If match $m(n,g)$ is the first match considered for $n$, then the match and the arrival time are stored at $n$. Otherwise, if the arrival time is smaller than the arrival time stored at $n$, match $m(n,g)$ is stored at $n$, along with the new arrival time. After all matches for $n$ have been considered, the arrival time of the best match is later used to compute the arrival times of the fanouts of $n$. In the fixed load method, the arrival times are not recomputed.

#### 6.3.3.2 Single Match Method

The single match cost computation algorithm is shown below.

```
SingleMatchCost(N, m(n,g),x_m,y_m)
1   n.load ← ComputePinLoad(n) + ComputeWireLoad(m(n,g),x_m,y_m)
2   n.arrival ← 0
3   foreach fanin f_i of m(n,g) do
4     pinArrival ← ComputeElmoreDelay(f_i.intercept, f_i.slope, m(n,g),x_m,y_m)
5     delay ← α(i) + β(i) × n.load
6     if (pinArrival + delay) > n.arrival then
7       n.arrival ← pinArrival + delay
8       n.intercept ← pinArrival +α(i)
9       n.slope ← β(i)
10      n.bestMatch ← m(n,g)
11    end if
12  end for
```

In the *single match* method, the load due to the fanouts of $n$ is computed in the same way as in the fixed load method. The interconnect delay is computed using the steiner tree model described in Section 4.3. After having computed the arrival time of $n$, the result is stored at $n$ as a tuple $(I_n, S_n)$. Let $pi_k$ be the latest arriving input of match $m(n,g)$ of node $n$. $I_n$ is the sum of the arrival time at $pi_k$ and the intrinsic delay of match $m(n,g)$ from $pi_k$ to $n$ (The arrival time at $pi_k$ is

the *pinArrival* variable in the pseudo-code). $S_n$ is the slope of the delay model from $pi_k$ to $n$ for the current match $m(n,g)$. $I_n$ and $S_n$ are used to compute the arrival times at the input pins of the fanouts of $n$ when they are visited. As an example, node $n$ together with all its fanouts and fanins is shown in Figure 6.8. If the length of net $n$ is $l_n$, then the arrival times at pin 1 and 2 of node $n$ (denoted as $A(i_1)$ and $A(i_2)$) and at the output pin of $n$ (denoted as $A(n)$) are

$A(i_1)$ = Elmore delay at pin 1 of $n$ computed using $I_1, S_1$, stored at $i_1$, pin load seen at $i_1$, and net $i_1$

$A(i_2)$ = Elmore delay at pin 2 of $n$ computed using $I_2, S_2$, stored at $i_2$, pin load seen at $i_2$, and net $i_1$

$$A(n) = \max_{k\in\{1,2\}}\{A(i_k)+\alpha(k)+\beta(k)\times(l_n\times c+C_{o_1}+C_{o_2})\}$$

$$p = \arg\max_{k\in\{1,2\}}\{A(i_k)+\alpha(k)+\beta(k)\times(l_n\times c+C_{o_1}+C_{o_2})\}$$

$$I_n = A(i_p)+\alpha(p)$$

$$S_n = \beta(p)$$

where $c$ is the capacitance per unit length of the interconnect, and $C_{o_1}$ and $C_{o_2}$ are the pin capacitances of the fanout $o_1$ and $o_2$. The parameters $\alpha(1), \alpha(2), \beta(1)$, and $\beta(2)$ are the intrinsic delay and drive strength from pin 1 and 2 to $n$ of match $m(n,g)$ as described in Section 2.3.

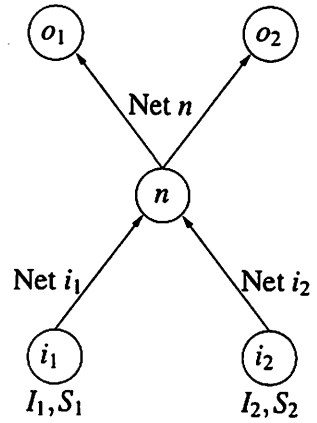



Figure 6.8: Single match method.

#### 6.3.3.3 Multiple Match Method

The multiple match cost computation algorithm is shown below.

**MultipleMatchCost**($\mathcal{N}$, $m(n,g), x_m, y_m$)
1 $pwl \leftarrow 0$
2 **foreach** fanin $f_i$ of $m(n,g)$ **do**
3 $pinArrival \leftarrow$ **PWLComputeDelay**($f_i.pwl$, $m(n,g), x_m, y_m$)
4 $pinArrival \leftarrow pinArrival +$ **ComputeWireResistanceDelay**($m(n,g), x_m, y_m$)
5 $pwl \leftarrow$ **PWLMax**($pwl$, $pinArrival + \alpha(i)$, $\beta(i)$)
6 **end for**
7 $n.pwl \leftarrow$ **PWLMin**($n.pwl$, $pwl$)

When considering a match $m(n,g)$ with gate $g$ at node $n$ in the *multiple match* method, the arrival time of $n$ is not computed. Rather, a piece-wise linear function $f$ is computed. This is best explained by an example. Consider again Figure 6.8. The delay model from both input pins of $g$ is characterized by $\alpha(1)$, $\beta(1)$, and $\alpha(2)$, $\beta(2)$ as shown in Figure 6.9(a) and (b). The maximum delay values for all load values, which are computed by the function **PWLMax**(), are shown in Figure 6.9(c). The result is a piece-wise linear function for all possible load values for match $m(n,g)$ [Tou90]. Each match at $n$ is therefore characterized by a piece-wise linear function. Instead of storing a single match at $n$ as in the case of fixed load and single match methods, a piece-wise linear function $h$ which is the minimum of all piece-wise linear functions (computed using the function **PWLMin**()) of all matches at $n$, is stored at $n$. Hence, a set of matches are stored at $n$. The best match is selected only when the load at the output of $n$ is known. At this point $A(i_1)$ and $A(i_2)$, the arrival times of the fanins of $n$ in the example above, are computed.

Since the delay model of a gate used in this thesis does not account for wire resistance when computing the delay, the delay computation has been divided into two separate functions in the pseudo-code, i.e. **PWLComputeDelay**(), and **ComputeWireResistanceDelay**(). The function **PWLComputeDelay**() computes the delay due to capacitive loading seen at the gate, and the function **ComputeWireResistanceDelay**() computes the interconnect delay due to resistance of the net of the gate as described in Section 2.3.

The backward tracing step of the multiple match method is different from the fixed load and the single match methods. Unlike the fixed load and the single match methods, the best match at a node $n$ is computed as a set of matches in the forward propagation step. In the backward tracing step, the best match is computed since the load seen at $n$ is known. The algorithm is shown below.

**MultMatchBuildNetwork**($\mathcal{N}_{\text{mapped}}$, $n$)
1 $m(n,g) \leftarrow$ **PWLComputeMatch**($n$, $n.pwl$)
2 $newNode \leftarrow$ createMappedNode($\mathcal{N}_{\text{mapped}}$, $m(n,g)$)
3 $n.mapped \leftarrow$ TRUE

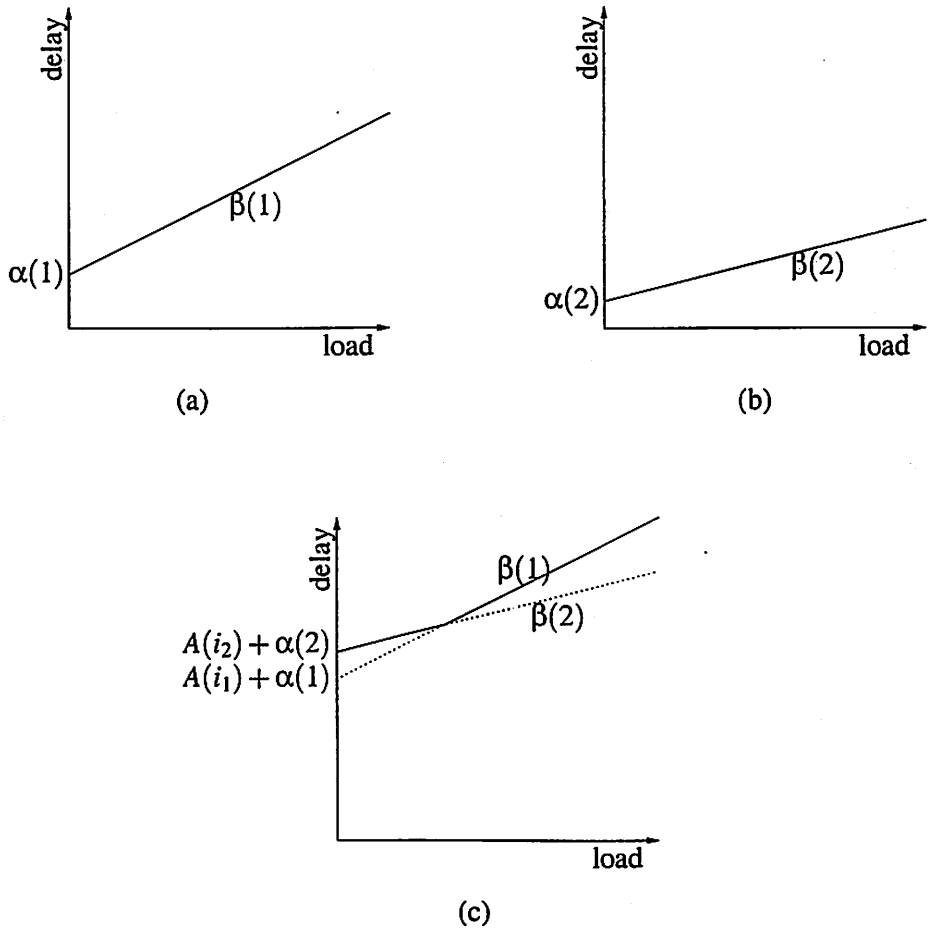



Figure 6.9: Delay models and maximum computation of piece-wise linear functions.

```
4   foreach fanin f of m(n,g) do
5     if f.mapped = FALSE then
6       TreeMapBuildNetwork(N_mapped, f)
7     end if
8   end for
```

When a new node is created, its location is estimated in the same way as in the area and wire-length minimization procedure, i.e. using the local placement algorithm as described in Section 6.1. The procedure of updating the positions of cells using the global placement algorithm is described in Section 6.3.2.

## 6.4 Experimental Results

The library used in our experiments is the MSU standard cell library [SSL+92]. The intrinsic delays, output resistances, and input capacitances of all cells in the library have been modified with values from SPICE characterizations of the $0.1\mu$m process technology described in Section 1.2.1. The detailed placement tool used is DOMINO [DJS91]. Routing is done using the WARP [Cad99] router from CADENCE. Although our $0.1\mu$ technology has as many as 8 layers of metal, only 4 layers are used for signal routing purposes. All other layers are assumed to be used for power and global signal routing. The post-routing interconnect delay is computed by modeling each wire segment as a $\pi$-segment.

The experimental setup is as follows. Twenty-five of the larger circuits in the 1991 MCNC benchmark set are selected for all experiments. For all experiments, the total delay is the delay of the critical path, and interconnect delay is the total delay including wire delay minus the total delay excluding wire delay.

### 6.4.1 Delay Correlation

The first set of experiments demonstrates the delay correlation before and after logic synthesis for both the traditional method and the proposed method. For the traditional method, circuits are synthesized by a logic synthesis tool with a wire-load model. The delay results are correlated with the delays of actual placed and routed results. For the proposed method, the circuits are synthesized using the integrated scheme, and the delay results are correlated with placed and routed circuits. For the traditional scheme, we first run SIS with the wire-load model shown in Section 1.2.3. For this wire-load model, the length of a 2-pin net, $l(2)$, was recomputed such that the

average error in delay is minimized for the circuits in this experiment. The same set of circuits are also synthesized using the proposed approach. Table 6.1 shows the results of this experiment. The columns under the heading "Wire" show the results of traditional synthesis (SIS with the wire-load model as described above). The columns under the heading "PILS" show the results of the proposed approach. All delay values are obtained after detail routing. The results illustrate the severity of the timing closure problem using traditional logic synthesis. As seen from this table, the SIS experiments with wire-load model over-estimates the actual delay in some cases (positive values) and under-estimates it in others (negative values). For the example *dalu*, the error in the estimated interconnect delay is as high as 57% and the error in the estimated total delay is 11%. The average error in the estimated interconnect delay is 29.0% and the average error in the estimated total delay is 4%. These averages are computed using the absolute values of the errors of all circuits. For the proposed approach, except for *dalu*, the estimated interconnect delay is small and the estimated total delay is within 1% of the actual delay. This table shows the effectiveness of the proposed approach, and demonstrates that timing closure is minimized the proposed method.

### 6.4.2 Area and Wire-Length Minimization

For area and wire-length minimization experiments, the *script.rugged* area optimization script of SIS is compared with the modified script.rugged where the proposed technology decomposition and technology mapping are used. Since tree mapping is used in the proposed scheme, the corresponding algorithm in SIS is used for comparison. The parameters $\alpha$ and $\beta$ used in the experiments are 0.1 and 10 respectively (See Section 6.2 and 6.3.1 for a description of these parameters).

Table 6.1: Estimated delay vs actual delay using wire-load model and our approach.

| | Interconnect Delay | | Total Delay | |
|---|---|---|---|---|
| Name | Wire | PILS | Wire | PILS |
| C1908 | −4% | 0% | 0% | 0% |
| C2670 | 8% | −2% | 1% | 0% |
| C3540 | 26% | 2% | 3% | 0% |
| C432 | 41% | −6% | 5% | −1% |
| C499 | −3% | −8% | 0% | −1% |
| C880 | −52% | −4% | −3% | 0% |
| b9 | −29% | 0% | −2% | 0% |
| dalu | 57% | −34% | 11% | −8% |
| k2 | 44% | 0% | 7% | 0% |
| Ave | 29% | 6% | 4% | 1% |

Table 6.2 compares the delay when area minimized technology mapping is performed. The traditional SIS technique is compared with the proposed approach (the column labeled "PILS"). In addition to the advantage of not having to estimate net lengths (and thereby solving the Timing Closure problem as shown in Table 6.1), the proposed scheme exhibits a significant reduction in interconnect delay as seen from Table 6.2. For the example *dalu*, this translates into a 23% reduction in total delay. Using our scheme, the average reduction in interconnect delay is 12.0%, and the average reduction in total delay is 7.7%.

Table 6.2: Area and wire-length minimization.

| | Interconnect Delay (ps) | | | Total Delay (ps) | | |
|---|---|---|---|---|---|---|
| Name | SIS | PILS | Change | SIS | PILS | Change |
| C1908 | 118 | 115 | −3% | 1364 | 1352 | −1% |
| C2670 | 140 | 95 | −32% | 1433 | 1245 | −13% |
| C3540 | 295 | 196 | −33% | 2195 | 2005 | −9% |
| C432 | 146 | 178 | 22% | 1549 | 1745 | 13% |
| C499 | 62 | 72 | 17% | 863 | 1000 | 16% |
| C6288 | 265 | 386 | 46% | 4926 | 4935 | 0% |
| C880 | 124 | 122 | −1% | 1737 | 1730 | −0% |
| apex6 | 245 | 134 | −45% | 1196 | 869 | −27% |
| cht | 42 | 31 | −24% | 444 | 318 | −28% |
| dalu | 417 | 266 | −36% | 2356 | 1816 | −23% |
| example | 122 | 114 | −6% | 904 | 841 | −7% |
| frg2 | 358 | 422 | 18% | 1645 | 2076 | 26% |
| i5 | 87 | 63 | −27% | 795 | 547 | −31% |
| i6 | 306 | 346 | 13% | 1622 | 1673 | 3% |
| i7 | 366 | 323 | −12% | 1700 | 1499 | −12% |
| i8 | 815 | 805 | −1% | 2884 | 2991 | 4% |
| i9 | 485 | 474 | −2% | 2522 | 2408 | −5% |
| pair | 546 | 330 | −40% | 2318 | 1745 | −25% |
| rot | 121 | 86 | −28% | 1147 | 1107 | −3% |
| term1 | 25 | 20 | −21% | 397 | 371 | −6% |
| ttt2 | 38 | 35 | −8% | 617 | 605 | −2% |
| vda | 130 | 137 | 5% | 884 | 826 | −7% |
| x1 | 68 | 43 | −37% | 525 | 426 | −19% |
| x3 | 226 | 128 | −43% | 1367 | 877 | −36% |
| x4 | 153 | 125 | −18% | 952 | 851 | −11% |
| Ave | – | – | −12.0% | – | – | −8.1% |

The improvement in interconnect delay of our method is accompanied by some penalty

in active area, as shown in Table 6.3. The average penalty in active area is 9.6%.

Table 6.3: Area of area and wire-length minimization (in $\mu^2$)

| Name | SIS | PILS | Change |
|---|---|---|---|
| C1908 | 5933 | 6036 | 2% |
| C2670 | 7874 | 9475 | 20% |
| C3540 | 14844 | 16410 | 11% |
| C432 | 2333 | 2356 | 1% |
| C499 | 5818 | 6019 | 3% |
| C6288 | 36858 | 37642 | 2% |
| C880 | 4758 | 5363 | 13% |
| apex6 | 8571 | 9360 | 9% |
| cht | 1797 | 1843 | 3% |
| dalu | 10783 | 12810 | 19% |
| example2 | 3923 | 4349 | 11% |
| frg2 | 8824 | 10598 | 20% |
| i5 | 2281 | 2701 | 18% |
| i6 | 6641 | 7171 | 8% |
| i7 | 8375 | 9556 | 14% |
| i8 | 11894 | 13720 | 15% |
| i9 | 7396 | 7569 | 2% |
| pair | 19020 | 21612 | 14% |
| rot | 8156 | 8617 | 6% |
| term1 | 2079 | 2074 | −0% |
| ttt2 | 2575 | 2851 | 11% |
| vda | 7033 | 7425 | 6% |
| x1 | 3738 | 4044 | 8% |
| x3 | 9210 | 10195 | 11% |
| x4 | 4677 | 5397 | 15% |
| Ave | – | – | 9.6% |

The parameter $\alpha$ is important in determining the quality of the results. The variation of interconnect delay, total delay and active area of the example *C3540* with respect to $\alpha$ is plotted in Figure 6.10. The $x$-axis represents $\alpha$. The left vertical axis is the delay in picoseconds. The right vertical axis is the area in $\mu m^2$. As expected, the active area increases as $\alpha$ increases. The delay decreases until $\alpha$ is about 0.1. For larger $\alpha$ values, the delay increases slowly since the increased circuit area results in an increase in net lengths. For $\alpha$ larger than about 0.9, there is a steeper rise in both area and delay.

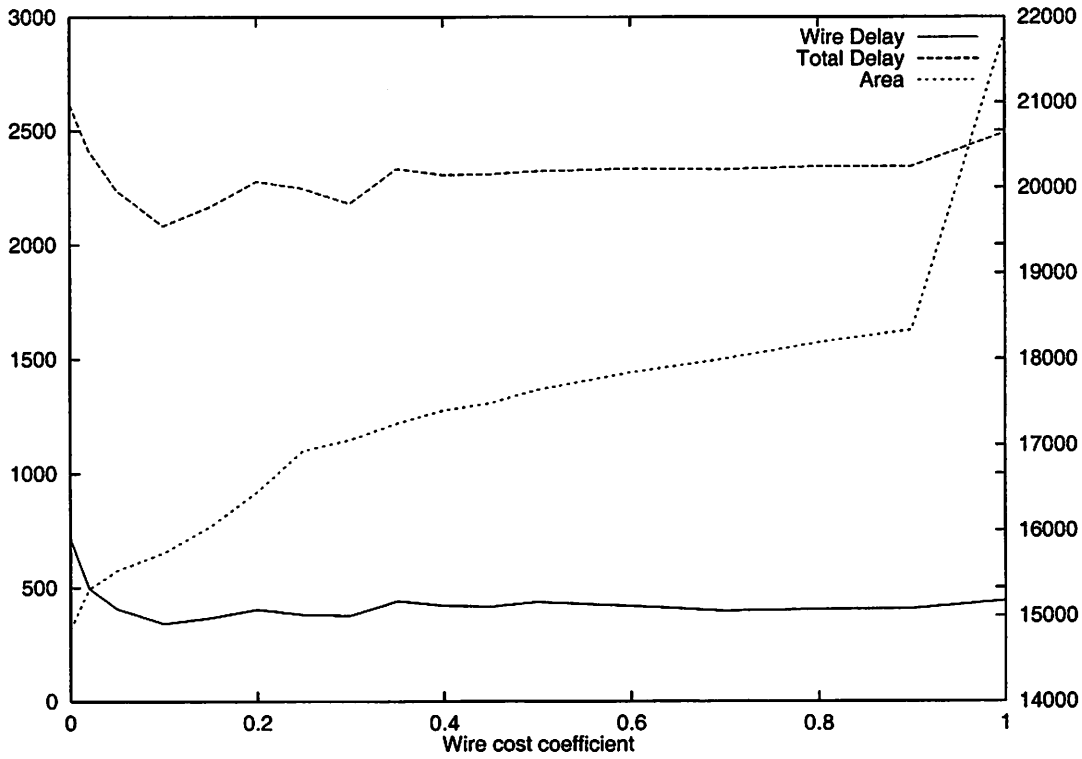



Figure 6.10: Delay and area variation with respect to $\alpha$, the wire cost coefficient.

### 6.4.3 Delay Optimization

For delay optimization, the three different optimization algorithms are compared against the delay minimization script *script.delay* in SIS. Modifications are made to script.delay such that the SIS algorithms being compared against utilized the same options as were utilized by the proposed algorithms.

Table 6.4 compares the delay when delay minimized technology mapping is performed using SIS and the fixed load method. The results for SIS are in columns labeled "SIS" and the results for fixed load method are in columns label "Fixed Load PILS". Although there is some improvement in almost all circuits, the amount of the interconnect and total delays are small. Fortunately, there is also an small reduction in area as seen from Table 6.5.

Table 6.6 shows the delay comparison between SIS and the single match method. The results of the single match method are in columns labeled "Single Match PILS". As expected, this algorithm outperforms the fixed load method since the arrival time computation is more accurate. On average, the interconnect delay is reduced by 3.2% and the total delay by 1.8%. The active area also reduces by an average of 0.7% as seen from Table 6.7.

Table 6.8 shows the delay comparison between multiple match algorithm in SIS (**map -n 1**) and the multiple match method in our approach. The results of the multiple match method are in columns labeled "Multiple Match PILS". As seen from this table, there is a significant reduction in interconnect delay (13.6%) and total delay (10.0%). Table 6.9 shows the area comparison of SIS and multiple match method. As seen from this table, there is a significant reduction in area (23.1%) in addition to the delay reduction seen from Table 6.8. So, in addition to minimizing Timing Closure problems, the proposed algorithm is able to reduce total circuit delay as well as total circuit area.

## 6.5 Conclusions and Future Work

In this chapter, we have presented an approach that addresses the timing closure problem in IC design. Our approach integrates the technology mapping step of logic synthesis with placement. We believe that success in integrating logic synthesis and placement is dependent on the ability to maintain a consistent placement during logic synthesis which closely approximates the final placement. We used incremental and global placement algorithms to achieve this goal.

We have introduced technology decomposition and technology mapping algorithms using this integrated flow. We attempt to minimize a weighted function of area and wire length while

Table 6.4: Fixed load delay minimization method.

| Name | Interconnect Delay (ps) | | | Total Delay (ps) | | |
|---|---|---|---|---|---|---|
| | SIS | Fixed Load PILS | Change | SIS | Fixed Load PILS | Change |
| C1908 | 197 | 196 | −0% | 1377 | 1386 | 1% |
| C2670 | 374 | 372 | −1% | 2193 | 2206 | 1% |
| C3540 | 522 | 503 | −4% | 2808 | 2752 | −2% |
| C432 | 142 | 140 | −2% | 1545 | 1559 | 1% |
| C499 | 84 | 83 | −1% | 874 | 873 | −0% |
| C6288 | 320 | 339 | 6% | 3451 | 3473 | 1% |
| C880 | 124 | 124 | −0% | 1193 | 1217 | 2% |
| apex6 | 576 | 582 | 1% | 1943 | 1943 | 0% |
| cht | 139 | 138 | −0% | 776 | 776 | −0% |
| dalu | 989 | 975 | −1% | 3296 | 3291 | −0% |
| example | 199 | 199 | 0% | 1091 | 1092 | 0% |
| frg2 | 546 | 517 | −5% | 2178 | 2079 | −5% |
| i5 | 64 | 67 | 3% | 617 | 620 | 0% |
| i6 | 750 | 689 | −8% | 3542 | 3465 | −2% |
| i7 | 1117 | 1109 | −1% | 4366 | 4359 | −0% |
| i8 | 1723 | 1743 | 1% | 5584 | 5551 | −1% |
| i9 | 601 | 580 | −3% | 2436 | 2442 | 0% |
| pair | 649 | 554 | −15% | 1950 | 1721 | −12% |
| rot | 151 | 154 | 3% | 1130 | 1138 | 1% |
| term1 | 47 | 47 | 1% | 506 | 505 | −0% |
| ttt2 | 51 | 37 | −26% | 445 | 431 | −3% |
| vda | 265 | 264 | −0% | 1204 | 1221 | 1% |
| x1 | 84 | 84 | −0% | 615 | 614 | −0% |
| x3 | 716 | 717 | 0% | 2109 | 2109 | 0% |
| x4 | 217 | 223 | 3% | 1243 | 1228 | −1% |
| Ave | – | – | −2.0% | – | – | −0.7% |

Table 6.5: Area of the fixed load delay minimization method. (in $\mu^2$)

| Name | SIS | Fixed Load PILS | Change |
|---|---|---|---|
| C1908 | 11318 | 11330 | 0% |
| C2670 | 14855 | 15085 | 2% |
| C3540 | 23679 | 23616 | −0% |
| C432 | 3105 | 3180 | 2% |
| C499 | 7718 | 8024 | 4% |
| C6288 | 55953 | 55872 | −0% |
| C880 | 7649 | 7690 | 1% |
| apex6 | 14907 | 14625 | −2% |
| cht | 2863 | 2863 | 0% |
| dalu | 19716 | 19647 | −0% |
| example2 | 5714 | 5731 | 0% |
| frg2 | 16295 | 15748 | −3% |
| i5 | 6244 | 6273 | 0% |
| i6 | 7989 | 7972 | −0% |
| i7 | 11002 | 10990 | −0% |
| i8 | 21594 | 21750 | 1% |
| i9 | 12269 | 11762 | −4% |
| pair | 30493 | 30246 | −1% |
| rot | 12355 | 12344 | −0% |
| term1 | 3514 | 3548 | 1% |
| ttt2 | 3203 | 3168 | −1% |
| vda | 15068 | 15057 | −0% |
| x1 | 5656 | 5656 | 0% |
| x3 | 14285 | 14308 | 0% |
| x4 | 7476 | 7275 | −3% |
| Ave | – | – | −0.2% |

Table 6.6: Single match delay minimization method.

| Name | Interconnect Delay (ps) | | | Total Delay (ps) | | |
|---|---|---|---|---|---|---|
| | SIS | Single Match PILS | Change | SIS | Single Match PILS | Change |
| C1908 | 197 | 211 | 7% | 1377 | 1441 | 5% |
| C2670 | 374 | 376 | 0% | 2193 | 2237 | 2% |
| C3540 | 522 | 458 | −12% | 2808 | 2815 | 0% |
| C432 | 142 | 138 | −3% | 1545 | 1421 | −8% |
| C499 | 84 | 72 | −14% | 874 | 849 | −3% |
| C6288 | 320 | 326 | 2% | 3451 | 3338 | −3% |
| C880 | 124 | 117 | −5% | 1193 | 1152 | −3% |
| apex6 | 576 | 561 | −3% | 1943 | 1911 | −2% |
| cht | 139 | 119 | −14% | 776 | 740 | −5% |
| dalu | 989 | 904 | −9% | 3296 | 3225 | −2% |
| example | 199 | 216 | 9% | 1091 | 1089 | −0% |
| frg2 | 560 | 545 | −3% | 2000 | 2251 | 13% |
| i5 | 64 | 68 | 6% | 617 | 595 | −4% |
| i6 | 750 | 645 | −14% | 3542 | 3344 | −6% |
| i7 | 1117 | 949 | −15% | 4366 | 4041 | −7% |
| i8 | 1723 | 1941 | 13% | 5584 | 5900 | 6% |
| i9 | 601 | 521 | −13% | 2436 | 2436 | 0% |
| pair | 649 | 539 | −17% | 1950 | 1798 | −8% |
| rot | 151 | 152 | 1% | 1130 | 1112 | −2% |
| term1 | 47 | 52 | 12% | 506 | 539 | 7% |
| ttt2 | 51 | 53 | 4% | 445 | 447 | 0% |
| vda | 237 | 271 | 14% | 1304 | 1187 | −9% |
| x1 | 84 | 83 | −1% | 615 | 582 | −5% |
| x3 | 716 | 682 | −5% | 2109 | 2056 | −2% |
| x4 | 217 | 208 | −4% | 1243 | 1202 | −3% |
| Ave | – | – | −3.2% | – | – | −1.8% |

Table 6.7: Area of the single match delay minimization method. (in $\mu^2$)

| Name | SIS | Single Match PILS | Change |
|---|---|---|---|
| C1908 | 11318 | 11854 | 5% |
| C2670 | 14855 | 14809 | −0% |
| C3540 | 23679 | 23570 | −0% |
| C432 | 3105 | 3191 | 3% |
| C499 | 7718 | 7551 | −2% |
| C6288 | 55953 | 55907 | −0% |
| C880 | 7649 | 7684 | 0% |
| apex6 | 14907 | 14861 | −0% |
| cht | 2863 | 2828 | −1% |
| dalu | 19716 | 19480 | −1% |
| example2 | 5714 | 5651 | −1% |
| frg2 | 16295 | 15621 | −4% |
| i5 | 6244 | 6244 | 0% |
| i6 | 7989 | 7932 | −1% |
| i7 | 11002 | 10414 | −5% |
| i8 | 21594 | 21951 | 2% |
| i9 | 12269 | 11958 | −3% |
| pair | 30493 | 30125 | −1% |
| rot | 12355 | 12090 | −2% |
| term1 | 3514 | 3393 | −3% |
| ttt2 | 3203 | 3151 | −2% |
| vda | 15068 | 15005 | −0% |
| x1 | 5656 | 5714 | 1% |
| x3 | 14285 | 14463 | 1% |
| x4 | 7476 | 7292 | −2% |
| Ave | – | – | −0.7% |

Table 6.8: Multiple match delay minimization method.

| Name | Interconnect Delay (ps) | | | Total Delay (ps) | | |
|---|---|---|---|---|---|---|
| | SIS | Multiple Match PILS | Change | SIS | Multiple Match PILS | Change |
| C1908 | 183 | 164 | −11% | 1419 | 1391 | −2% |
| C2670 | 258 | 230 | −11% | 1937 | 1811 | −6% |
| C3540 | 452 | 385 | −15% | 2603 | 2243 | −14% |
| C432 | 165 | 132 | −20% | 1607 | 1408 | −12% |
| C499 | 61 | 66 | 8% | 830 | 806 | −3% |
| C6288 | 280 | 299 | 7% | 3720 | 3559 | −4% |
| C880 | 108 | 90 | −16% | 1194 | 981 | −18% |
| apex6 | 636 | 491 | −23% | 2287 | 1883 | −18% |
| cht | 173 | 86 | −50% | 967 | 523 | −46% |
| dalu | 618 | 524 | −15% | 2589 | 2277 | −12% |
| example | 181 | 134 | −26% | 1092 | 878 | −20% |
| frg2 | 546 | 557 | 2% | 2178 | 2020 | −7% |
| i5 | 74 | 81 | 9% | 631 | 586 | −7% |
| i6 | 417 | 349 | −16% | 1620 | 1611 | −1% |
| i7 | 817 | 603 | −26% | 2855 | 2571 | −10% |
| i8 | 2017 | 1701 | −16% | 6653 | 6091 | −8% |
| i9 | 588 | 564 | −4% | 2635 | 2607 | −1% |
| pair | 884 | 465 | −47% | 2417 | 1630 | −33% |
| rot | 97 | 104 | 8% | 806 | 942 | 17% |
| term1 | 39 | 30 | −24% | 474 | 400 | −16% |
| ttt2 | 46 | 46 | 1% | 466 | 416 | −11% |
| vda | 265 | 248 | −7% | 1204 | 1279 | 6% |
| x1 | 63 | 56 | −11% | 532 | 495 | −7% |
| x3 | 766 | 519 | −32% | 2506 | 1880 | −25% |
| x4 | 320 | 281 | −12% | 1370 | 1322 | −4% |
| Ave | – | – | −13.3% | – | – | −10.2% |

Table 6.9: Area of multiple match delay minimization method. (in $\mu^2$)

| Name | SIS | Multiple Match PILS | Change |
|---|---|---|---|
| C1908 | 13916 | 11330 | −19% |
| C2670 | 18887 | 13317 | −29% |
| C3540 | 28155 | 22383 | −20% |
| C432 | 3894 | 3600 | −8% |
| C499 | 8531 | 6676 | −22% |
| C6288 | 70767 | 60710 | −14% |
| C880 | 8870 | 7240 | −18% |
| apex6 | 17240 | 12326 | −28% |
| cht | 3727 | 2367 | −36% |
| dalu | 25476 | 19728 | −23% |
| example2 | 8329 | 5478 | −34% |
| frg2 | 21905 | 14694 | −33% |
| i5 | 6682 | 5979 | −11% |
| i6 | 10639 | 7425 | −30% |
| i7 | 14400 | 9579 | −33% |
| i8 | 29094 | 21652 | −26% |
| i9 | 15592 | 10489 | −33% |
| pair | 35222 | 29635 | −16% |
| rot | 14498 | 12038 | −17% |
| term1 | 4205 | 3491 | −17% |
| ttt2 | 4015 | 3277 | −18% |
| vda | 20736 | 15546 | −25% |
| x1 | 6578 | 5334 | −19% |
| x3 | 15368 | 12678 | −18% |
| x4 | 10040 | 6970 | −31% |
| Ave | – | – | −23.1% |

interleaving incremental and global placement with logic operations. We also implemented a delay minimization algorithm.

The benefits of our approach are:

• The main result is the demonstration of a significant reduction in Timing Closure problems. Timing closure results in traditional logic synthesis underestimating interconnect delay by 29% on average. Our scheme reduces this error to 6%.

• Both incremental and global placement algorithms utilize the same core placement algorithm, and the same net model. This helps maintain a consistent placement during logic operations.

• Additionally, our scheme results in average reductions of about 12% in interconnect delay, and about 8.1% in total circuit delay for area and wire-length minimization. This is accompanied by an area penalty of 9.6%.

• For delay minimization algorithm, we achieve an average reduction of about 13.3% in interconnect delay, and about 10.2% in total circuit delay. In addition to this, we gain about 23.1% in area.

In the next chapter, we extend the current approach to include technology independent optimization as well.

# Chapter 7

# Technology Independent Optimization

In the previous chapter, we presented integrated algorithms for technology dependent optimization of logic synthesis and placement. In this chapter, we describe technology independent optimizations integrated with placement. As described in Section 2.2, there are a number of logic operations commonly used in the technology independent step, like kernel extraction, re-substitution, and simplification of Boolean nodes. Kernel extraction extracts common kernels from Boolean nodes and new nodes are created for the kernels. Re-substitution re-expresses a node in terms of other nodes in the Boolean network. Node simplification uses satisfiability and observability don't cares to simplify each Boolean node separately. Among these logic operations, kernel extraction changes the structure of the network extensively, allowing significant opportunities to improve the quality of the results. For this reason, we focus our attention on integrating kernel extraction and placement.

The cost function used in technology independent optimization is the literal count of the network. The kernel extraction algorithm iterates through the Boolean network. In each iteration, it traverses the network to find a kernel that reduces the literal count maximally and extracts it as a new node in the circuit. The Boolean network is then re-expressed using the new kernel. The iteration stops when the literal count of the network cannot be reduced anymore.

Since no interconnect information is present when using literal count as the cost function, two different kernel extraction algorithms are introduced in this chapter, depending on how the cost functions are defined: kernel extraction with *wire cost* and kernel extraction with *wire-planning*. In kernel extraction with wire cost, the cost of a kernel consists of two components: a literal count reduction component and a wire length reduction component. In the sequel, this will be referred to as *wired* kernel extraction. In the kernel extraction with wire-planning, heuristics are used to allow

duplication of kernels. The potential increase in literal count due to such duplication is computed in the cost function. In the sequel, this kernel extraction will be referred to as *wire-planned* kernel extraction.

## 7.1 Preliminaries

### 7.1.1 Value of a Kernel

The *value* of a kernel is the number of literals in the Boolean network that can be decreased if the kernel is extracted. The value $value(k)$ of a kernel $k$ can be computed easily. First, express the Boolean function at each of the fanout nodes in factored form. Let $n_k$ be the number of times kernel $k$ appears in the factored form of all its fanouts. Also, let $l_k$ be the number of literals in $k$. The value of $k$ is

$$value(k) = n_k(l_k - 1) - l_k$$

because for each appearance of $k$ in the fanout, $(l_k - 1)$ literals are reduced and $l_k$ literals are needed to represent kernel $k$ as a new Boolean node in the network.

### 7.1.2 Placement Interaction

As in Chapter 6, where we integrated technology mapping and placement, global placement is invoked several times during kernel extraction. Since the number of iterations is not known before the kernel extraction algorithm terminates, the number of global placement invocations cannot be determined in advance. Instead, a parameter $\gamma$ is introduced. The global placement is invoked after every $\gamma$ iterations. Intervening global placement invocations utilize the incremental mode.

As mentioned above, the kernel extraction algorithm is an iterative algorithm. In each iteration, a single kernel is selected from all generated kernels. The costs of all kernels need to be computed, which means that the location of all kernels need to be estimated. For this purpose, the local placement algorithm (described in Section 6.1) is used.

During technology independent optimization, the circuit structure is typically very different from the final circuit generated after technology dependent optimization. It is therefore not necessary to use interconnect delay as one of the criteria in determining the best kernel to extract. However, the relative positions of Boolean nodes along input/output paths during kernel extraction

will likely remain the same as the placement of the final circuit. For this reason, the cost function of interconnect length is preferred to interconnect delay in computing the weight of a kernel. As a result, semi-perimeter estimate is used to compute the length of a net (instead of the more computationally expensive Steiner tree estimate) as described in Section 4.3.

As in the technology decomposition algorithms and reasons explained in Chapter 6, each Boolean node is treated as a fixed-ratio cell that is scaled according to the number of nodes in the network. Updating the positions of all Boolean nodes is straightforward since the resulting Boolean network after kernel extraction is placed without adding or deleting any nodes.

## 7.2 Wired Kernel Extraction

In wired kernel extraction, a wire cost component of a kernel $k$ is computed and used along with the value of kernel $k$ to compute the cost function of $k$. The cost of $k$ is referred to as the weight of $k$, or $w(k)$. We seek the maximum of this value to reduce the literal count of the network.

The interconnect weight of a kernel $k$, $w_i(k)$ is simply the length of interconnect that is reduced by employing kernel $k$. Figure 7.1 shows an example of a part of a Boolean network before and after extracting a kernel $k$ before and after extraction. The interconnect length before the extraction of $k$ is the sum of the lengths of nets $i_1$ and $i_2$. The interconnect length after the extraction of $k$ is the sum of the lengths of nets $k$, $i_1$, and $i_2$ after extracting $k$. The interconnect weight of kernel $k$, $w_i(k)$, is the difference between the interconnect lengths before and after extraction[1].

The weight of a kernel $k$ is the weighted sum of the value of $k$ and the interconnect weight of $k$, i.e.

$$w(k) = value(k) + \lambda w_i(k)$$

## 7.3 Wire-planned Kernel Extraction

Wire-planned kernel extraction is a heuristic approach based on the results in the wire-planning approach in Chapter 3. The basic idea is to relax the monotonic path constraint and to apply legality locally.

[1] It is possible that after extracting a kernel $k$ from a node $n$, some fanins of $k$ are still the fanin of $n$. For example, in the example in Figure 7.1, $i_1$ can still be an immediate fanin of $o_1$ after extracting $k$.

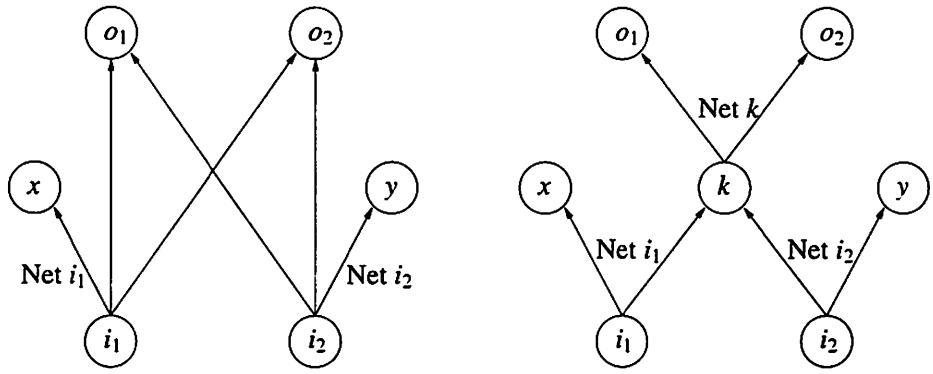



Figure 7.1: Before and after extracting kernel $k$ in *wired* kernel extraction.

### 7.3.1 Weight of a Kernel

For a node $k$ with placement constraint $\{i_1, i_2, \ldots, i_m\}_{o_1,o_2,\ldots,o_l}$ (i.e. the transitive fanins of $k$ include primary inputs $i_1, i_2, \ldots, i_m$, and the transitive fanouts of $k$ include primary outputs $o_1, o_2, \ldots, o_l$), the *input box* of $k$ is defined as the smallest rectangle that encloses the primary inputs of $k$ and the *output box* of $k$ is the smallest rectangle that encloses the primary outputs of $k$. With these definitions, the conditions under which a node is legal can be restated by the following lemma.

**Lemma 7.1** *For a legal node $k$ with placement constraint $\{i_1, i_2, \ldots, i_m\}_{o_1,o_2,\ldots,o_n}, m > 1, n > 1$, its legal region intersects with its output box only at its closest point, i.e. the point closest to any output. Similarly, its legal region intersects with its input box only at its furthest point, i.e. the point furthest from any output.*

**Proof**: By the definition of *region* and the *intersection rule* in Section 3.4, the lemma follows. ■

The *primary output angle* of a node $k$ is defined to be the angle subtended by $k$ and all its primary outputs. Figure 7.2(a) illustrates this definition. Lemma 7.1 says that the primary output angle of a legal node $k$ is at most 90°, which is when $k$ is placed at the closest point of its legal region. Similar to the definition of primary output angle, the *fanout angle* of a node $k$ is defined to be the angle subtended by node $k$ and all its immediate fanouts, denoted by $\alpha_k$. Figure 7.2(b) illustrates this definition.

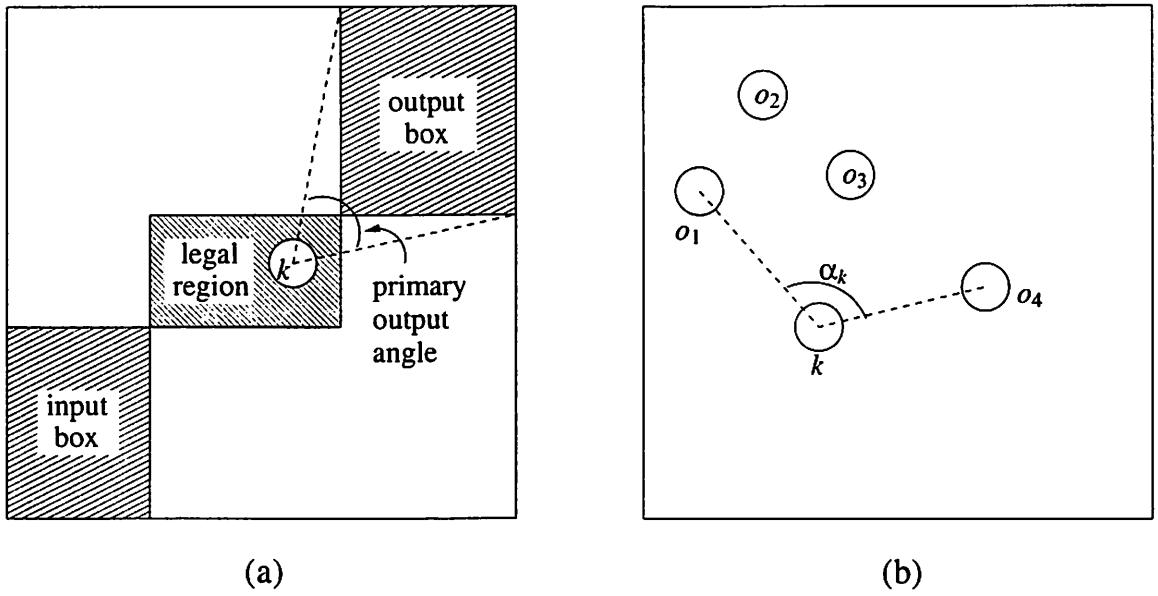



Figure 7.2: Primary output and fanout angles in wire-planned kernel extraction.

The basic idea behind the wire-planned kernel extraction is to restrict the fanout angle $\alpha_k$ of a kernel $k$ to be at most 90°. If $\alpha_k$ is larger than 90°, then the weight of the kernel is penalized by the number of literals needed to duplicate the kernel in order to maintain a maximum fanout angle of 90° for each kernel and its duplicates. Hence, the weight $w(k)$ of a kernel $k$ is

$$w(k) = value(k) - \left\lfloor \frac{\alpha_k}{90°} \right\rfloor \times l_k$$

where $l_k$ is the number of literals of $k$.

Depending upon which two fanouts of $k$ are chosen as the end points when computing the fanout angle of $k$, its value can vary considerably. For minimum kernel duplication, two consecutive fanouts in a circular traversal around $k$ that form the largest angle with $k$ are chosen as the end points. For example, nodes $o_1$ and $o_4$ are the end points of the example shown in Figure 7.2(b) because the angle subtended by $o_1$, $k$, and $o_4$, which is $180° - \alpha_k$, is the largest among any consecutive nodes in a circular traversal.

### 7.3.2 Wire-planned Duplication

In the wire-planned kernel extraction described above, the weights of kernels are used as the objective function of the algorithm. However kernels are not duplicated during or after the

algorithm. Boolean nodes can be very large after kernel extraction. Duplicating nodes at this point can increase the literal count considerably. The duplication process is therefore proposed to be a separate operation, which is called a *wire-planned duplication* logic operation.

The wire-planned duplication algorithm traverses the Boolean network and duplicates nodes whose fanout angles are larger than 90°. Eventually, the fanout angles of the nodes and their duplicates are smaller than or equal to 90°. The algorithm is guaranteed to terminate because in the worst case each node only has a single fanout.

Let $l$ be the number of fanouts of $k$ and let $o_1$ and $o_l$ be the two end points found when computing the fanout angle $\alpha_k$ of $k$. If $\alpha_k > 90°$, then a circular order of the fanouts is computed, denoted as $FO^s(k) = (o_1, o_2, \ldots, o_l)$. The duplication algorithm partitions the set of all fanouts into $\lceil \frac{\alpha_k}{90°} \rceil$ sets. The partitions are created by traversing the fanouts in the order of the sorted list $FO^s(k)$. A partition starts from $o_1$, and nodes are added until a fanout is reached that would result in a fanout angle larger than 90°. A new partition is then started and the procedure is repeated.

The interaction with placement is the same as that of wired and wire-planned kernel extraction. When a duplicate node is created, its position is obtained using the local placement algorithm. The global placement is invoked after every $\psi$ nodes have been created during duplication.

## 7.4 Experimental Results

Both wired and wire-planned kernel extraction algorithms have been implemented as options to the *fast_extract* command of SIS. In all experiments, $\gamma$ is 20, i.e. the global placement is called after every 20 kernels have been extracted. For the wired kernel extraction algorithm, $\lambda$ is chosen to be $5\% \times \frac{W+H}{2}$, where $W$ and $H$ are the width and height of the core (or placement area) respectively. This choice of $\lambda$ means that if a kernel $k$ reduces the length of the interconnect by $\lambda$, then its value is increased by 1. The parameter of $\psi$ was chosen to be 5, i.e. the global placement is called after every 5 nodes are created during the wire-planned duplication algorithm. The same benchmark circuits are used in this chapter as were used in the previous.

### 7.4.1 Area and Wire-Length Minimization

In the first experiment, both wired and wire-planned kernel extraction algorithms are invoked within an area minimization script. The details of the SIS experiment are the same as in the previous chapter. The only difference between SIS minimization script and the new kernel extrac-

tion scripts is in the call to *fast_extract*, where appropriate options have been used to invoke the new algorithms. In the wire-planned kernel extraction runs, the duplication step follows the technology decomposition step. This is done to minimize area penalty, since after technology decomposition, all nodes are small, i.e. 2-input NAND/NOR gates and inverters. Table 7.1 shows the interconnect delay comparison between the results from the previous chapter (using only technology dependent optimizations) and the results using kernel extraction algorithms proposed in this chapter along with technology dependent optimizations. Columns labeled "SIS" and "Area" are results from the previous chapter (i.e. SIS results, and results of technology dependent optimization for area and wire-length minimization). Columns labeled "Wired Fx + Area" are the results using wired kernel extraction followed by technology dependent optimization for area and wire-length minimization. Columns labeled "Wp Fx + Area" are the results using wire-planned kernel extraction followed by technology dependent optimization for area and wire-length minimization. As seen from this table, the wire-planned kernel extraction achieves significant reduction in interconnect delay (30.8% on average) while wired kernel extraction is not effective (the interconnect delay actually increases as compared to the results obtained using only technology dependent optimization). Similar improvements are obtained in the total delay as shown in Table 7.2. As seen from this table, the wire-planned kernel approach followed by the technology dependent approach from the previous chapter results in a 20.3% reduction in circuit delay on average. The wired kernel extraction approach results in an increase in the total delay (compared to the results obtained by using only the technology dependent approach). The penalty in area is shown in Table 7.3. The wired kernel extraction algorithm has an area penalty of 12.0% on average, or 2.4% higher than simply using technology dependent optimization. The wire-planned approach has a higher area penalty in area because it duplicates cells (18.5% on average).

### 7.4.2 Delay Optimization

In the second experiment, both the wired and wire-planned kernel extraction algorithms are invoked within a delay optimization script. The scripts for SIS and the proposed schemes are the same as in the previous chapter. The only difference is that in the proposed schemes, the kernel extraction step utilizes either wired or wire-planned kernel extraction. As in the area minimization experiments, the duplication step follows the technology decomposition step in the wire-planned kernel extraction runs. The technology dependent optimization used here is the *multiple match* method since it gives the best results. Table 7.4 shows the interconnect delay comparison between

Table 7.1: Comparing interconnect delay for different kernel extraction algorithms in area minimization (delays in ps).

| | | Area | | Wired Fx + Area | | Wp Fx + Area | |
|---|---|---|---|---|---|---|---|
| Name | SIS | Val | Percent | Val | Percent | Val | Percent |
| C1908 | 118 | 115 | −3% | 133 | 12% | 84 | −29% |
| C2670 | 140 | 95 | −32% | 123 | −12% | 109 | −22% |
| C3540 | 295 | 196 | −33% | 166 | −44% | 170 | −42% |
| C432 | 146 | 178 | 22% | 178 | 22% | 68 | −53% |
| C499 | 62 | 72 | 17% | 74 | 19% | 79 | 28% |
| C6288 | 265 | 386 | 46% | 386 | 46% | 259 | −2% |
| C880 | 124 | 122 | −1% | 117 | −6% | 119 | −4% |
| apex6 | 245 | 134 | −45% | 133 | −46% | 152 | −38% |
| cht | 42 | 31 | −24% | 32 | −23% | 32 | −24% |
| dalu | 417 | 266 | −36% | 288 | −31% | 187 | −55% |
| example2 | 122 | 114 | −6% | 110 | −9% | 70 | −42% |
| frg2 | 358 | 422 | 18% | 407 | 14% | 258 | −28% |
| i5 | 87 | 63 | −27% | 64 | −27% | 55 | −37% |
| i6 | 306 | 346 | 13% | 346 | 13% | 135 | −56% |
| i7 | 366 | 323 | −12% | 329 | −10% | 164 | −55% |
| i8 | 815 | 805 | −1% | 813 | −0% | 441 | −46% |
| i9 | 485 | 474 | −2% | 469 | −3% | 246 | −49% |
| pair | 546 | 330 | −40% | 385 | −29% | 357 | −35% |
| rot | 121 | 86 | −28% | 91 | −24% | 83 | −31% |
| term1 | 25 | 20 | −21% | 27 | 9% | 27 | 9% |
| ttt2 | 38 | 35 | −8% | 34 | −10% | 32 | −15% |
| vda | 130 | 137 | 5% | 151 | 16% | 134 | 3% |
| x1 | 68 | 43 | −37% | 38 | −45% | 24 | −65% |
| x3 | 226 | 128 | −43% | 125 | −45% | 107 | −52% |
| x4 | 153 | 125 | −18% | 107 | −30% | 109 | −29% |
| Ave | — | — | −12.0% | — | −9.7% | — | −30.8% |

Table 7.2: Comparing total delay for different kernel extraction algorithms in area minimization (delays in ps).

| | | Area | | Wired Fx + Area | | Wp Fx + Area | |
|---|---|---|---|---|---|---|---|
| Name | SIS | Val | Percent | Val | Percent | Val | Percent |
| C1908 | 1364 | 1352 | −1% | 1404 | 3% | 1283 | −6% |
| C2670 | 1433 | 1245 | −13% | 1410 | −2% | 1297 | −9% |
| C3540 | 2195 | 2005 | −9% | 1695 | −23% | 1769 | −19% |
| C432 | 1549 | 1745 | 13% | 1745 | 13% | 1148 | −26% |
| C499 | 863 | 1000 | 16% | 998 | 16% | 1047 | 21% |
| C6288 | 4926 | 4935 | 0% | 4935 | 0% | 4331 | −12% |
| C880 | 1737 | 1730 | −0% | 1505 | −13% | 1719 | −1% |
| apex6 | 1196 | 869 | −27% | 869 | −27% | 924 | −23% |
| cht | 444 | 318 | −28% | 319 | −28% | 318 | −28% |
| dalu | 2356 | 1816 | −23% | 1936 | −18% | 1406 | −40% |
| example2 | 904 | 841 | −7% | 878 | −3% | 615 | −32% |
| frg2 | 1645 | 2076 | 26% | 1990 | 21% | 1483 | −10% |
| i5 | 795 | 547 | −31% | 541 | −32% | 636 | −20% |
| i6 | 1622 | 1673 | 3% | 1673 | 3% | 848 | −48% |
| i7 | 1700 | 1499 | −12% | 1504 | −12% | 753 | −56% |
| i8 | 2884 | 2991 | 4% | 3098 | 7% | 1929 | −33% |
| i9 | 2522 | 2408 | −5% | 2406 | −5% | 1547 | −39% |
| pair | 2318 | 1745 | −25% | 1858 | −20% | 1785 | −23% |
| rot | 1147 | 1107 | −3% | 1158 | 1% | 1154 | 1% |
| term1 | 397 | 371 | −6% | 380 | −4% | 410 | 3% |
| ttt2 | 617 | 605 | −2% | 605 | −2% | 598 | −3% |
| vda | 884 | 826 | −7% | 909 | 3% | 847 | −4% |
| x1 | 525 | 426 | −19% | 419 | −20% | 353 | −33% |
| x3 | 1367 | 877 | −36% | 855 | −37% | 708 | −48% |
| x4 | 952 | 851 | −11% | 890 | −7% | 778 | −18% |
| Ave | — | — | −8.1% | — | −7.4% | — | −20.3% |

Table 7.3: Comparing area for different kernel extraction algorithms in area minimization (in $\mu^2$).

| Name | SIS | Area | | Wired Fx + Area | | Wp Fx + Area | |
|---|---|---|---|---|---|---|---|
| | | Val | Percent | Val | Percent | Val | Percent |
| C1908 | 5933 | 6036 | 2% | 6209 | 5% | 6612 | 11% |
| C2670 | 7874 | 9475 | 20% | 9729 | 24% | 9469 | 20% |
| C3540 | 14844 | 16410 | 11% | 15114 | 2% | 17349 | 17% |
| C432 | 2333 | 2356 | 1% | 2356 | 1% | 2650 | 14% |
| C499 | 5818 | 6019 | 3% | 5979 | 3% | 6215 | 7% |
| C6288 | 36858 | 37642 | 2% | 37647 | 2% | 42584 | 16% |
| C880 | 4758 | 5363 | 13% | 5230 | 10% | 5374 | 13% |
| apex6 | 8571 | 9360 | 9% | 9510 | 11% | 10547 | 23% |
| cht | 1797 | 1843 | 3% | 1843 | 3% | 1976 | 10% |
| dalu | 10783 | 12810 | 19% | 13087 | 21% | 13496 | 25% |
| example2 | 3923 | 4349 | 11% | 4452 | 14% | 4821 | 23% |
| frg2 | 8824 | 10598 | 20% | 10714 | 21% | 11526 | 31% |
| i5 | 2281 | 2701 | 18% | 2782 | 22% | 3099 | 36% |
| i6 | 6641 | 7171 | 8% | 7171 | 8% | 7327 | 10% |
| i7 | 8375 | 9556 | 14% | 9590 | 15% | 9740 | 16% |
| i8 | 11894 | 13720 | 15% | 13997 | 18% | 14751 | 24% |
| i9 | 7396 | 7569 | 2% | 7580 | 2% | 8335 | 13% |
| pair | 19020 | 21612 | 14% | 22372 | 18% | 23397 | 23% |
| rot | 8156 | 8617 | 6% | 8974 | 10% | 9210 | 13% |
| term1 | 2079 | 2074 | −0% | 2333 | 12% | 2172 | 4% |
| ttt2 | 2575 | 2851 | 11% | 2713 | 5% | 3076 | 19% |
| vda | 7033 | 7425 | 6% | 9487 | 35% | 9026 | 28% |
| x1 | 3738 | 4044 | 8% | 4136 | 11% | 4337 | 16% |
| x3 | 9210 | 10195 | 11% | 9971 | 8% | 11146 | 21% |
| x4 | 4677 | 5397 | 15% | 5622 | 20% | 6036 | 29% |
| Ave | — | — | 9.6% | — | 12.0% | — | 18.5% |

the various approaches. Columns labeled "SIS" and "Mult. Match" correspond to the results of running SIS and the multiple match method respectively (as seen in the previous chapter). Columns labeled "Wired Fx + Mult. Match" show the results of running wired kernel extraction and the multiple match method. The results of running wire-planned kernel extraction and the multiple match method are shown in columns labeled "Wp Fx + Mult. Match". As seen from this table, the proposed kernel extraction algorithms achieve an improvement in interconnect delay. In the wire-planned kernel extraction approach, an average of 23.8% reduction is achieved. The results for total delay are shown in Table 7.5, where the wired and wire-planned kernel extraction techniques achieve average reductions of 12.3% and 14.8% (as opposed to a delay reduction of 10.2% for the multiple match method) respectively. Table 7.6 shows the area needed by all approaches. In order to achieve the additional delay reduction, the area utilization of the proposed kernel extraction approaches increases when compared with the multiple match method. However, the area utilization is still 22.3% and 17.9% smaller when compared with SIS.

Table 7.4: Comparing interconnect delay for different kernel extraction algorithms in delay minimization (delays in ps).

| | | Mult. Match | | Wired Fx + Mult. Match | | Wp Fx + Mult. Match | |
|---|---|---|---|---|---|---|---|
| Name | SIS | Val | Percent | Val | Percent | Val | Percent |
| C1908 | 183 | 164 | −11% | 155 | −16% | 185 | 1% |
| C2670 | 258 | 230 | −11% | 214 | −17% | 209 | −19% |
| C3540 | 452 | 385 | −15% | 349 | −23% | 384 | −15% |
| C432 | 165 | 132 | −20% | 136 | −17% | 79 | −52% |
| C499 | 61 | 66 | 8% | 63 | 4% | 71 | 16% |
| C6288 | 280 | 299 | 7% | 308 | 10% | 292 | 4% |
| C880 | 108 | 90 | −16% | 88 | −18% | 55 | −49% |
| apex6 | 636 | 491 | −23% | 486 | −24% | 447 | −30% |
| cht | 173 | 86 | −50% | 86 | −50% | 83 | −52% |
| dalu | 618 | 524 | −15% | 484 | −22% | 546 | −12% |
| example2 | 181 | 134 | −26% | 113 | −38% | 121 | −33% |
| frg2 | 560 | 545 | −3% | 414 | −26% | 292 | −48% |
| i5 | 74 | 81 | 9% | 81 | 9% | 81 | 8% |
| i6 | 417 | 349 | −16% | 350 | −16% | 315 | −24% |
| i7 | 817 | 603 | −26% | 605 | −26% | 591 | −28% |
| i8 | 2017 | 1701 | −16% | 1278 | −37% | 1378 | −32% |
| i9 | 588 | 564 | −4% | 435 | −26% | 248 | −58% |
| pair | 884 | 465 | −47% | 457 | −48% | 703 | −20% |
| rot | 97 | 104 | 8% | 95 | −2% | 95 | −2% |
| term1 | 39 | 30 | −24% | 30 | −23% | 26 | −34% |
| ttt2 | 46 | 46 | 1% | 44 | −4% | 30 | −33% |
| vda | 237 | 271 | 14% | 247 | 4% | 210 | −12% |
| x1 | 63 | 56 | −11% | 40 | −36% | 63 | 1% |
| x3 | 766 | 519 | −32% | 519 | −32% | 514 | −33% |
| x4 | 320 | 281 | −12% | 287 | −10% | 189 | −41% |
| Ave | — | — | −13.3% | — | −19.3% | — | −23.8% |

Table 7.5: Comparing total delay for different kernel extraction algorithms in delay minimization (delays in ps).

| | | Mult. Match | | Wired Fx + Mult. Match | | Wp Fx + Mult. Match | |
|---|---|---|---|---|---|---|---|
| Name | SIS | Val | Percent | Val | Percent | Val | Percent |
| C1908 | 1419 | 1391 | −2% | 1381 | −3% | 1464 | 3% |
| C2670 | 1937 | 1811 | −6% | 1763 | −9% | 1796 | −7% |
| C3540 | 2603 | 2243 | −14% | 2360 | −9% | 2263 | −13% |
| C432 | 1607 | 1408 | −12% | 1423 | −11% | 1190 | −26% |
| C499 | 830 | 806 | −3% | 856 | 3% | 973 | 17% |
| C6288 | 3720 | 3559 | −4% | 3516 | −5% | 3539 | −5% |
| C880 | 1194 | 981 | −18% | 869 | −27% | 902 | −24% |
| apex6 | 2287 | 1883 | −18% | 1865 | −18% | 1677 | −27% |
| cht | 967 | 523 | −46% | 523 | −46% | 527 | −45% |
| dalu | 2589 | 2277 | −12% | 2204 | −15% | 2260 | −13% |
| example2 | 1092 | 878 | −20% | 800 | −27% | 829 | −24% |
| frg2 | 2000 | 2251 | 13% | 2224 | 11% | 1641 | −18% |
| i5 | 631 | 586 | −7% | 587 | −7% | 568 | −10% |
| i6 | 1620 | 1611 | −1% | 1612 | −1% | 1488 | −8% |
| i7 | 2855 | 2571 | −10% | 2573 | −10% | 2475 | −13% |
| i8 | 6653 | 6091 | −8% | 4421 | −34% | 5049 | −24% |
| i9 | 2635 | 2607 | −1% | 2261 | −14% | 1525 | −42% |
| pair | 2417 | 1630 | −33% | 1672 | −31% | 2337 | −3% |
| rot | 806 | 942 | 17% | 898 | 11% | 881 | 9% |
| term1 | 474 | 400 | −16% | 374 | −21% | 429 | −10% |
| ttt2 | 466 | 416 | −11% | 417 | −11% | 370 | −21% |
| vda | 1304 | 1187 | −9% | 1440 | 10% | 1192 | −9% |
| x1 | 532 | 495 | −7% | 440 | −17% | 565 | 6% |
| x3 | 2506 | 1880 | −25% | 1832 | −27% | 1802 | −28% |
| x4 | 1370 | 1322 | −4% | 1366 | −0% | 891 | −35% |
| Ave | — | — | −10.2% | — | −12.3% | — | −14.8% |

Table 7.6: Comparing area for different kernel extraction algorithms in delay minimization (in $\mu^2$).

| Name | SIS | Mult. Match | | Wired Fx + Mult. Match | | Wp Fx + Mult. Match | |
|---|---|---|---|---|---|---|---|
| | | Val | Percent | Val | Percent | Val | Percent |
| C1908 | 13916 | 11330 | −19% | 11295 | −19% | 12568 | −10% |
| C2670 | 18887 | 13317 | −29% | 13784 | −27% | 14907 | −21% |
| C3540 | 28155 | 22383 | −20% | 22873 | −19% | 24157 | −14% |
| C432 | 3894 | 3600 | −8% | 3588 | −8% | 4038 | 4% |
| C499 | 8531 | 6676 | −22% | 6699 | −21% | 7540 | −12% |
| C6288 | 70767 | 60710 | −14% | 60273 | −15% | 64524 | −9% |
| C880 | 8870 | 7240 | −18% | 7517 | −15% | 7390 | −17% |
| apex6 | 17240 | 12326 | −28% | 12436 | −28% | 12943 | −25% |
| cht | 3727 | 2367 | −36% | 2367 | −36% | 2396 | −36% |
| dalu | 25476 | 19728 | −23% | 20177 | −21% | 21001 | −18% |
| example2 | 8329 | 5478 | −34% | 5875 | −29% | 6077 | −27% |
| frg2 | 21905 | 14694 | −33% | 14734 | −33% | 15782 | −28% |
| i5 | 6682 | 5979 | −11% | 6163 | −8% | 6215 | −7% |
| i6 | 10639 | 7425 | −30% | 7442 | −30% | 7148 | −33% |
| i7 | 14400 | 9579 | −33% | 9579 | −33% | 9746 | −32% |
| i8 | 29094 | 21652 | −26% | 21335 | −27% | 23484 | −19% |
| i9 | 15592 | 10489 | −33% | 10253 | −34% | 10644 | −32% |
| pair | 35222 | 29635 | −16% | 30200 | −14% | 31784 | −10% |
| rot | 14498 | 12038 | −17% | 12217 | −16% | 12874 | −11% |
| term1 | 4205 | 3491 | −17% | 3635 | −14% | 3623 | −14% |
| ttt2 | 4015 | 3277 | −18% | 3364 | −16% | 3444 | −14% |
| vda | 20736 | 15546 | −25% | 16088 | −22% | 18104 | −13% |
| x1 | 6578 | 5334 | −19% | 5270 | −20% | 5674 | −14% |
| x3 | 15368 | 12678 | −18% | 12678 | −18% | 13507 | −12% |
| x4 | 10040 | 6970 | −31% | 6687 | −33% | 7476 | −26% |
| Ave | — | — | −23.1% | — | −22.3% | — | −17.9% |

# Chapter 8

# Conclusions and Future Research

In this chapter we summarize the contributions of this thesis and point out some future directions in which this work can be extended.

## 8.1 Conclusions

The focus of this thesis has been the effects of interconnect on the design of integrated circuits, particularly, we studied two such effects – the effects of increasing interconnect delay along with the increasing size of circuits being synthesized by logic synthesis tools, and the timing closure problem (which is the large number of iterations needed to perform logic and layout synthesis before the results converge satisfactorily). We showed (analytically and experimentally) how the delay due to interconnect becomes increasingly important as the minimum feature sizes shrink. We also showed how the inaccuracy in estimating interconnect length, results in inaccuracies in estimating interconnect delay, thereby causes the timing closure problem. Specifically, we showed that the widely used wire-load model causes timing closure problems.

In Chapter 3 we proposed a novel logic synthesis (wire-planning) in which all the delay is assumed to be in the interconnect. We showed with an example why circuits synthesized using conventional logic synthesis tools can have long wires when placed. By simply examining the primary inputs and primary outputs of a Boolean node, we were able to determine whether there would be a diversion in some path through the node *regardless of placement.* A node which cannot be placed without diversions is called an illegal node. Since only the primary inputs and primary outputs of a node are examined to check for the legality of the node, we presented an efficient legality checking algorithm. The legality notion of a node was extended to a Boolean network.

A monotonic point placement is guaranteed to exist if every node in the network is individually legal. Moreover, given an arbitrary Boolean network, we showed that it can always be legalized. Logic operations that maintain legality while restructuring the Boolean network were introduced. Although the area penalty is large, the wire-planning approach provides a theoretical understanding of the interaction between logic synthesis and placement.

Observing that the cause of the timing closure problem is the difficulty in estimating interconnect length, we proposed an approach in Chapter 4 that integrates logic synthesis and global placement. With this approach, the interconnect length can be estimated more accurately since logic synthesis and placement are integrated, as confirmed by our results. The strength of our approach is the ability to maintain a placement of cells which is relatively similar to the placement of the final circuit. This is achieved by carefully using global and local placement algorithms along with synthesis operations. In addition, the same net model is used in our local placement algorithm (where we place nodes with respect to its neighbors only) and the global placement algorithm (which is a quadratic placement algorithm).

In the technology dependent optimization phase of the integrated algorithm, we showed a novel technology decomposition algorithm followed by four different technology mapping algorithms in Chapter 6. The heuristic technology decomposition algorithm is based on the understanding of what constitutes a good circuit for placement from the wire-planning approach. Our results showed a significant reduction of interconnect delay of the circuits which translates into a 10% average reduction in total delay for the benchmark circuits. We attribute this reduction to the ability of our scheme to minimally perturb the placement of unmapped and mapped nodes during the executions of the algorithms.

In the technology independent optimization phase of the proposed integrated approach, we introduce in Chapter 7 two different kernel extraction algorithms. The first one directly takes into account the interconnect length when searching for the best kernel to extract. For delay optimization, this results in an average improvement of 6% in the interconnect delay and 2% in the total delay (over the results of simply employing the technology dependent algorithms). The second kernel extraction algorithm is a heuristic based on the wire-planning approach. Here, kernel extraction is followed by a step that duplicates nodes such that the circuit is easier to place, resulting in a smaller interconnect delay. In addition to the delay reduction of the technology dependent phase, this kernel extraction and duplication scheme results in an average reduction of 10.5% in interconnect delay and 4.6% in total delay.

In summary, we have shown a theoretical approach that characterizes when a circuit has

long interconnect and we have proposed a practical solution that integrates logic synthesis and global placement. The main contribution of this integrated approach is that it is able to maintain the minimal perturbation of the placement of Boolean nodes. This results in a significant reduction in interconnect delay and total delay of the circuit.

## 8.2 Future Work

As mentioned above, the main contribution of this work is the ability to minimally perturb the positions of Boolean nodes in the integrated algorithm. In addition to integrating local and global placement, it is important to assign an effective objective function based on the optimization being performed. It is because of such differences in objective functions that the multiple-match method performs better that the fixed-load and single-match methods, and the wire-planned kernel extraction performs better than the wired kernel extraction. Besides the operations considered in this thesis, integrating other operations like graph mapping, and fanout optimization will be an interesting extension to this research.

The results of the experiments in this thesis are based on the MSU standard cell library [SSL+92]. This library is small, and only one size of each type of gate is available. An experiment with larger libraries will demonstrate the effectiveness of the proposed approach on richer libraries. It is likely that the results of the proposed approach will show even more improvement, since more options are available.

The wire-planning approach provides an understanding of how interconnect length is affected by logic synthesis. Besides the wire-planning based heuristics used in this thesis, other wire-planning heuristics which improve the integrated approach can be devised. An example of this is the selective application of wire-planned kernel extraction only on longer paths.

With the scaling of technology, cross-talk is becoming an important problem. By integrating the proposed approach with global routing, more accurate delay computation can be performed since second order effects like cross-talk can be estimated and avoided. The cross-talk avoidance algorithms proposed in [Kir97] can be used in this context.

# Bibliography

[Ass97] Semiconductor Industry Association. *National Technology Roadmap for Semiconductors*. 1997.

[ASSP90] P. Abouzeid, K. Sakouti, G. Saucier, and F. Poirot. multilevel synthesis minimizing the routing factor. In *The Proceedings of the Design Automation Conference*, pages 365–368, June 1990.

[Bak90] H. B. Bakoglu. *Circuits, Interconnections, and Packaging for VLSI*. Addison-Wesley Pub. Co., 1990.

[BHMSV84] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[BM82] R. K. Brayton and C. McMullen. The Decomposition and Factorization of Boolean Expressions. In *The Proceedings of the International Symposium on Circuits and Systems*, pages 49–54, May 1982.

[Bra00] R. K. Brayton. Personal communication. 2000.

[BRSVW87] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A multiple-level logic optimization system. *IEEE Trans on CAD*, CAD-6(6):1062–1081, November 1987.

[Cad99] Cadence Design Systems, Inc., 555 River Oaks Parkway, San Jose, CA 95134, USA. *Envisia Silicon Ensemble Place-and-route Reference*, Nov 1999.

[CK84] C.K. Cheng and E.S. Kuh. Module placement based on resistive network optimization. *IEEE Trans on CAD*, CAD-3(3):218–225, July 1984.

[CK94] J. Cong and C. K. Koh. Simultaneous driver and wire sizing for performance and power optimization. *IEEE Transactions on VLSI Systems*, 2(4):408–425, December 1994.

[CP98] J. Cong and Z. Pan. Interconnect Performance Estimation Models for Synthesis and Design Planning. In *The Proceedings of the International Workshop on Logic Synthesis*, June 1998.

[DJS91] K. Doll, F.M. Johannes, and G. Sigl. Domino: deterministic placement improvement with hill-climbing capabilities. *Proceedings of the IFIP International Conference on VLSI*, pages 91–100, August 1991.

[DK85] A. Dunlop and B. Kernighan. A procedure for placement of standard-cell VLSI circuits. *IEEE Trans on CAD*, CAD-4(1):92–98, January 1985.

[EJ98] H. Eisenmann and F.M. Johannes. Generic global placement and floorplanning. In *The Proceedings of the Design Automation Conference*, pages 269–274, June 1998.

[Elm48] W. C. Elmore. The transient analysis of damped linear networks with particular regard to wideband amplifiers. *J. Applied Physics*, (19):55–63, 1948.

[FJ98] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. volume 3, pages 1381–1384, May 1998.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman and Company, 1979.

[GvL96] G. H. Golub and C. F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore and London, 3 edition, 1996.

[HK97] D. J. Huang and A. B. Kahng. Partitioning-based standard-cell global placement with an exact objective. In *The Proceedings of the International Symposium on Physical Design*, pages 18–25, April 1997.

[Hu99] Chenming Hu. Personal communication. 1999.

[HV97] S. Hojat and P. Villarrubia. An integrated placement and synthesis approach for timing closure of powerpc microprocessors. In *Intl. Workshop on Logic Synthesis*, 1997.

[JS99] Y. Jiang and S. S. Sapatnekar. An integrated algorithm for combined placement and libraryless technology mapping. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 102–105, November 1999.

[Kir97] Desmond A. Kirkpatrick. *The implications of deep sub-micron technology on the design of high performance digital vlsi systems*. PhD thesis, University of California Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, December 1997.

[KMB+99] S.P. Khatri, A. Mehrotra, R.K. Brayton, A.L. Sangiovanni-Vincentelli, and R Otten. A novel VLSI layout fabric for deep sub-micron applications. In *Proceedings of the Design Automation Conference*, New Orleans, June 1999.

[KP89] F. J. Kurdahi and A. C. Parker. Techniques for Area Estimation of VLSI Layouts. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 8(1):81–92, January 1989.

[KSJA91] J.M. Kleinhans, G. Sigl, F.M. Johannes, and K.J. Antreich. GORDIAN: VLSI placement by quadratic programming and slicing optimization. *IEEE Trans on CAD*, 10(3):356–365, March 1991.

[LCP99] J. Lou, W. Chen, and M. Pedram. Concurrent logic restructuring and placement for timing closure. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 31–35, November 1999.

[LSP97] J. Lou, A. H. Salek, and M. Pedram. An exact solution to simultaneous technology mapping and linear placement problem. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 671–675, November 1997.

[OB98] R. H. J. M. Otten and R. K. Brayton. Planning for Performance. In *The Proceedings of the Design Automation Conference*, June 1998.

[PB91a] M. Pedram and N. Bhat. Layout driven logic restructuring/decomposition. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 134–137, 1991.

[PB91b] M. Pedram and N. Bhat. Layout driven technology mapping. In *The Proceedings of the Design Automation Conference*, pages 99–105, June 1991.

[Rud89] Richard Rudell. *Logic synthesis for VLSI design.* PhD thesis, University of California, Berkeley, April 1989. Tech. Report No. UCB/ERL M89/49.

[Sav92] Hamid Savoj. *Don't Cares in Multi-Level Network Optimization.* PhD thesis, University of California Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, May 1992.

[SCK92] A. Srinivasan, K. Chaudhary, and E. S. Kuh. Ritual: a performance driven placement algorithm. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 39(11):825–840, November 1992.

[She98] N. Shenoy. Personal communication. 1998.

[SID+99] N. Shenoy, M. Iyer, R. Damiano, K. Harer, H. Ma, and P. Thilking. A robust solution to the timing convergence problem in high-performance design. In *The Proceedings of the International Conference on Computer Design*, pages 250–257, October 1999.

[SK98] D. Sylvester and K. Keutzer. Getting to the bottom of deep submicron. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 203–211, 1998.

[SRRJ97a] G. Stenz, B. M. Riess, B. Rohfleisch, and F. M. Johannes. Timing Driven Placement in Interaction with Netlist Transformations. In *The Proceedings of the International Symposium on Physical Design*, Napa Valley, CA, 1997.

[SRRJ97b] G. Stenz, B.M. Riess, B. Rohfleisch, and F.M. Johannes. Timing driven placement in interaction with netlist transformations. In *The Proceedings of the International Symposium on Physical Design*, pages 36–41, 1997.

[SSL+92] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Technical Report UCB/ERL M92/41, UC Berkeley Electronics Research Laboratory, Univ. of California, Berkeley, CA 94720, May 1992.

[SSV84] C. Sechen and A. L. Sangiovanni-Vincentelli. The TimberWolf placement and routing package. In *The Proceedings of the Custom Integrated Circuits Conference*, May 1984.

[Tou90] Hervé J. Touati. *Performance-Oriented Technology Mapping*. PhD thesis, University of California Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, November 1990. Memorandum No. UCB/ERL M90/109.

[vG90] L. P. P. P. van Ginneken. Buffer placement in distributed rc-tree networks for minimal elmore delay. In *The Proceedings of the International Symposium on Circuits and Systems*, pages 865–868, May 1990.

[VP93] H. Vaishnav and M. Pedram. Routability-Driven Fanout Optimization. In *The Proceedings of the Design Automation Conference*, pages 230–235, June 1993.

[VP95] H. Vaishnav and M. Pedram. Minimizing the Routing Cost During Logic Extraction. In *The Proceedings of the Design Automation Conference*, pages 70–75, June 1995.

[VR90] J. Vasudevamurthy and J. Rajski. A method for concurrent decomposition and factorization of Boolean expressions. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 510–513, November 1990.