Timing-Driven Physical Design EDA Algorithms for Tackling Process Variations

Evriklis Kounalakis

December 2011

University of Crete

Department of Computer Science

Thesis submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Doctoral Thesis Committe: Christos Sotiriou, Associate Professor (Supervisor)
Manolis Katevenis, Professor
Ioannis Tollis, Professor
Apostolos Traganitis, Professor
Giorgos Georgakopoulos, Associate Professor
Dionysios Pnevmatikatos, Professor
Giannis Papaeystathiou, Assistant Professor

University of Crete

Department of Computer Science

Timing-Driven Physical Design EDA Algorithms for Tackling Process Variations

Dissertation submitted by

Evriklis Kounalakis

In partial fulfillment of the requirements for
the PhD degree in Computer Science

Author:

Evriklis Kounalakis, University of Crete

Examination Committee:

Christos Sotiriou, Associate Professor, University of Crete

Manolis Katevenis, Professor, University of Crete

Ioannis Tollis, Professor, University of Crete

Apostolos Traganitis, Professor, University of Crete

Giorgos Georgakopoulos, Associate Professor, University of Crete

---

Dionysios Pnevmatikatos, Professor, Technical University of Crete

---

Giannis Papaeystathiou, Assistant Professor, Technical University of Crete

Approved by:

Angelos Bilas, Chairman of Graduate Studies
Heraklion, December 2011

Acknowledgments

This work has been partly funded by FORTH-ICS, Greece, which the author would like to acknowledge.

I would like to thank Nanochronous Logic, Inc, which helped me uncover real problems that the industry is facing and has proven as an inspiration for this work. Being part of Nanochronous' R&D team, I also had the chance to expand on my software engineering skills, which helped me towards the demanding implementation aspects of this thesis.

I would like to thank my supervisor, Christos P. Sotiriou, who has been patient with me throughout all the years of this thesis' development. His advice has been valuable as to what kind of problems were most promising for research, as to helping me stay on track during my research and as to identifying the time to conclude my thesis.

I would like to thank my thesis internal committee, Christos Sotiriou, Manolis Katevenis and Ioannis Tollis, who have provided me with valuable directions and advice. I would also like to thank the other members of my thesis committee, Apostolos Traganitis, Giorgos Georgakopoulos, Dionysios Pnevmatikatos and Yiannis Papaeystathiou for their suggestions and advice.

I would like to thank my family for their encouragement and support, especially during the last 15 months, which proved to be the hardest.

Lastly, I would like to thank all the people who have been near me during all the years this research took.

Abstract

Moore's law states that the total number of transistors of an integrated circuit approximately doubles every two years. Maintaining this trend, requires tools able to cope with the ever-increasing complexity of chip design. Electronic Design Automation (EDA) has so far addressed this problem by providing automated tools and flows which enabled designers to handle chips consisting of more than a few millions transistors.

However, the ever shrinking of the size of transistors and interconnects, now poses new obstacles for designers and automated EDA flows. Smaller dimension devices, although providing more speed and less area, pose new challenges. Contemporary Deep-Sub-Micron (DSM) fabrication processes suffer from the presence of manufacturing variations, due to unpredictability in the exact dimensions and characteristics of transistors and wires. These variations now affect high-level characteristics of the chips such as their speed and power consumption. Technology vendors have always provided a number of characterizations for each circuit element at different operating scenaria (operating corners). Nowadays, more corners are needed to account for process variations, which adds to the complexion of achieving closure for all corners simultaneously.

One way to mitigate this phenomenon is to integrate multiple operating scenaria into a single, unified model, which can then be incorporated into existing flows. Statistical models offer this capability. They can encapsulate each corner into a random distribution, which can reflect the variation of speed and power consumption characteristics of the circuit elements. In this case, the delay and power becomes statistical rather than deterministic. Although such statistical models exist, their use in EDA flows has not been demonstrated. An alternative approach for combating variations is to design circuits which include clock-less or asynchronous speed-independent designs. These, possess the property of adjusting to their operating conditions instead of failing for fixed constraints. This approach requires further development of asynchronous circuits, the implementation of which has not been proven viable in EDA flows. Currently, there is significant lack of EDA tools capable of handling asynchronous circuits, making their use impossible in industrial designs.

In this work, we have developed and evaluated placement and post-placement optimization algorithms, which aim to tackle the problem of process variations in contemporary EDA flows. We present a novel placement algorithm, SCPlace, which based on a statistical timing model in its optimization engine, alleviates the need for multi-corner placement. SCPlace is the first

large-scale statistical optimization tool appearing in literature targeting placement, which is the cornerstone of physical implementation. SCPlace exploits statistical wire delay bounds, generated by our novel statistical slack assignment algorithms, which distribute slack according to statistical distributions. We have also developed a post-placement statistical leakage reduction algorithm, which is able to perform in-place statistical leakage reduction without negatively affecting statistical delay. Our third contribution is CPlace, a fully automated placer for asynchronous, cyclic circuits. CPlace is able to meet both performance and speed independent constraints.

Experimental results indicate that SCPlace compares favourably with state-of-the-art, industrial and academic placers, providing routable designs which achieve superior timing yield computed from the resulting statistical delay distributions. Our statistical leakage reduction flow achieves 20% average leakage reduction, without affecting the statistical delay of the preplaced circuit. Our results also show that CPlace provides routable placements for asynchronous circuit and superior placements compared to state-of-the-art industrial and academic placers which cannot guarantee speed independent constraints. All three of our flows have been designed with ease of integration into contemporary EDA flows in mind, through the use of only industry-standard formats and by collaborating with commercial EDA tools.

Περίληψη

Ο νόμος του Moore υποδεικνύει ότι ο αριθμός των τρανζίστορ σε ένα ολοκληρωμένο κύκλωμα διπλασιάζεται κάθε δύο χρόνια. Για να διατηρηθεί αυτή η τάση, απαιτείται τόσο οι διαστάσεις των τρανζίστορ να συρρικνώνονται, όσο και να υπάρχουν εργαλεία ικανά να χειριστούν την αυξανόμενη πολυπλοκότητα των κυκλωμάτων. Ο τομέας του Ηλεκτρονικού Σχεδιαστικού Αυτοματισμού (ΗΣΑ) μέχρι τώρα, αντιμετώπισε το πρόβλημα αυτό, προσφέροντας ροές και εργαλεία, τα οποία έκαναν δυνατό το χειρισμό κυκλωμάτων με πλήθος τρανζίστορ αρκετών εκατομμυρίων. Σήμερα όμως, τα εργαλεία ΗΣΑ πρέπει να αντιμετωπίσουν και το φαινόμενο της κατασκευαστικής μεταβλητότητας, το οποίο εισάγει αβεβαιότητα σε σημαντικά χαρακτηριστικά των κυκλωμάτων, όπως ο χρονισμός και η κατανάλωση.

Σε αυτήν τη διατριβή, αναπτύξαμε και αξιολογήσαμε αλγορίθμους βελτιστοποίησης για το στάδιο της τοποθέτησης και της βελτιστοποίησης μετά την τοποθέτηση, ώστε να αντιμετωπιστεί το φαινόμενο της μεταβλητότητας. Παρουσιάζουμε έναν καινοτόμο αλγόριθμο τοποθέτησης, SCPlace, ο οποίος βασιζόμενος στη στατιστική χρονική ανάλυση του κυκλώματος, χειρίζεται την αβεβαιότητα στον χρονισμό. Επιπλέον, αναπτύξαμε ένα εργαλείο βελτιστοποίησης της κατανάλωσης λόγω ρεύματος διαρροής, μετά την τοποθέτηση, το οποίο βελτιστοποιεί για κατανάλωση χωρίς να επηρεάζει αρνητικά το στατιστικό χρονικό ωφέλιμο του κυκλώματος. Τέλος, η τρίτη συνεισφορά της διατριβής αυτής είναι το εργαλείο CPlace, ένα εργαλείο τοποθέτησης, το οποίο μπορεί να χειρίζεται ασύγχρονα κυκλώματα, τα οποία είναι λιγότερο επιρρεπή στα αποτελέσματα της μεταβλητότητας.

Τα πειραματικά αποτελέσματα δείχνουν ότι το SCPlace πετυχαίνει καλύτερα αποτελέσματα από τα καλύτερα βιομηχανικά και ακαδημαϊκά εργαλεία όσο αφορά τη στατιστική συμπεριφορά του χρονισμού του κυκλώματος μετά την τοποθέτηση. Η ροή μας για βελτιστοποίηση της κατανάλωσης πετυχαίνει 20% μείωση της κατανάλωσης χωρίς καμία επίδραση στο στατιστικό χρονικό ωφέλιμο του κυκλώματος. Τέλος, το CPlace μπορεί με επιτυχία να χειριστεί ασύγχρονα κυκλώματα σεβόμενο τις χρονικές υποθέσεις οι οποίες είναι απαραίτητες για τη λειτουργία τους. Όλα τα εργαλεία που αναπτύξαμε συμμορφώνονται με βιομηχανικά στάνταρ και έχουν σχεδιαστεί ώστε να μπορούν να ενταχθούν άμεσα σε βιομηχανικές ροές υλοποίησης κυκλωμάτων.

Contents

Acknowledgments **iii**

Abstract **v**

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**
1.1 Electronic Design Automation 2
1.2 Major Technology Challenges 6
1.3 Drawbacks of Existing EDA Practices 7
1.4 Contributions of this Thesis 8

2 Background **11**
2.1 Timing Analysis 11
2.1.1 Synchronous Static Timing Analysis 12
2.1.2 STA-based Timing Optimization 14
2.1.3 Statistical Static Timing Analysis 15
2.1.3.1 SSTA-based Timing Optimization 16
2.1.4 Asynchronous Timing Analysis 18
2.1.5 Asynchronous Timing Analysis Models 19
2.1.5.1 Timing Separation of Events 20
2.2 Placement Algorithms 22
2.2.1 Optimization Objectives and Constraints 23
2.2.2 Requirements for a Placer 25
2.3 Algorithmic Approaches to Placement 26
2.3.1 Iterative vs Constructive Placement 27
2.3.2 Global vs Detailed Placement 27
2.3.3 Standard Cell Placement vs Mixed-size placement 28
2.3.4 Wire Bounds as an Optimization Directive 29

2.3.5 Taxonomy of Placers . . . 29
2.3.6 Challenges for Contemporary Placers . . . 30
2.3.6.1 Multi-Corner Placement . . . 30
2.3.6.2 Asynchronous Circuits Placement . . . 31
2.3.6.3 Post-placement Optimization . . . 32
2.3.7 Limitations of Contemporary Placers . . . 33
2.3.8 Our Approach to the Placement Problem . . . 34
2.3.8.1 Statistical/Multi-Corner Timing-Driven Placement . . . 34
2.3.8.2 Placement for Asynchronous Circuits . . . 35
2.3.8.3 Post-Placement Optimization . . . 35

3 Statistical Delay Bounds 37
3.1 Statistical Static Timing Analysis . . . 37
3.1.1 Statistical Gate Delay . . . 38
3.1.2 Statistical Delay Propagation . . . 38
3.2 Minimum Sigma Propagation . . . 44
3.2.1 Motivation and Intuition for MSSA . . . 44
3.2.2 MSSA for a Single Gate . . . 45
3.3 Minimum Sigma Slack Assignment . . . 47
3.3.1 MSSA Superfluous Constraints . . . 49
3.3.2 Runtime Issues . . . 50
3.4 Target Sigma Propagation . . . 50
3.4.1 Motivation and Intuition for TSZSA . . . 50
3.4.2 TSZSA for a Single Gate . . . 52
3.4.3 TSZSA Algorithm . . . 53
3.4.4 TSZSA Wire Delay Propagation . . . 53
3.5 Target Sigma Zero Slack Assignment . . . 54
3.6 LP slack assignment . . . 56
3.6.1 LP formulation for Statistical Slack Assignment . . . 58
3.6.2 Runtime Improvement Through Hierarchical LP . . . 59

4 SCPlace 63
4.1 Motivation for SCPlace . . . 64
4.2 Description and Intuition for SCPlace . . . 65
4.3 Requirements for Statistical Placement . . . 66
4.4 SCPlace Interface . . . 67
4.5 Optimization Objectives . . . 68
4.6 The SCPlace Flow . . . 69
4.7 Implementation Details . . . 71
4.7.1 Constructive Process . . . 71
4.7.2 Reconstruction . . . 72

4.7.3 Perturbation . . . 75
4.7.4 Finalization . . . 77
4.7.5 Routability . . . 78
4.7.6 Legalization . . . 78
4.7.7 Slack Reassignment . . . 79
4.8 SCPlace Hierarchical Approach . . . 83

5 Post-Placement Statistical Leakage Optimization 85
5.1 Statistical Leakage Optimization . . . 85
5.2 Statistical Leakage Optimization Requirements . . . 86
5.2.1 Physical Information . . . 87
5.2.2 Timing Analysis and Leakage Model . . . 87
5.2.3 Timing and Leakage Constraints . . . 88
5.2.4 Gate Substitution . . . 89
5.3 Statistical Leakage Optimization Interface . . . 89
5.4 Optimization Objectives . . . 90
5.5 Leakage Optimization Flow . . . 92
5.6 Optimization Flow Details . . . 95
5.6.1 Statistical Slack Assignment . . . 95
5.6.2 Gate Sorting . . . 96
5.6.3 Incremental SSTA . . . 97
5.6.4 Slack Reassignment . . . 98
5.7 Routability and Legalization . . . 102
5.8 Runtime Issues . . . 103

6 CPlace 105
6.1 Asynchronous Placement Requirements . . . 105
6.2 CPlace's Interface . . . 106
6.3 CPlace Objectives . . . 107
6.4 Slack Assignment for Asynchronous Circuits . . . 109
6.4.1 Wire-Delay Bounds . . . 109
6.4.2 LP Formulation . . . 110
6.5 The CPlace Flow . . . 112
6.6 CPlace Implementation Details . . . 114
6.6.1 Constructive Process . . . 114
6.6.2 Reconstruction . . . 114
6.6.3 Perturbation . . . 116
6.6.4 Finalization . . . 116
6.6.5 Routability and Legalization . . . 117
6.7 Runtime Issues . . . 117

7 Results **119**
7.1 Benchmark Set . . . 120
7.1.1 Synchronous Benchmarks . . . 120
7.1.2 Asynchronous Benchmarks . . . 121
7.2 Slack Assignment Results . . . 122
7.2.1 Slack Assignment Runtime . . . 127
7.3 SCPlace Results . . . 129
7.3.1 Timing Yield . . . 129
7.3.2 Routability . . . 132
7.4 Leakage Recovery Results . . . 133
7.4.1 Leakage Recovery vs Industrial . . . 135
7.4.2 Delay-Leakage Tradeoff . . . 136
7.4.3 Leakage Recovery Runtime . . . 136
7.5 CPlace Results . . . 138
7.5.1 CPlace vs Industrial and Capo . . . 139
7.5.2 QDI Satisfaction . . . 140
7.5.3 CPlace Runtime . . . 143

8 Conclusions **145**
8.1 Future Work . . . 146

A Synchronous and Asynchronous Timing Models **149**
A.1 Static Timing Analysis Models . . . 149
A.2 Statistical Timing Analysis Models . . . 151
A.3 Variation and Correlation Models . . . 152

B Placement Approaches **159**
B.1 Simulated Annealing . . . 159
B.2 Genetic . . . 160
B.3 Min-Cut . . . 161
B.4 Analytical . . . 162

C Statistical Distributions **163**
C.1 Normal Distribution . . . 163
C.2 Log-Normal Distribution . . . 164

References **167**

List of Figures

1.1 Contemporary design flow . . . 3
1.2 Thesis contributions . . . 9

2.1 STA example annotation . . . 13
2.2 STA violating and non-violating paths . . . 15
2.3 Timing yield . . . 16
2.4 Cases of distributions' relative position . . . 18
2.5 Statistical optimizations . . . 18
2.6 A simple petri-net with initial marking . . . 19
2.7 A *xyz*-STG . . . 20
2.8 A *xyz*-ER . . . 21
2.9 A process graph . . . 21
2.10 A process graph unfolding . . . 22
2.11 Layout manufacturing grid . . . 26

3.1 Progression of SSTA . . . 42
3.2 Monte Carlo analysis compared with SSTA result . . . 43
3.3 MAX of two normal distributions . . . 45
3.4 3-input gate MSSA delay assignment example . . . 46
3.5 σ(MAX), as a function of mean . . . 47
3.6 MSSA example with a superfluous constraint . . . 49
3.7 Applying TSZSA offset to gate inputs . . . 53
3.8 Optimizing successor level bounds - possible cases . . . 54
3.9 Hierarchical support for LP . . . 60

4.1 Bounding box estimation . . . 67
4.2 SCPlace's flow . . . 69
4.3 Upper and lower bound constraint types . . . 71
4.4 Multiple physical distance constraints . . . 72
4.5 Reconstruction example . . . 73
4.6 Placement of a cell on a valid location . . . 79

4.7 SCPlace slack reassignment example . . . 81
4.8 Hierarchical SCPlace . . . 84

5.1 Statistical leakage extrapolated PDF example . . . 88
5.2 Leakage reduction flow . . . 93
5.3 Re-Sizing example . . . 95
5.4 Distributing unused slack with incremental SSTA . . . 99
5.5 Re-distributing slack after one full optimization iteration . . . 101
5.6 Overlaps after standard cell resizing . . . 103

6.1 Absolute and relative constraints . . . 110
6.2 CPlace flow . . . 112
6.3 CPlace constructive bounds . . . 114

7.1 TSZSA timing yield gains . . . 125
7.2 ZWD, ZSA, MSSA and TSSA comparison . . . 126
7.3 TSSA mean, sigma tradeoff curve . . . 127
7.4 SCPlace timing yield improvement . . . 130
7.5 **b05** after detailed routing . . . 133
7.6 Timing yield loss after leakage reduction . . . 136
7.7 Delay-leakage tradeoff . . . 137
7.8 Experimental flow . . . 142

A.1 Rise (fall) transition depending on input rise (fall) transition and output driving capacitance . . . 150
A.2 Physical layout quadrisection . . . 157

C.1 Normal Distribution pdf and cdf . . . 164
C.2 Log-Normal Distribution pdf and cdf . . . 165

List of Tables

7.1 Synchronous benchmarks . . . 120
7.2 Synchronous large benchmarks . . . 121
7.3 Asynchronous benchmarks . . . 122
7.4 Comparison of zero wire delay with MSSA . . . 123
7.5 Comparison of ZSA and TSZSA results for given slack, illustrating yield improvement . . . 124
7.6 MSSA and TSZSA runtime . . . 128
7.7 Placement results comparison . . . 129
7.8 SCPlace runtime breakdown . . . 131
7.9 Initial placement . . . 134
7.10 Timing yield comparison . . . 135
7.11 Leakage optimization runtime breakdown . . . 138
7.12 Synchronous placement results comparison . . . 139
7.13 Comparison of relative constraints violations for CPlace and the industrial placer 141
7.14 CPlace runtime breakdown . . . 144

Chapter 1

Introduction

Moore's law, which states that the number of transistors in a chip doubles every two years, has always been then driving force of the semiconductor industry, driving for faster and more power efficient circuits.

If Moore's law is met, then a positive feedback phenomenon takes place. The faster and more efficient chips, allow for more integration, which in turn, allows for fabricating even faster circuits. However, by the mid-70s, although circuits were still designed by hand, it became apparent that the complexity of the design process for chips with large scale integration made it difficult to follow Moore's law [30]. The first step to mitigate this effect was to partition the implementation process into discrete steps. However, the complexity of even a single step was starting to become overwhelming for designers. Steps of the flow like translation of circuit into logic gates, placement of the gates and interconnecting them, could no longer be done by hand. The ever increasing complexity called for fully automated approaches, which would enable designers to cope with the exponential growth in circuit sizes. Automated tools began to appear, creating a new section of the semiconductor industry, *i.e.* Computer Aided Design (CAD) which evolved into fully automated flows, described by the term Electronic Design Automation (EDA). EDA encompasses a set of tools available to circuit designers, which enable them to implement and fabricate complex circuits with more than a few million transistors. According to the International Technology Roadmap for Semiconductors (ITRS) [34], versatile EDA tools will be needed in the forthcoming years, coupled with more complex flows.

1.1 Electronic Design Automation

The first EDA tools, during the 80's, consisted mainly of placement and routing tools [30]. EDA tools of this era, helped designers cope with the increasing number of transistors which had to be placed and routed on the layout. However, they were still away from fully automated tools, as they required for the designer to make important decisions, which affected chip characteristics, such as timing and power.

As EDA technology continued to develop, the complexity of, invariably, all steps of design flow began to overwhelm the designers. Thus, the need for automated tools became more prominent. By the mid-80s an increasing number of EDA tools appeared, both in academic and industrial environments [30]. One major breakthrough occurred in 1986, when the hardware description language (HDL) *Verilog* was introduced. In 1987, a second description language, *VHDL* was developed. The introduction of HDLs enabled a high-level more comprehensive description of circuits. Additionally, it allowed for formal interaction between EDA tools, targeting different steps of the flow.

In the next few years, basic steps of the standard design flow were defined, so as to divide the complex problem of chip implementation into smaller, manageable sub-problems. As a consequence, specialized EDA tools were developed, for virtually every step and aspect of the flow. As EDA tools became more advanced, their automation capabilities and features increased. Standard design practices were also developed so as to allow for more comprehensive specifications and better debugging. The design practice of synchronous circuits, which assumes that all synchronization is done with one, or more global signals, became standardized. This allowed for the definition of the frequency at which the circuit operates. Advances in EDA tools and standardization of design implementation led to the emergence of high-level constraints, which guided their algorithms, with the most important constraint being timing constraints, which specify that the circuit must operate at a given frequency.

The importance of timing constraints have led EDA tool developers to integrate static timing analysis engines within the optimization process in order to achieve timing closure at all steps of the design flow. This led to timing-driven algorithms and tools, which are commonly used today and the focus of EDA shifted entirely to synchronous circuits.

Despite the complexity of modern EDA tools, they have become essential for designing complex chips and meeting time-to-market constraints. It is not uncommon for a contemporary chip to go through ten or more dedicated EDA tools and steps, each covering a fraction of the

implementation aspects and interacting only with the previous and the next EDA tool in the design chain.

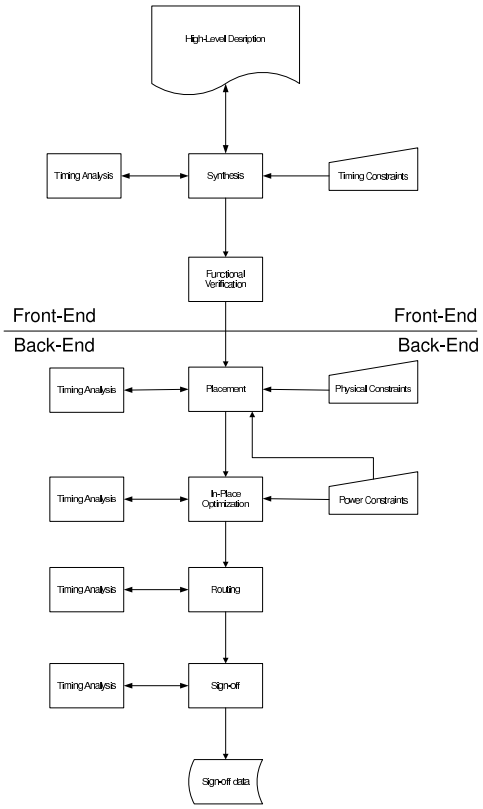


Figure 1.1: Contemporary design flow

Figure 1.1 shows a typical design flow for contemporary, synchronous designs, which typically target three optimization objectives, *i.e* power, performance and area. Initially, a high-level description of the circuit is provided by the designer. The first step is to translate the description, which is typically in *Verilog* or *VHDL* into logic circuit components. These components are logic gates or logic macros such as memories, adders, multipliers etc. The description derived from this translation step, and termed *Synthesis*, uses generic gates or macros, independent of technology. The next step is to map each generic component into a real technology-specific element, provided and described in a technology library. This is performed in the *Technology Mapping* phase. After technology mapping, the circuit consists entirely of technology components. These can be gates, small macros, or larger macros. In the former case, each component is a standard cell, whose characteristics such as area, timing, power are described in the technology libraries. In the latter cases, each component is a small or large logic block,

whose characteristics are described by separate technology files. Both synthesis and technology mapping in contemporary flows, use static timing analysis to achieve timing closure. Technology mapping is the last step of *Front-End* design, which refers to the design steps that lack physical information.

Back-end design on the other hand, refers to the physical steps of the design flow. The first step is *Placement*. Placement's objective is to identify appropriate locations for all components of the design, *i.e.* standard cells and macros, within a physical layout. Placement's quality is determined according to the constraints that a placer supports. One type of constraints are physical constraints, which refer to the size and shape of the layout. Such constraints describe the locations on which any component can be placed. In contemporary industrial placers, timing constraints guide the optimization towards timing closure. There may be other types of constraints, *e.g.* power constraints, which can also affect the relative locations of components. After placement, there may be a step for in-place optimization, which allows for local optimizations with respect to timing, area and power with little changes over standard cells. This may refer to replacing standard cells with other, equivalent cells, in order to *e.g.* reduce a cell's power consumption. However, the degree of freedom for an optimization algorithm at this step is significantly smaller than that of a placer, so the amount of optimization that can be achieved in this step is significantly smaller than the potential optimization through placement. Consequently, it is important that the placement process does not introduce violations, which cannot be fixed by in-place optimization. After the design has been placed, the components must be connected by wires, which is done in the next step, *Routing*. Routing typically works closely with STA, similarly with the previous steps of the design flow. Routing is the last step where optimization is possible, *i.e.* after routing only small corrections can be made by experienced engineers through Engineering Change Order (ECO). It is also the last step, which can significantly affect circuit's timing, as it has control over the length and dimension of individual wires. Any violations introduced in this step are not likely to be fixed in any step beyond that. The final flow step is static sign-off where the circuit is validated against all constraints. The design undergoes detailed STA using accurate delay models, which encapsulate cell and wire delays under worst-case conditions. Power and design-rule analysis is also done in order to validate the design against the required power performance and manufacturing requirements.

A typical timing-driven design flow for synchronous circuits relies heavily on STA. This has led to the development of accurate STA tools and methodologies. STA's task is to calculate the arrival times of data from the circuit's startpoints to all endpoints for all circuit's paths.

The slowest path defines the worst-case delay of the circuit.

At first, timing analysis was done by hand, without the use of any automated tools. The designer would compute the amount of time required for data to propagate from inputs to outputs, by taking into account estimated gates and wires delays. However, due to the large number of computations, doing this by hand requires too much time. Soon, it became clear that timing analysis should be automated to be efficient.

The most prominent approach, still used today, is called Static Timing Analysis (STA), which is perfectly suited for the synchronous-design practice. Synchronous circuits are modeled after the notion that the whole circuit is synchronized with a global signal, called clock. The computations are done with combinational elements and the results are stored in sequential elements at each clock pulse. Using this assumption, STA can estimate whether the combinational elements can perform their task during one clock cycle and how long this clock cycle is required to be. STA operates on an annotated directed and acyclic timing graph, from which the delay for each timing element (gate or wire) can be extracted. By traversing the graph, it computes the worst-case arrival times at all points in the circuit and determines the longest delay paths, which define the circuit's delay. STA has proved to be very accurate and fast and has established itself as the golden sign-off tool for guaranteeing the delay and the frequency at which the circuit can operate after fabrication.

Another method of timing analysis is with the use of simulators [7]. This kind of analysis refers to dynamic timing analysis. Dynamic timing analysis applies vectors to the inputs of the circuit and then calculates the time that is required for the result at the endpoints to be stabilized. Since delay depends on input vectors, it is essential that a number of different vectors are applied. Then, the worst-case, the best-case and the average-case arrival times may be inferred. This approach is very accurate if the number of inputs is very small so that all possible combinations for inputs can be applied. However, for contemporary circuits, where the number of inputs and startpoints can be more than a few thousands, simulators for timing analysis have become obsolete.

Although flows for synchronous circuits have been well established in EDA today, synchronous design is not the only promising design methodology. Asynchronous circuits offer an alternative way to designing circuits. These, do not rely on a global clock signal as to when operations should be performed. Instead, data propagate through the circuit in accordance with local "handshaking" signals, *i.e.* an element communicates with its neighbouring elements as to when they are ready to receive new data. This way, operations are performed as soon as

possible rather than waiting for a global clock signal. Potential benefits of asynchronous design include less power consumption, due to the absence of the power-consuming clock network, modularity, adaptation to operating conditions, less electromagnetic interference, due to the lack of global synchronization and robustness to process variations, as elements synchronize locally depending on their true speed characteristics [59, 19]. Although the synchronous practice has been well established today, still parts of contemporary circuits are largely asynchronous in nature and operate in an autonomous manner. It is projected by ITRS [34], that by 2020, 40% of a general purpose CPU will consist of asynchronous parts.

Despite the advances in EDA tools industry, over all aspects of design implementation, there are still challenges, which need to be addressed. In the next section we describe the major challenges that contemporary EDA tools need to tackle and we discuss deficiencies of current EDA methodologies, which are starting to emerge.

1.2 Major Technology Challenges

The first challenge stems from the fact that circuit elements constructed in Deep Sub-Micron (DSM) technologies exhibit uncertainty in their characteristics. This is the result of process variations, which stem from fluctuations in physical and electrical characteristics. Other operating variations include temperature and voltage levels. Fluctuations in temperature and voltage have always had impact on timing and power of circuit elements, which has led technology vendors to supply characterizations (corners) under different operating scenaria. With the introduction of process variations, the number of corners has increased further. Worst, best and average case scenaria are provided for any combination of process-voltage-temperature case. A designer must account for all corners so as to guarantee that the design will work under any operating and fabrication condition by employing multi-corner analysis techniques. EDA tools then, must be coupled with methods to achieve closure for all corners. Although this is possible, it requires a lot of effort from the engineers, as the constraints needed for achieving closure for one corner, *e.g.* worst-case may be contradicting with the constraints for another corner, *e.g.* best-case. Typically, engineers guard-band optimization constraints on one corner by applying a margin so as to ensure that the design will not fail another corner. This can directly lead to loss of speed or power, as the margin is there only to enhance the probability that the design achieves closure at all corners [29].

On the other hand, a unified approach to enable simultaneous optimization for all corners,

would reduce the effort required by the designer and would eliminate the need for excessive margins. One possible way towards this direction is the inclusion of all corners into a statistical model [39, 3]. This model will encapsulate the probability that any corner will manifest itself during the circuit's lifetime making optimization statistical in nature. Today, there have been various proposed methodologies for describing and optimizing for statistical delay and power [13, 9, 36, 5, 79]. However, there is still the need for incorporating these methodologies into current EDA flows.

A second challenge for EDA tools is that the diverse nature of contemporary circuits drives designers outside the framework of typical design flows. It is not uncommon today, for example, to design cyclic circuits, which breaks the fundamental rule that any circuit graph should be acyclic, so as STA can be applied. Cyclic circuits are essential for the implementation of architectures that implement local synchronization instead of relying on a global clock, in order to take advantage of potential gains in power and adaptability [59, 19, 6]. These are asynchronous circuits, whose operation is of concurrent nature, requiring different modeling than circuits of synchronous, sequential nature. The challenge lies on the requirement to account for the acyclic nature of circuits in EDA flows.

Another challenge for EDA tools is that leakage power is becoming increasingly important in contemporary designs. Leakage current's magnitude has traditionally been negligible compared to other sources of power consumption such as dynamic power. However, leakage current's exponential dependence on threshold voltage, renders it increasingly important. The lower the threshold voltage, the higher the leakage and this trend is likely to continue in the forthcoming technologies, where threshold voltage is expected to be further decreased to allow for lower supply voltage levels without negatively impacting performance [28]. Furthermore, variations in the physical dimensions of circuit elements and electrical properties, directly impacting threshold voltage, cause fluctuations in leakage current, making imperative multi-corner or statistical analysis.

In the next section, we explore the drawbacks of existing EDA tools, stemming directly from the challenges they fail to address.

1.3 Drawbacks of Existing EDA Practices

The first serious issue is that multi-corner analysis is not efficient for chips with tight constraints. Multi-corner analysis requires the introduction of margins, even for the best or typical

cases to allow for closure, minimizing the probability of failing in other corners. Multi-corner analysis is also too time consuming and reduces time-to-market for circuit engineers. Statistical approaches, which could help mitigate these problems have not yet been fully incorporated into EDA tools.

The second issue is that STA, which is the backbone of contemporary timing-driven optimization tools, can only be applied to circuits from which a Directed Acyclic timing Graph (DAG) can be constructed. This is easy for synchronous circuits, where sequential elements are boundary elements and thus, no cycles are present in the corresponding timing graph. However, for asynchronous circuits this is not true. Thus, efficient timing-driven optimization of asynchronous circuits is not currently supported.

A third issue stems from the fact that leakage current is becoming more important than dynamic power consumption in DSM chips. Like timing, closure for leakage will require multi-corner analysis or statistical optimization. However, contemporary EDA tools do no currently support statistical leakage analysis or optimization.

1.4 Contributions of this Thesis

In this thesis we address solutions for the lack of EDA tools able to handle statistical optimization for timing and leakage and lack of flows for alternative design approaches like asynchronous circuits. We have created three EDA tools, each one addressing one of the three specific deficiencies that we have identified for contemporary EDA tools. Next, we briefly describe each one of our contributions.

We have developed a placement algorithmic tool, called *SCPlace* [48], which incorporates a statistical timing analysis engine in its inner loop of optimization. SCPlace employs the use of novel statistical timing bounds for wires, which guarantee that the circuit's endpoints meet statistical constraints, with respect to both the mean and the standard deviation of the statistical delay.

We have developed *CPlace* [47], a constructive placement algorithm able to handle asynchronous circuits. CPlace is timing-driven in the sense that it uses novel bounds for the delay that each wire is allowed to have. The bounds are inferred by Asynchronous Timing Analysis (ATA), which guarantees both the performance and the correctness of the asynchronous circuit's concurrent behaviour.

We have also developed a post-placement leakage recovery flow [49], which employs the

same wire bounds as SCPlace in order to reduce the statistical leakage without affecting the statistical timing of the circuit. This is an in-place optimization flow, which works by identifying gates that can afford an increase in their statistical delay without violating any constraints. These gates are then optimized so that their statistical leakage is reduced.

A visualization of this thesis' contributions and the way they fit into a contemporary EDA flow is shown in Figure 1.2.

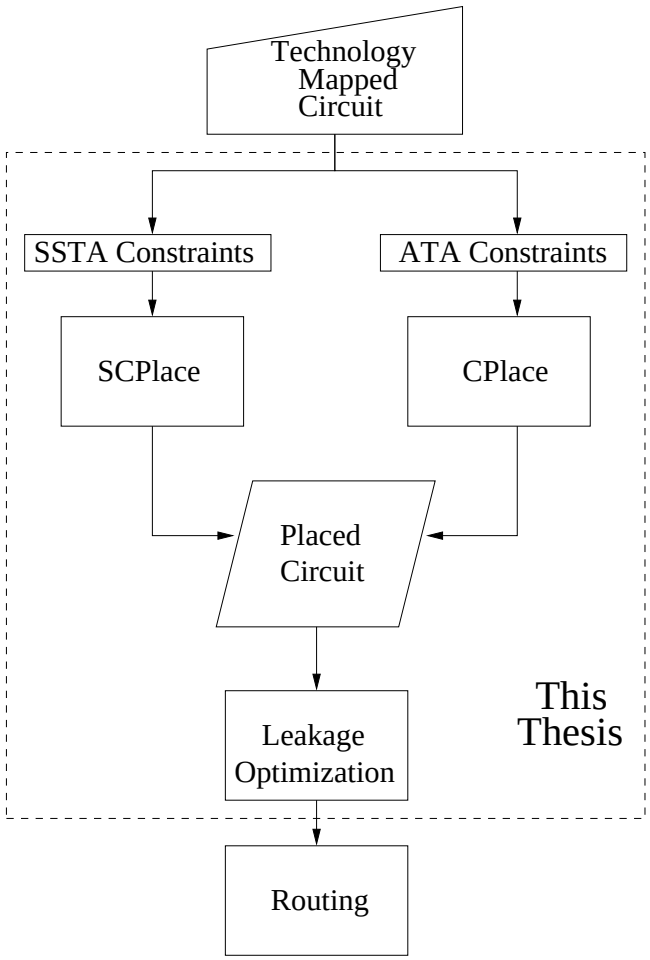


Figure 1.2: Thesis contributions

The rest of this thesis is structured as follows. Chapter 2 provides background on timing analysis, leakage and the placement problem. Chapter 3 explains our framework for extracting statistical bounds for wires, which will then be utilized by our optimization tools. Chapter 4 details our constructive statistical placement algorithm, *SCPlace*. Chapter 5 describes our

post-placement statistical leakage recovery flow, which can be applied either after SCPlace, or after any industrial placement tool. Chapter 6 presents our constructive placement tool, *CPlace* which can handle asynchronous circuits. Chapter 7 shows the evaluation of our flows and tools and Chapter 8 presents the conclusions of this thesis.

Chapter 2

Background

In this chapter, we present the general framework under which (i) timing analysis is performed, (ii) the placement problem is formulated and (iii) the leakage problem is addressed. Our approach to these issues will be addressed in the next chapters.

2.1 Timing Analysis

The benefits of timing analysis have been well understood by academia and industry. Timing analysis has been incorporated into the main steps of design flow, including synthesis, placement and routing. Most algorithms for these steps use timing analysis in their inner loop of optimization in order to validate on the spot the progress of optimization. Thus, timing analysis must be kept up to date with advances in delay modeling, advances in the flow of optimization algorithms and must also be applicable to as many types of circuits as possible.

Timing analysis can be performed by either dynamic or static methods. The former is done by applying input vectors and measuring the time required for outputs to appear. However, this method is cumbersome and prone to errors if the input vectors are not exhaustive, or if the simulation scenario does not fit with the actual conditions the circuit will experience. Static methods do not depend on input vectors, but measure the delay using case analysis. Worst, best or average delays can be inferred by the circuit and the case characterization of the circuit's elements delay behaviour. Both types of timing analysis can be further categorized as deterministic or statistical, depending on the actual delay model they employ.

Regardless of the nature of timing analysis, optimization algorithms can benefit from it.

First, timing analysis provides estimates as to whether the optimization algorithm has met the timing constraints. Secondly, if the timing constraints are not met, timing analysis can offer guidelines as to which components are most critical, or most violating and thus need to prioritized in optimization. As timing analysis can also quantify violations, the optimization process also knows how hard it must try to meet the constraints, or if the constraints are unrealistic and thus infeasible.

2.1.1 Synchronous Static Timing Analysis

Static timing analysis (STA) is the cornerstone of timing in EDA. STA is performed on a timing graph representation of the circuit. The timing graph, which must be directed and acyclic, consists of vertices for gate pins and edges for the wires. A set of startpoints and endpoints allows for a forward traversal whereupon STA determines the arrival times at each node. Arrival time for each node is one of the two values timing analysis aims at determining. The second one is required time, which can be inferred by the timing constraint. Required time is the absolute time at which computations must be finished in the circuit's internal nodes, so that computations at the circuit's endpoints do not violate the timing constraints. Next, *slack*, is defined as the difference of required time minus arrival time. Slack quantifies the amount of timing violation for each node. Positive slack means that the circuit is faster than required, while negative slack means that the circuit violates the timing constraint. In a typical synchronous circuit, each path starts from either a primary input or a sequential element, *e.g.* a flip-flop. which are boundaries for paths. Each path then, ends at either a primary output or a sequential element. A gate may not appear more than once in a single path, thus the timing graph will not contain any cycles. STA requires annotation on the timing graph regarding the timing characteristics of each timing node. A gate characterization library provides the delay of each gate (vertex). In the case that the wire lengths are known, *i.e.* post-placement, then a number corresponding to the delay of the wire can be applied to each edge. STA traverses the graph, typically in Breadth-First-Search (BFS) fashion and at each node it annotates a number which corresponds to the delay of the longest path leading to that node. After all nodes have been annotated, the delay of the circuit is defined as the largest delay in the set of endpoints, in the case of max-delay analysis. In the case of min-delay analysis, the endpoint with the smallest delay is selected. An important property of STA is that it can store, for each timing node, its preceding timing node which causes the delay currently annotated. This allows for

path reconstruction with a simple traceback. Path reconstruction is important, as the delay of each path, as well as the amount of timing violation for each path, can be inferred.

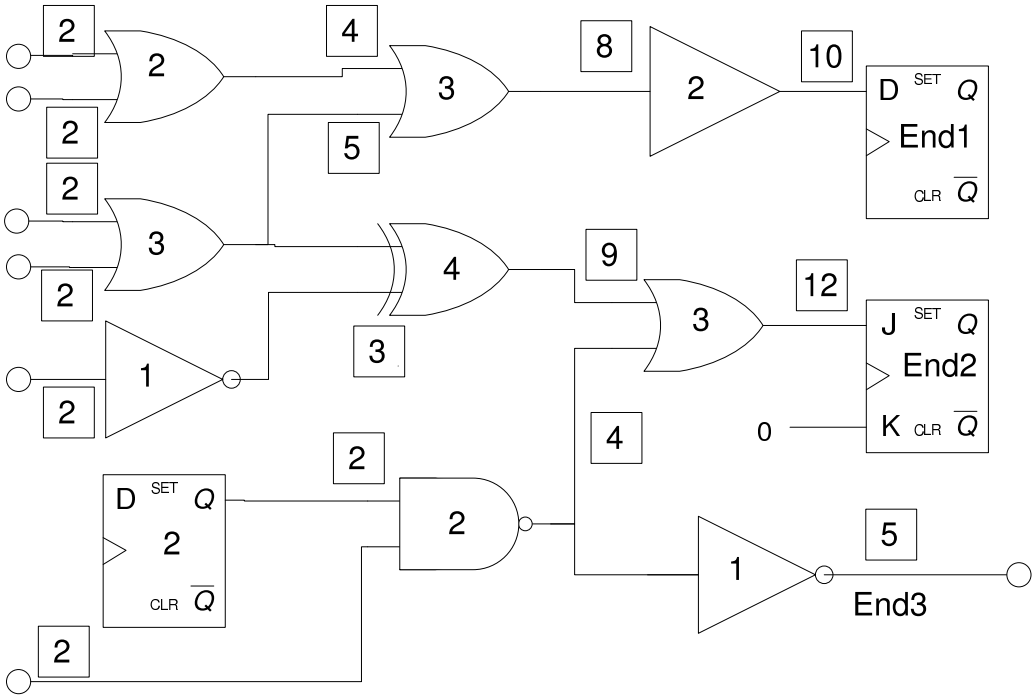


Figure 2.1: STA example annotation

Figure 2.1 shows an example of max-delay annotation for a simple circuit. The numbers in each gate denote gate delay. The numbers on each wire denote the path delay up to that point. All inputs are assumed to have an arrival time of 2 timing units. The annotation depicted in Figure 2.1 is the result of STA and is shown in boxes over each gate pin. In the example of Figure 2.1, the max delay of the circuit is 12 timing units, as defined by the delay of the slowest endpoint *End2*.

In order to be able to perform static timing analysis, a number of requirements are necessary. These are

- **Timing characteristics for all gates that constitute the design.** In contemporary flows, these are derived from characterization libraries for standard cells. These libraries provide timing information for all standard cells under a number of possibilities like the possible drive strengths, loads and number of fanouts. A number of technology libraries for each operating corner is also necessary, as the designer must guarantee that the chip

will always work under all possible conditions with respect to the most important factors affecting timing performance: process, voltage and temperature fluctuations.

- **A gate-level netlist representation of the circuit.** Otherwise, STA will not be able to extract the timing information for each design element from the technology library.

- **A directed graph representing the circuit.** Since STA identifies paths on the circuit starting from the inputs, or startpoints and ending at the outputs, or endpoints, a suitable graph representation of the circuit is required.

- **In case of industry-standard tools, *LEF*, *LIB* files.** *LEF* stands for Library Exchange Format [50] and contains all the physical descriptions of the cells, including wire capacitance and the number of metal layers. The unit capacitance and resistance values are used in wire delay models. *LIB* stands for the Liberty format [52] and is a technology library which contains timing and power information for all standard cells of the library.

2.1.2 STA-based Timing Optimization

STA, not only can provide timing information about the design, but can also help towards design optimization. Regardless of the actual implementation stage of the chip, STA may be used to direct optimization efforts. In timing-based optimization, the designer imposes a timing constraint for the circuit. STA is used to calculate the delay of each path. Compared to the timing constraint, some paths will meet the constraint and some will violate the constraint, as shown in Figure 2.2. The task of the optimization process is to fix, at each iteration, as many paths as possible, without creating new violating paths. At each iteration the number of violating paths should be decreased, making optimization more focused on less violating paths.

The most common way of implementing the optimization process is to employ a slack allocation algorithm. Path slack, *i.e.* the amount of timing violation for each path in the case that slack is negative, is transformed into net/gate slack. Net/gate slack is then, the amount of optimization that must be applied to each net/gate. Thus, the optimization algorithm will typically identify the elements of the circuit with the most negative slack, which are in greatest need for optimization and will try to fix them. After the slack of some nets/gates is fixed, STA is performed again for validation and identification of the new violating elements.

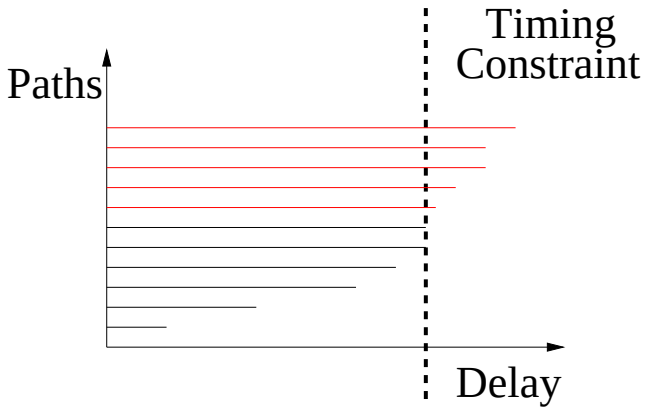


Figure 2.2: STA violating and non-violating paths

2.1.3 Statistical Static Timing Analysis

Statistical static timing analysis (SSTA) is the extension of STA where all delays are expressed as random variables instead of real numbers. Like STA, SSTA is performed on a timing DAG representation of the circuit with the goal of determining the arrival times, required times and slack of all the circuit's elements. All these metrics correspond to random variables. This leads to additional requirements that must be considered for the development of an SSTA engine. These are:

- **Selection of random distribution for gate delay.** The expression of a gate's delay must encompass all the delay possibilities under all process and operating conditions. Normal distributions are widely adopted in literature [62]. Experimental results on actual delays have shown that normal distributions correlate well with the actual delays [24]. Alternative approaches are described in Appendix A.

- **Correlation model.** Experimental results have shown that there is strong physical correlation in terms of variability. This is intuitively explained by the fact that gates which are in close proximity tend to be affected in similar ways by the sources of variation, *e.g.* temperature, voltage, process. Correlation models in literature [69, 62, 24, 3] use functions which assign high correlation to gates which have small geometric distance. An overview of correlation models is given in Appendix A.

- **Statistical performance evaluation.** Two delays in terms of STA are easily compared; the smaller delay is usually better in terms of optimization. Normal distributions, in terms

of SSTA, however, are described with two numbers, one denoting the *mean* and the other the *standard deviation (sigma)*. Comparing two normal distributions must be done in statistical terms. The statistical metric that is used in literature is that of timing yield, which expresses the probability that the delay meets the constraint. Given a distribution and a constraint, this probability can be computed analytically. Timing yield is illustrated in Figure 2.3. It essentially is the area of the distribution which is to the left of the timing constraint.

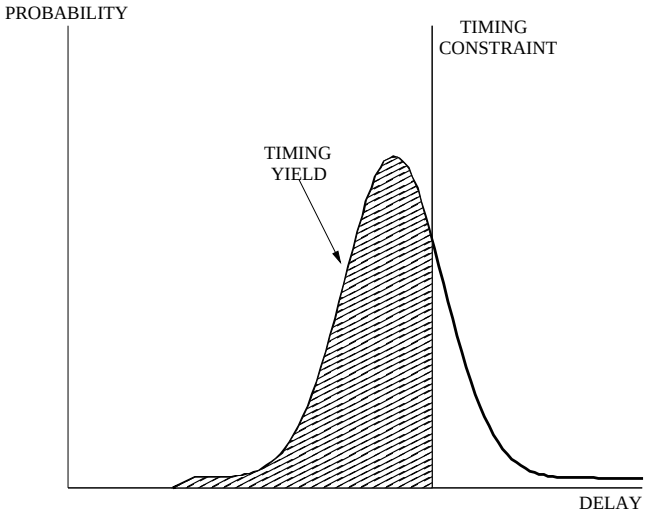


Figure 2.3: Timing yield

2.1.3.1 SSTA-based Timing Optimization

Given the framework for statistical timing analysis, we can perform statistical timing optimizations. Compared to traditional non-statistical timing analysis, SSTA-based optimization allows for a greater number of different optimization types.

In non-statistical optimization the target is clear; optimize the delay. The delay is expressed as a single number, so optimizing for delay means that this number is reduced, or placed below a constraint. With the introduction of statistical timing, delay is no longer a simple number, but is represented by a distribution. Analytically, the distribution, which in our case is a normal distribution, can be adequately described by its first two moments, *i.e.* its mean and

its standard deviation (sigma). Statistically optimizing the delay can mean optimizing for mean delay, sigma of delay, or both.

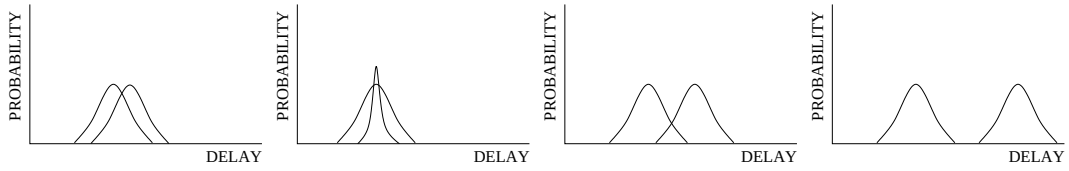
Figure 2.5a illustrates the first type of optimization. Here, the goal of optimization is to shift the distribution to the left. This means that the probability that the random variable takes smaller values is increased. In other words, the target is to minimize the mean delay, while not increasing the standard deviation.

The second type of optimization is shown in Figure 2.5b. Here the standard deviation is minimized. This directly corresponds to minimizing uncertainty of delay. As shown in Figure 2.5b the range of values that the random variable is allowed to take has been reduced. Although in this particular example the mean delay is increased, reduction of standard deviation does not necessarily incur a penalty in the mean delay. Even if it does so, it might be preferable for the designer to minimize uncertainty at the cost of some delay, particularly if the new mean delay does not violate the timing constraint.

A combined type of optimization is shown in Figure 2.5c. In this case, the goal of optimization is to maximize the timing yield. By appropriately manipulating the mean delay and the standard deviation, the statistical yield may be optimized.

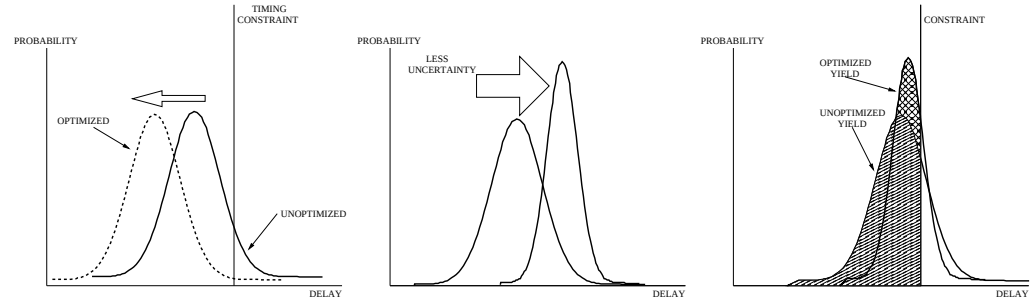
Statistical optimization can lead to serious implications in the type of constraints that must be used. Intuitively, shaping the distribution, which might be the result of a function between two, other, distributions, might mean that the two distributions must overlap in a specific way. This can mean that the means of the two distributions might need to be close, that one distribution is "inside" the other distribution, that the mean of one distribution is close to the edge of the other distribution, that the two distributions do not overlap at all, or any other complex constraint. The aforementioned example cases are depicted in Figure 2.4. It is not uncommon, that this will lead to upper and lower bounds on the mean and the sigma of one distribution or both. This makes statistical optimization harder than deterministic optimization (where usually there are only upper bounds) as two-sided constraints are introduced. This new property needs to be accounted for in any optimization framework, as algorithms which are optimized for one-sided constraints may not be efficient for two-sided constraints.

In the next section we turn our focus to asynchronous timing analysis, which is better suited for cyclic circuits.



(a) Means are close (b) Full overlap (c) Almost no overlap (d) No overlap

Figure 2.4: Cases of distributions' relative position



(a) Statistical optimization of delay (b) Statistical optimization of variance (c) Statistical optimization of yield

Figure 2.5: Statistical optimizations

2.1.4 Asynchronous Timing Analysis

Timing analysis of asynchronous circuits differs from STA, or SSTA for synchronous circuits both in methodology and in their goals. The goal of STA is to calculate the frequency at which the clock can be set. In asynchronous circuits, where the clock is absent, other metrics must be employed. One such metric is the time required for two successive occurrences of the same event. In a high level of abstraction, two events could be two full computations the circuit performs. For example, in the case of an adder the performance metric could be the time required for two successive additions. The state-of-the-art procedure for finding the time required for two events to happen is called timing separation of events. It essentially computes the minimum, or maximum, time required for two events to happen in succession, or the same event to happen twice.

2.1.5 Asynchronous Timing Analysis Models

The STA model is not suitable for asynchronous circuits due to the cycles in circuit representation. However, the concurrent behaviour of asynchronous circuits can be effectively modeled by concurrent representation models such as petri-nets [54]. A petri-net is a graph which consists of nodes and directed edges. Nodes can be either places or transitions. Places are conditions for transitions and are represented as cycles. Transitions are events that may occur and are represented by bars. Edges connect the transitions to places and vice-versa. An edge represents the conditions that are needed for a transition to occur. Additional elements, tokens, represent data values in places. Tokens can move through the graph and represent the way data flow through the circuit. For initialization, a marking is used. Marking consists of a number of tokens placed in the graph which signifies the initial state of the system. Figure 2.6 shows an example of a petri-net with an initial marking. The concurrent behaviour of the system is signified by the movement of tokens. All tokens can move independently of each other as soon as they are enabled. A token is allowed to move if all conditions are satisfied. In Figure 2.6, transitions `Z+` and `A-` are allowed to execute, as there may be tokens on all the edges leading to these transitions (all conditions are met). These transitions can execute at the same time, highlighting the concurrent nature of the system. `Z-` on the other hand is not allowed to execute, as there is a token on only one of its conditions.

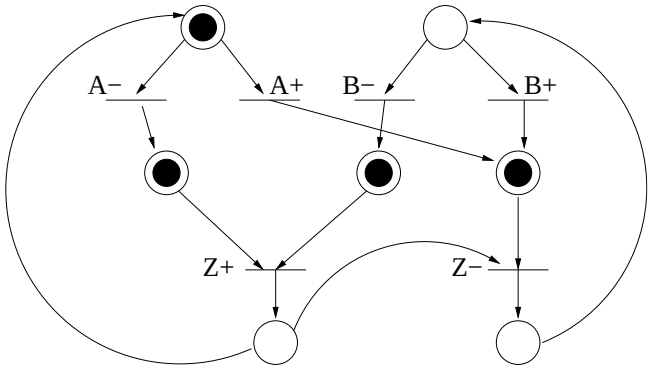


Figure 2.6: A simple petri-net with initial marking

For more efficient representation of asynchronous circuits, a subset of petri nets is used. This subset is called Signal Transition Graphs (STGs). The property that is omitted from the general petri-nets is that each place has exactly one input and one output transition, *i.e.* there is no choice-place. This represents the causality of an asynchronous circuit. The elimination

of choices means that the graphical representation of an STG is simpler than that of a general petri-net, *i.e.* places can be omitted. All the other properties of a petri-net hold. There are tokens signifying the data flow and for a transition to execute, all its conditions must be satisfied. Figure 2.7 shows an example of an STG.

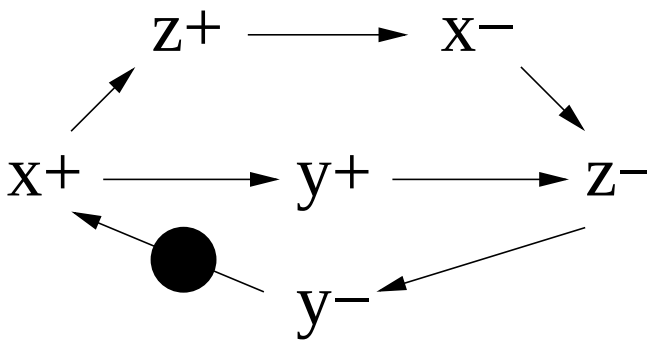


Figure 2.7: A *xyz*-STG

In the STG of Figure 2.7, only `x+` can execute at the initial state. This will create two tokens, one on the edge {`x+`,`z+`} and one on the edge {`x+`,`y+`}. The new state will enable the transitions `z+` and `y+` to execute independently. After `y+` executes, a token will appear on the edge {`y+`,`z-`}. `z-` however, cannot execute until a token has appeared on the edge {`x-`,`z-`}, which will happen after `x-` has executed. When `z-` is ready to execute, then `y-` will also be enabled arriving back to the initial state.

STGs can be enriched with timing values on the arcs to represent the delay of each execution. Typically, a minimum and a maximum delay, or a range of delays is inserted. The new representation forms an Event Rule (ER) system, which can be used for the timing analysis of the corresponding asynchronous circuit. The *xyz*-STG, which is now transformed into an *xyz*-ER is shown in Figure 2.8. On each edge there is a delay range $[d_i, D_i]$, where d_i represents the minimum delay for the corresponding transition and D_i represents the maximum delay. Event-rule representation is the basis for the state-of-the-art timing analysis framework of asynchronous circuits, called Timing Separation of Events (TSE).

2.1.5.1 Timing Separation of Events

Timing separation of events (TSE) starts from an event-rule representation and can answer questions like what is the minimum time of a full computation, or how late can event A can

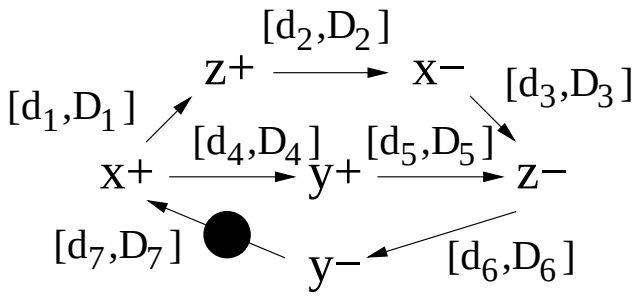


Figure 2.8: A *xyz*-ER

occur after event *B*. It does so by finding bounds on the minimum and maximum separation two events may have with respect to delay. The idea of TSE has been successfully incorporated into a number of timing optimizations such as synthesis and verification of asynchronous circuits [12], interface timing verification [83] and scheduling of concurrent systems [33].

An efficient algorithm for TSE has been presented by Hulgaard *et al.* in [33]. This algorithm can handle cyclic representations of concurrent systems. It works on an event-rule specification and extracts tight lower and upper bounds on the separation between events. The algorithm is based on the idea of process graph unfolding. An improvement over TSE has been presented by Kasapaki in [41]. In [41], TSE has been used to create an asynchronous timing analysis tool, which can operate in closed loop. The author of this work has showed that by an efficient implementation of TSE, critical cycles and critical gates of the asynchronous circuits can be identified. This information is the basis for optimizing for speed or area.

The event-rule specification used in [33] is a process graph. It is essentially an event-rule specification with the inclusion of a *root* node. The *root* node marks the point of the graph at which computation begins. No node may contain an edge *to* the *root* node. An example of a process graph is given in Figure 2.9.

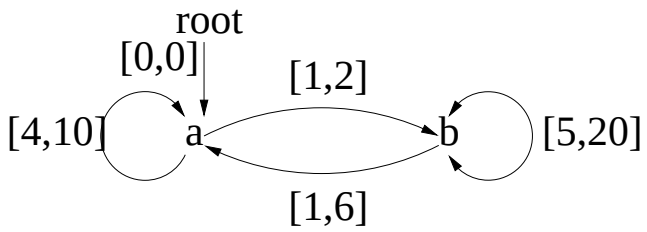


Figure 2.9: A process graph

In the process graph of Figure 2.9, any event can occur only after all the conditions for this event are met. This means that all the events leading to the event waiting to happen must also have happened. Given the maximum delay required for each event, the maximum delay for the occurrence of a pair of events can be found with unfolding the process graph. An unfolding of the process graph of Figure 2.9 is shown in Figure 2.10. The numbers on each node denote the maximum, absolute time at which this event can happen. The indices under the name of each event denote the occurrence index of this event. For example, a_2 denotes the second occurrence of event a. One can observe from Figure 2.10 that the timing separation between any two events fluctuates in the first few unfoldings, but converges later to a delay value. Hulgaard's algorithm unfolds the process graph as many times as needed until the timing separation between any two events converges to a final value. Additionally, methods to detect and prove convergence are also proposed.

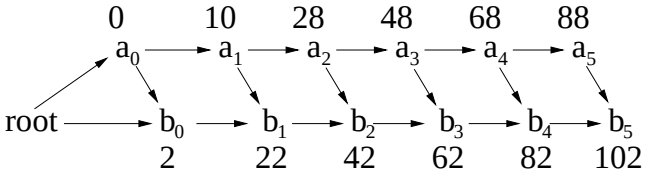


Figure 2.10: A process graph unfolding

Having presented our timing analysis framework, we present in the next section an overview of state-of-the-art placement algorithms with emphasis on timing-driven placers.

2.2 Placement Algorithms

Placement is the first step in the back-end flow, *i.e.* the physical steps required for the final fabrication of chips. It is the problem of finding optimal locations for all circuit elements on a fixed die and placing them accordingly. The best locations are defined by a cost function which can evaluate the quality of a given placement. Even in the case of standard cell placement with the simple objective of minimizing the interconnect length, the problem is known to be NP-hard [51]. On the other hand, placement is the cornerstone of the back-end flow, as it defines the framework on which the latter steps, like routing, will try to apply optimizations. Thus, it is imperative that placements are of good quality. Since after placement the degrees of freedom for optimization are seriously decreased, it is unlikely that a bad placement, negatively

affecting the high level performance metrics of the chip, *e.g.* speed, can be corrected at a latter step. This highlights the need for effective placers which can accurately evaluate the quality of their placements and drive the optimization towards efficient optimization.

In this section we present the placement problem. We then discuss their deficiencies and our approach to placement which leads to our placement tools, SCPlace and CPlace, described in Chapters 4 and 6.

2.2.1 Optimization Objectives and Constraints

Given a technology mapped netlist and adequate layout area, any placement algorithm will try to find the best placement possible. The quality of placement is defined by a set of constraints which are provided to the placer by the designer. The type of constraints can vary and can be either performance-oriented, design-rule-oriented or, most commonly, both. Performance constraints are imposed so that the resulting placement has desirable properties with respect to a performance metric, *e.g.* timing or power. Design-rule constraints ensure correctness of the placement with respect to its suitability for fabrication. We now present a list of common constraints that a placer must satisfy.

- **Wirelength minimization.** Wirelength has been the traditional metric for the performance of placers, both in academia and in industry, before the emergence of timing-driven placers. The objective is to minimize the total wirelength required for all wires connecting the cells. Since during the placement process routing is not available yet, wirelength is estimated using heuristics like the half-perimeter wirelength of the bounding box (BB HPWL). The bounding box referring to a single wire is the smallest box that contains all pins the wire is connected to. The hindsight for optimizing wirelength is that smaller wires will be easier to route.

- **Timing constraints.** Nowadays, most industrial placement algorithms are timing-driven in the sense that they actively perform timing analysis during placement and try to meet specific constraints on timing. Timing-driven placement algorithms typically try to borrow slack from cells which are not critical and give it to cells that are most critical or violate the timing constraint. If all cells are placed and no path violates the timing constraint, then the placement is considered successful. For the estimation of wire delays, wire delay models are used. The more accurate the wire delay models are, the more

accurate the timing analysis during placement will be. This, in turn, will make timing closure easier during the later stages of fabrication such as routing.

- **Power constraints.** Power constraints during placement usually fall into two different categories. The one is minimization of dynamic power and the other is minimization of leakage power. The former, depends on minimizing the capacitance of wires which switch most. This usually means that the length of these wires must be minimized. A placement algorithm under this type of constraint must bound the distance between cells belonging to the critical net in order for the net's capacitance to be minimized. Minimizing leakage consumption (or essentially leakage current) means changing the standard cell itself. This is accomplished by replacing the standard cell with a smaller one that performs the same function as the initial cell but exhibits less leakage current at the cost of a degradation in speed. A placement algorithm with timing and leakage constraints may choose appropriate cells for substitution, using the set of alternative cells from the technology library in order to co-optimize for timing and leakage.

- **Fixed cells and blockages.** It is not uncommon for a placement algorithm to be required to place specific cells or larger macros in specific locations. These cannot be moved throughout the optimization process and must be treated as fixed blockages, as no other cell may be placed on the locations they occupy.

- **Allowable regions.** After all fixed cells have been placed, the shape of the allowable region for all other cells is revealed. This shape is not always rectangular and must be identified by the placement algorithm so that it is efficiently utilized.

- **Density constraints.** The layout area given to the placer is usually much larger than the total area required by standard cells. There is typically no less than 35% of free space in order to avoid over-congestion. The placer may face strict constraints for congestion which could instruct the placement algorithm to limit the local congestion over a small portion of the layout area to, *e.g* 70% and the total congestion to, *e.g* 65%. In this case, the placer may face artificial blockages caused by congestion issues, a fact which aggravates the complexity of the problem of placement.

- **Legal locations.** The placer must place all cells in locations which are considered legal. Legality is defined by design rules which cannot be violated, otherwise the chip cannot

be fabricated. One of the most important constraints is that all cells must be placed on locations which cause no overlaps. Other second-order common constraints is that all cells must be aligned to a predefined grid and must have a suitable orientation.

2.2.2 Requirements for a Placer

Regardless of the actual algorithms a placer employs, a set of requirements is needed for cell placement. These are mentioned below.

- **A technology mapped netlist.** The circuit to be placed must have been mapped into a specific technology, which provides standard cells for all gates of the circuit. High-level descriptions or hardware-description language representations are not enough for a placement algorithm.
- **A set of constraints or objective functions.** The placer needs a set of directives in order to place the cells into locations which are optimal in the way the designer wants. The set of constraints may be as simple as wirelength minimization, or more complicated such as joint timing and power optimization. Moreover, constraints on local congestion may also be present in order to enhance routability.
- **A strictly defined layout area.** Cell placement is performed on a predefined layout area which allows for placement of all cells and accounts for routing, which will be done at a later fabrication stage. The layout is divided into rows, the height of which equals that of a standard cell. Each row can also be divided into several segments by the manufacturing grid, which can be envisioned as a set of dense vertical lines, which in conjunction with the rows, form small rectangles, called "sites" in each row, as shown in Figure 2.11. A cell must be aligned to the manufacturing grid, which means that its leftmost side must be aligned with the leftmost boundary of a site. Any standard cell can span multiple sites in the same row and any larger block may span multiple rows and sites.
- **Characterization of cells.** This includes characterization for both the dimensions and the timing/power performance of each cell. It is obvious that the placer must have knowledge of the exact dimensions of each cell in order for it to be placed on a legal location. The characterization for timing and power is critical for the placer to be able to compute the cost functions it employs throughout the optimization process.

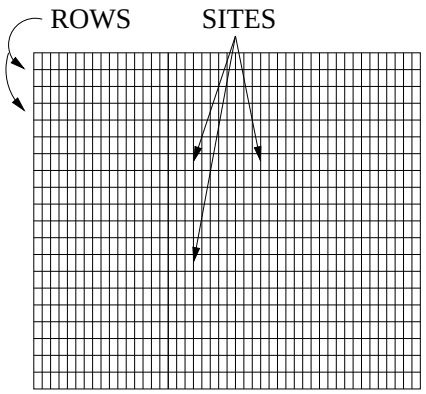


Figure 2.11: Layout manufacturing grid

- **Wire delay modeling.** Although no exact estimation of wire delay is possible before routing, the placement algorithm, especially in the case of timing-driven placement, must utilize an approximation model for wire delays.

- **In case of industry-standard tools, *DEF* file support.** *DEF* stands for Design Exchange Format [50] and is the industry standard for placement description. It contains the physical locations for all cells on the layout area. It also contains information about the layout itself, such as the number of rows, the width of each row and the manufacturing grid on which each cell must be aligned.

2.3 Algorithmic Approaches to Placement

The problem of placement has been well studied since the appearance of the first EDA tools. Thus, a number of different algorithmic approaches have been proposed and tried in practice. Before we describe specific algorithms in detail, we provide a fair description of the general algorithmic framework. We distinguish between iterative *vs* constructive and global *vs* detailed placement. We also describe the differences between standard cell and mixed-size placement. Later, we offer a taxonomy of contemporary placers which can be categorized as stochastic, partitioning-based and analytical.

2.3.1 Iterative vs Constructive Placement

Iterative placers typically start from an initial placement, which evaluates poorly with respect to the cost function and then iteratively correct the placement guided by the cost function. Constructive placers typically place a few seed cells on the layout and then place the remaining cells on locations which meet their constraints.

Pure constructive placers typically exhibit inferior results compared to their iterative counterparts, as most of their local algorithmic decisions are made with an incomplete global view. For instance, a local, random choice of seed cell may determine the quality of a complete placement. Min-cut and partition-based placers address this deficiency and improve on the local decision making process through global knowledge and repairing techniques such as re-clustering. Another characteristic deficiency of constructive placers is their tendency, due to their construction process, to produce layouts with mixed sections of densely and sparsely populated areas respectively, *i.e.* significant congestion differences, which can lead to routability problems.

However, constructive placers are characterized by their correct-by-construction approach. Every time a cell is placed on a location, it may remain there without the need to be corrected in a future optimization step. This is true, since the location of every cell is determined using the information of the already placed cells with which it shares a constraint. Thus, constructive placers are especially suitable when conflicting constraints are in place. Such types of constraints may over-constrain an iterative placer and force it to oscillate between solutions which violate one or the other constraint. Constructive placers on the other hand, will first determine the best location, which satisfies all constraints, if this is possible, and then place the cell. Successful constructive placers need to make sure that they can keep placing cells as long as there are unplaced cells. Cell placement can be stopped due to unfortunate placement of "seed" cells which may cause unplaced cells to be over-constrained. The challenge for a constructive placer is to employ efficient heuristics which escape from local minima of this kind.

2.3.2 Global vs Detailed Placement

A number of placement algorithms distinguish between global and detailed placement as separate problems of placement, which they address of in different optimization loops. Global placement can be best described in the case of a partitioning-based placer. This kind of placer, first finds an optimal segmentation of the circuit using a number of heuristics. Each segment

is then assigned to a region of the layout. Global placement finishes by finding an optimal matching between the list of layout regions and the circuit segments, without placing the cells into actual locations. Detailed placement then, operates on each segment by placing each cell on locations which satisfy the placement design rules. These rules can dictate that there can be no overlaps among cells and that cells must be aligned on rows or on a manufacturing grid. The division of the placement problem into global and detailed can also be found in other algorithmic approaches, such as stochastic or analytic placers. In this case, global placement refers to placing cells in much greater granularity than detailed placement and not paying too much attention to satisfying overlaps and design rules. In any case, care must be taken that detailed placement is allowed enough freedom to actually be able to arrange all cells in suitable locations.

2.3.3 Standard Cell Placement vs Mixed-size placement

Modern circuits may not comprise solely of standard cells but of a mix containing standard cells and blocks of varying size. Blocks may be "hard" or "soft". Hard blocks have fixed size and dimensions which cannot be altered in any way. They can only be treated as large standard cells spanning multiple rows and having relatively large widths. Characteristic examples of hard blocks are *RAM* cells. Soft blocks on the other hand, often refer to logic blocks consisting of several standard cells. The area of a soft block is known, but its dimensions can alter, the only constraint often being on a minimum/maximum aspect ratio. The shape of a soft block may even not be strictly rectangular allowing for more flexibility during the placement process.

If the design contains mixed-size cells, then the task of placing all cells is often called floorplanning. This is a different problem from placement which employs different metrics for performance. In modern circuits, the number of hard/soft blocks usually does not exceed few hundreds which is significantly smaller than the number of standard cells that can easily scale beyond few millions. Algorithms for floorpacking or block-packing typically try to minimize the area required for all blocks, or to pack the blocks in such a way such that the unused area among the blocks is minimized. Two very successful algorithms for block packing are *Sequence pair* [53] and B^**-tree* [14]. *Sequence pair* maintains two ordered lists which describe the geometric relations among blocks. These relations stem from the relative ordering of the blocks. The objective of *sequence pair* is to find the optimal ordering. The solution space is often explored with stochastic algorithms like simulated annealing. B^**-tree* on the other hand

is a binary tree representation of block-packing. The objective of B^*-*tree* is to balance the tree in such a way such that the nodes are most densely packed. This will result in minimizing the area for floorplanning all the blocks.

The problems of placement and floorplanning have recently being merged into an approach called *floorplacement* [67]. *Floorplacement* initially considers all cells, standard cells and blocks alike. Min-cut is employed until the segments contain a number of blocks for which block-packing can be applied. Block-packing is employed using traditional floorplanning methods. After blocks have been placed, then only standard cells are left to be placed. These are placed on the empty areas of the layout treating the placed blocks as obstacles. However, care must be taken so that the initial placement of blocks does not over-constrain the placement of standard cells.

2.3.4 Wire Bounds as an Optimization Directive

Any placer will, at some point, need to make a local decision as to where to place a specific cell. In iterative approaches this action is typically performed by swapping two cells. In constructive approaches, the location is selected according to the cell's constraints. In both cases, the action of placing a cell, or changing its location, has implications on the expected length of the wires connecting this cell to its neighbours. The decision-making process of the placement algorithm, for the placement of a cell, must include a check on the expectation of these wire lengths. In order to simplify this process, wire bounds are extensively used in this situation. A wire bound is usually an upper bound (although it can be a lower bound) on the allowable delay of each wire. These bounds are usually derived by performing timing analysis on the circuit, deriving the slack for each path and assigning slacks to wires. After slacks are known, bounds on the wire lengths can be directly inferred. A placement algorithm can then check if a candidate location for a cell creates any wire which violates its bound. If this is the case, then this location can be rejected, otherwise, the candidate location can be considered for optimization.

2.3.5 Taxonomy of Placers

In this section we offer a taxonomy of state-of-the-art and earlier placers, based on their general framework and their optimization approach.

- **Stochastic.** Stochastic placers are of iterative nature and use heuristics to optimize the placement at each iteration. The most widely adopted category is placers which use simulated annealing. Another category of stochastic approaches consists of genetic approaches. A detailed description of these approaches is given in Appendix B.

- **Min-Cut.** A different category of placement algorithms consists of approaches which recursively divide the netlist and the layout area. These are based on the idea of "divide and conquer". They divide the problem of full placement into smaller placement problems which can be solved more efficiently and then combine the solutions of the smaller, simpler problems into a full placement solution. Min-cut algorithms divide the netlist into segments and try to minimize the number of nets that span among segments. A detailed description of these approaches is given in Appendix B.

- **Analytical.** A different class of placement algorithms use analytical methods to mathematically optimize an analytically expressed cost function. They can be further divided into non-linear and quadratic algorithms, depending on the type of the cost function they employ. A detailed description of these approaches is given in Appendix B.

- **Constructive.** The main idea behind every constructive-based approach is to progressively place all standard cells on the layout on locations which satisfy all the constraints of the standard cell.

2.3.6 Challenges for Contemporary Placers

As technology advances, a number of assumptions employed by contemporary placers are becoming out of date. In this section we describe issues that any placer must overcome before it becomes obsolete in view of advancing technology.

2.3.6.1 Multi-Corner Placement

In a typical industrial flow, different technology libraries, which define the characteristics of standard cells under different operating conditions, are used. These include estimations of the best case and worst case scenarios for temperature, voltage and process fluctuations. Thus, a number of corners is available to the designer, each of them describing a different scenario under which the chip should not violate its constraints.

In order to guarantee correct operation at all times, the placement tool must perform placement at all corners. However, optimizing for one corner will usually cause a violation at another corner posing the risk of oscillating between solutions which are optimal for a few corners, but violating for other corners. Nowadays, optimizing for all corners is done manually using trial-and-error approaches. The designer runs the optimization at one corner and allows for some slack on all constraints with the hope that no violations will be reported on other corners. This can be a lengthy process which may also cause the final placement not being optimal for any corner.

A way to mitigate this problem could be a unification of all corners into a single model. Placing the design using the unified model could enable simultaneous optimization for all corners, thus eliminating the need for multiple runs on different corners. One possible model encapsulating the characteristics of all corners is a statistical model. In this case, optimization should be done statistically and the type of constraints should also be statistical.

2.3.6.2 Asynchronous Circuits Placement

Asynchronous circuits is the design-level solution to the variation problem due to their ability to adapt to the environmental and process conditions. An EDA placement tool, capable of implementing asynchronous designs, (or both synchronous and asynchronous designs) must have additional capabilities compared to conventional EDA tools. First, some form of asynchronous timing analysis must be supported, whereby the critical delay cycle, rather than the critical path will be taken into account for optimization. Second, timing constraints will be required to be potentially relative, *i.e.* between internal circuit signals, instead of absolute, relative to one or more global clocks. Specifically, a placer should be able to satisfy both absolute timing constraints, *i.e.* a maximum or minimum delay bound for a given circuit portion, and relative timing constraints, *i.e.* relative delays between signals. The latter type of constraints may satisfy Quasi-Delay Insensitivity (QDI) requirements, *i.e.* isochronic forks, with a given allowed delay margin, or Speed-Independent (SI) requirements, *i.e.* wire delay contribution being a small percentage compared to gate delay contribution. Third, as the period of the asynchronous circuit will depend on its critical cycle(s), as identified by asynchronous timing analysis, a slack assignment strategy is required to account for delay distribution to internal circuit wires. These additional capabilities may be combined with traditional non-timing driven placement heuristics, such as total wirelength.

2.3.6.3 Post-placement Optimization

During the fabrication of a contemporary chip, rarely it is the case that only one performance metric is of interest to the designer. That is, the designer will most frequently want to optimize not only for speed, but also for power and area. It is not unusual that one of the metrics have priority over the others. This enables the designer to optimize for the primary metric during the placement process and then run a cleanup process to optimize for the other metrics without affecting the result of the finalized placement. This process is called post-placement optimization and usually consists of small tweaks on the placement. The challenge for the placer is to allow enough room in the solution space for this optimization to take place. This means, that the final placement should not be exactly on the edge of failing the primary constraint in case even a small change is made on the placement. Thus, placers must allow for some room for post-placement optimization, given the constraints that that this particular optimization intends to use.

Contemporary placement tools use timing as the primary objective and power as a secondary one. One of the most important sources of power consumption is leakage power, which depends only on the circuit itself and not on how fast the circuit operates. Leakage power is consumed due to leakage current in transistors even when they do not perform any computation. Thus, it is crucial that the circuit is fabricated using gates which are less prone to leaking current, and will thus consume less leakage power.

Leakage current has three main sources. The first, is source/drain junction leakage current due to the appearance of reverse-biased nodes in an *OFF* transistor. The second, gate tunneling leakage is due to current flowing to the substrate through the oxide insulation. The third, and most important is subthreshold current, which is due to current flowing through an *OFF* transistor that is in the subthreshold region. Subthreshold region is becoming more prominent in contemporary and future technologies, where the threshold voltage continuously decreases, minimizing the gap between the *OFF* and the *ON* voltage levels of a transistor.

Leakage due to subthreshold voltage is expected to become even more important in overall power dissipation estimation of a circuit in future technologies. The reason is its exponential dependence on threshold voltage, which is continuously decreasing. This is illustrated by Equation 2.1,where K, η and n depend on technology, V_{DS} and V_{GS} are the *ON* and *OFF* levels and V_T is the threshold voltage. It is clear from Equation 2.1 that decreasing V_T, there must

be expected an exponential increase in leakage current.

$$I_{DS} = K\left(1 - e^{(\frac{-V_{DS}}{V_T})}\right) e^{\left(\frac{-V_{GS}-V_T+\eta V_{DS}}{nV_T}\right)} \tag{2.1}$$

Threshold voltage is especially prone to variations, both in process and in operating conditions. Process variations can directly affect the threshold voltage by fluctuations in oxide thickness, which is a very common source of variation in contemporary fabrication flows. Temperature variations on the other hand, is the main source of fluctuations in threshold voltage. Typically, technology vendors provide characterizations for leakage current under a number of different operating scenaria (corners) as is the case in timing analysis. Thus, designers need to perform multi-corner analysis in order to make sure that the design meets the constraints on the maximum allowable leakage current. One way to overcome the need for multi-corner analysis is to model the information from all corners into statistical distributions and perform statistical leakage optimization. However, although there have been proposed a number of statistical modeling approaches for leakage, there is still lack of fully automated large-scale statistical leakage optimization flows.

Based on the aforementioned challenges, we now present the limitations of contemporary placement algorithms and tools.

2.3.7 Limitations of Contemporary Placers

Current state-of-the-art placers focus on either optimizing the total wirelength of a circuit, or meeting the clock period timing constraints. These goals are generally one-sided, *i.e.* improvement during a step of the optimization process may be evaluated by direct comparison with an absolute value. For example, in the case of timing driven placement, if the delay of the critical path is decreased, while the delay of no other path is increased, then the new placement is considered an improvement over the previous one. Limitations of current state-of-the-art placers become evident when they must deal with two-sided constraints. This is the case for both placement for statistical-based optimization and placement of asynchronous circuits.

In the case of statistical-based optimization, the metric that needs to be optimized consists of two values, *i.e.* the mean and the sigma. Optimizing for one value, often requires some sort of sacrifice for the other value. When the optimization is made with the use of constraints, the constraints leading to optimization of one value generally contradict the constraints for the other

value. This property makes the general framework of contemporary placement optimization tools to fail, as it is not suited for double-sided constraints. In fact it is prone to make the placer oscillate between two solutions, optimizing in one iteration one value, *i.e.* the mean and on the next iteration the other value, *i.e.* the sigma.

Furthermore constraints for statistical optimization can be relative, rather than absolute. This is a serious problem even for the most successful placers known to academia and industry, as they are developed for use with absolute constraints. Typical examples of absolute constraints is that total wirelength must not exceed a maximum bound, the slowest path must have a maximum delay and the total power consumption must be below a certain limit. Statistical optimization on the other hand, may require certain paths to have a minimum delay or the difference in delay between two paths be bounded.

In the case of asynchronous placement, two-sided constraints may generally need to be met. Certain wires, *i.e.* legs of isochronic forks, will require both a minimum and a maximum allowable delay constraint. Such bounds may not be known a priori, as they are relative to other wires.

Furthermore, conventional STA engines used during timing-driven placement assume that the circuit is acyclic. When cycles are encountered during STA, STA engines will typically break them arbitrarily and analyze the resultant acyclic timing graph. Asynchronous control circuits are cyclic circuits, therefore cannot be effectively analyzed using STA, or timing-driven placed by existing synchronous placers with a given performance goal such as asynchronous period. Thus, an efficient placer for asynchronous circuits must incorporate a timing analysis engine which can handle cycles, like TSE, and must also be able to tackle the contradicting targets created by the relative and two-sided constraints.

2.3.8 Our Approach to the Placement Problem

Having identified the limitations of contemporary placers, we have tackled these specific problems by developing our physical placement and post-placement flows.

2.3.8.1 Statistical/Multi-Corner Timing-Driven Placement

In order to combat the problem of multi-corner placement, we propose the use of a statistical approach. We model the timing characteristics of gates across all available corners as normal distributions and apply optimizations through the use of SSTA, statistical slack assignment and

our statistical placement tool, *SCPlace*. SCPlace is the first large-scale industry-compatible statistical optimization placement tool.

2.3.8.2 Placement for Asynchronous Circuits

We have tackled the problem of satisfying timing assumptions during the placement of asynchronous circuits by developing a novel slack assignment procedure which we employ in our placement tool, *CPlace*. CPlace is the first placer known to literature which can create performance-efficient placements of asynchronous circuits without violating their timing assumptions.

2.3.8.3 Post-Placement Optimization

We have addressed the problem of post-placement optimization by developing a post-placement cleanup algorithm, which minimizes leakage without affecting the statistical delay of the circuit. We have targeted synchronous circuits only, as this allowed for integration of our SSTA engine with our statistical leakage analysis process. Our leakage optimization flow is the first flow which can guarantee the initial statistical delay.

Having concluded the overview of state-of-the-art placement algorithms, their limitations and our approach to handle the problems that are starting to emerge, we now present the first contribution of this thesis, which is the derivation of bounds for statistical optimization.

Chapter 3

Statistical Delay Bounds

In this chapter we describe our methodology for deriving statistical delay bounds. We derive bounds for propagating appropriate delay distributions across the timing graph, so that the statistical delay of endpoints meets statistical constraints.

The motivation for developing algorithms of such type is the recent emergence of statistical methods to account for circuit delay and the lack of large scale statistical optimization tools. Traditional slack allocation algorithms derive maximum slacks on wires to account for physical wire delay and to guide the physical optimization process. We derive bounds which control the statistical delay for use in a statistical physical optimization algorithm.

In the next sections we present our algorithms for deriving statistical bounds. First, we demonstrate our methodology for SSTA. Then, we show our Minimum Sigma Slack Assignment (MSSA) algorithm which derives bounds for propagating delay distributions with minimum sigma to the endpoints. Next, we present our Target Sigma Zero Slack Assignment (TSZSA) algorithm, which derives bounds for propagating delay distributions that meet a target on the mean and a target on the sigma for the delay of the endpoints.

3.1 Statistical Static Timing Analysis

In this section we extend STA into our statistical timing analysis engine. First, we describe our model for gate delay and then we show how delays are propagated across the circuit as timing analysis progresses.

3.1.1 Statistical Gate Delay

Any statistical timing analysis engine must possess the notion of the statistical delay of a single gate. The assumption that the delay distribution of a single gate follows a normal (Gaussian) distribution is widely adopted in literature and has been proven consistent by experimental results on fabricated chips. Thus, we use the same assumption in our SSTA framework.

Unlike STA, no technology descriptions for the statistical delay of circuit elements is available to us. On the contrary, we possess a number of technology descriptions for different operating corners. A statistical model should encapsulate the behaviour of the elements under different operating corners. Thus, we infer statistical delay distributions from the given technology corners for every circuit element. Since we assume normal distributions, two values are necessary for their exact description. The first is the mean value and the second is the standard deviation (sigma). In normal distributions, the mean value is also the expected value, which is sometimes referred to as the typical value. Thus, we extract delays from the *typical* technology library and assume that these values are the mean (μ) values for each elements' delay distributions. For the standard deviation (σ), we use the typical corner and the worst-case corner. As explained in Appendix C.1, 99.7% of the normal distribution's samples lie within ($\mu \pm 3 * \sigma$). Thus, the worst-case corner should lie on ($\mu + 3 * \sigma$). We calculate the standard deviation as $\sigma = \frac{\mu_{wc} - \mu_{typ}}{3}$, where μ_{wc} is the delay at worst-case and μ_{typ} is the delay at typical case.

Having established the delay model for a single gate, we now describe the propagation of delays across the circuit.

3.1.2 Statistical Delay Propagation

SSTA aims at calculating the arrival time at the circuit's endpoints. In order to do so, we first construct the timing graph of the circuit. This is a direct mapping of the circuit, where the nodes of the graph are the gates' ports and the circuits' ports. Then, we assign any initial delay distributions to the startpoints, which are the primary inputs and the outputs of sequential elements, as in the case of static timing analysis. Next, we traverse the timing graph in BFS until we reach the endpoints, which are the primary outputs and the sequential elements' inputs. In order to assign delay distributions to all circuit's endpoints, propagation of normal delay distributions is needed across the timing graph.

Thus, two basic operations are needed. The first is the SUM operation and the second is the MAX operation. Although these operations are straightforward in STA, in SSTA additional

computations are needed.

Since we assume that the delay distributions follow the normal distribution, the SUM operation does not pose any difficulties. The sum of two normal distributions is known to follow a normal distribution [64]. Thus, for two normal distributions X and Y with $X \sim (\mu_X, \sigma_X^2)$ and $Y \sim (\mu_Y, \sigma_Y^2)$. their sum Z with $Z \sim (\mu_Z, \sigma_Z^2)$ is given by:

$$Z = SUM(X, Y) = X + Y \tag{3.1}$$

$$\mu_Z = \mu_X + \mu_Y \tag{3.2}$$

$$\sigma_Z^2 = \sigma_X^2 + \sigma_Y^2 + 2cov(X, Y), \tag{3.3}$$

where $cov(X, Y)$ is the covariance between the normal distributions X and Y. Covariance can be found through the correlation $\rho(X, Y)$by $cov(X, Y) = \rho(X, Y)\sigma_X\sigma_Y$.

Calculation of MAX, however, is not straightforward. In the general case, the MAX of two normal distributions does not follow a normal distribution. However, it has been found [15] that the MAX can be reasonably approximated by a normal distribution using the following procedure:

$$Z = MAX(X, Y) \tag{3.4}$$

$$\mu_Z = \mu_X\Phi(\alpha) + \mu_Y\Phi(-\alpha) + \beta\phi(\alpha) \tag{3.5}$$

$$\sigma_Z^2 = (\mu_X^2 + \sigma_X^2)\Phi(\alpha) + (\mu_Y^2 + \sigma_Y^2)\Phi(-\alpha) + (\mu_X + \mu_Y)\beta\phi(\alpha) - \mu_Z^2 \tag{3.6}$$

where

$$\alpha = (\mu_X - \mu_Y)/\beta \tag{3.7}$$

$$\beta^2 = \sigma_X^2 + \sigma_Y^2 - 2\sigma_X\sigma_Y\rho \tag{3.8}$$

where Φ is the cumulative density function (CDF) and ϕ is the probability density function (pdf) of a normal distribution with mean 0 and standard deviation 1. We use the aforementioned formulas for the calculation of SUM and MAX. This allows for propagation of normal distributions across the timing graph and to the endpoints.

One subtle detail lies in the operation of SUM and MAX in case the operands are more than two. This case is straightforward in STA, as addition is associative and finding the maximum of a set of real numbers does not depend on the order at which the numbers are examined.

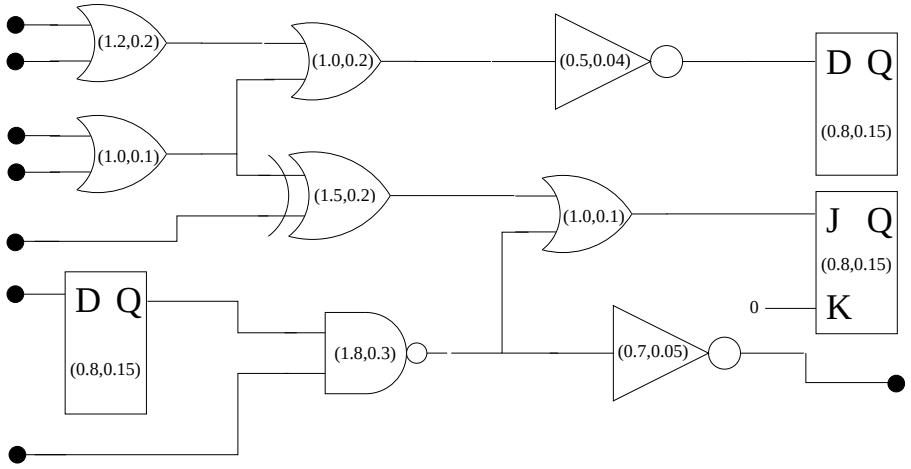
However, working with distributions, these operations may depend on the relative order of the operands.

We tackle the problem of finding the *SUM* of more than two normal distributions by eliminating the need for this particular calculation. The only point at which the *SUM* operation is needed is at the calculation of the delay of a gate output, related to the delay of a particular input of this gate. This operation requires two operands; the delay of the input and the propagation delay of the gate. Thus, we do not need to calculate sums of more than two normal distributions.

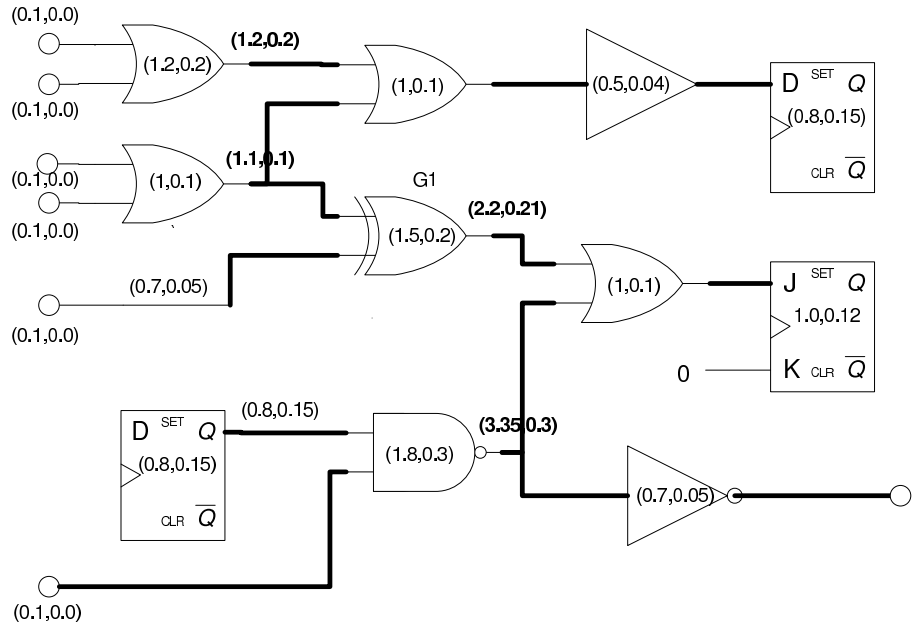
However, in the case of *MAX*, we may have a set of normal distributions from which the *MAX* must be derived. This is the case of a gate having more than two inputs. The *MAX* of the inputs' delay must be calculated in order to be combined with the gate's propagation delay, in order to derive the final delay at the gate's output. Additionally, the *MAX* of all circuit's endpoints is required for the derivation of the circuit's delay. The calculation of the *MAX* over a set of normal distributions must be done in pairs, as the aforementioned approximation does not extend to more than two distributions. This can introduce errors, especially if the distributions overlap. Sinha *et. al* [72] quantified the error stemming from different approaches in the order the pairs are selected from the set of distributions. Their results showed that reasonable accuracy is expected if the normal distributions are sorted with decreasing mean and the the *MAX* done in pairs, choosing first the distributions with the largest mean. The error of this approach was shown to be less than 1% compared to Monte-Carlo simulations. Thus, we have adopted this approach in our framework.

Figure 3.1 shows an example of a circuit on which SSTA is performed. On each timing node (gate input/output or primary input/output) the statistical delay is annotated in terms of the mean and the sigma of delay. In Figures 3.2a to 3.1d, the delays which have just been updated are shown in bold. First, the statistical arrival times at all startpoints are annotated, as shown in Figure 3.2a. Then, BFS starts by updating the statistical delay at the gates belonging to the first level after the startpoints (Figure 3.2b). BFS progresses then, as shown in Figure 3.1c towards the endpoints, passing through the second level of gates. Note that in Figures 3.2b and 3.1c, gate *G1* has been updated twice, as in its first update, not all of its inputs had taken their final values. In Figure 3.1d, SSTA has updated all the timing nodes, and thus the whole circuit has been statistically analyzed.

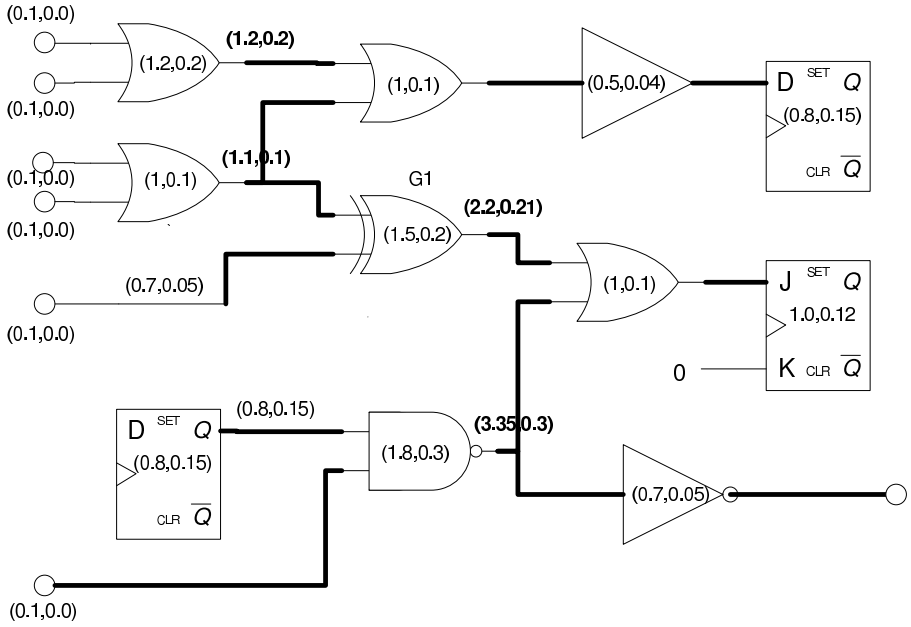
Our SSTA engine results were validated against Monte-Carlo simulations. Monte Carlo results are illustrated in Figure 3.2, which compares an SSTA result with Monte Carlo sim-



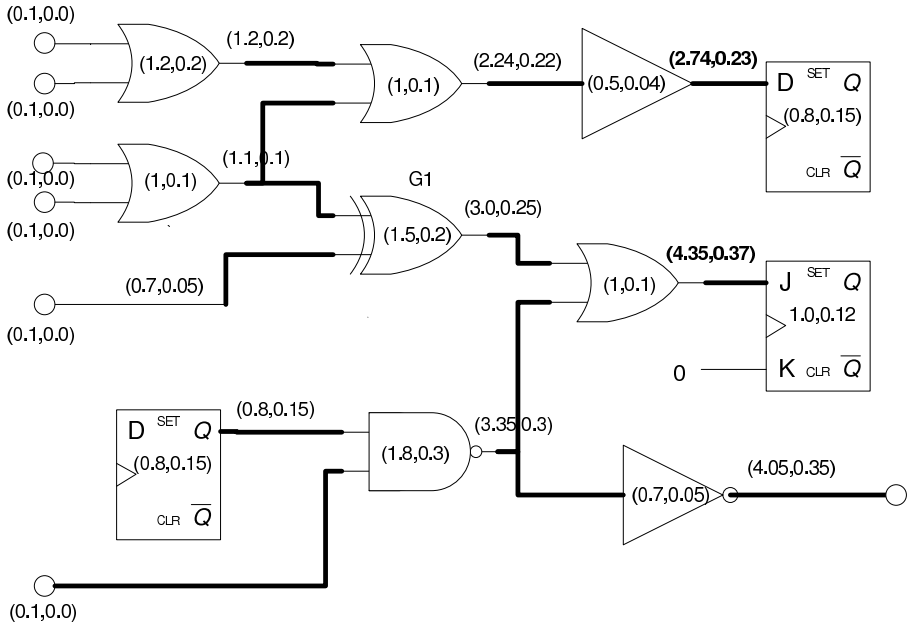
(a) Statistical arrival times



(b) First BFS level



(c) Second BFS level



(d) SSTA converges

Figure 3.1: Progression of SSTA

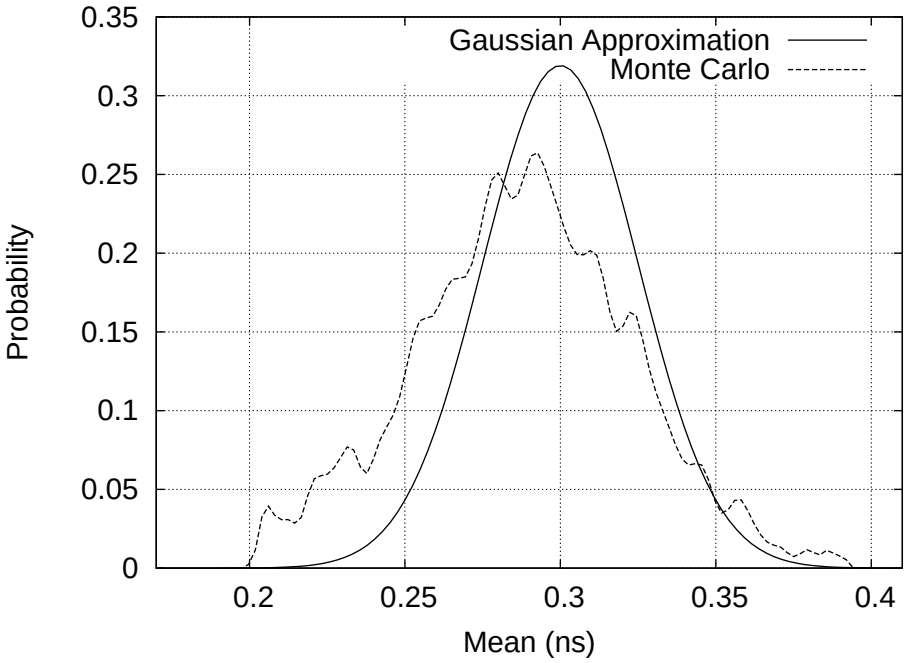


Figure 3.2: Monte Carlo analysis compared with SSTA result

ulation for 1,000 samples. The results were taken for a small circuit of 64 gates, which is part of the benchmark set we use later for evaluation. We used the following methodology for Monte-Carlo simulation. The startpoints of the circuit were assigned a delay distribution to account for arrival time. This was then translated into a set of 1,000 samples which represents the distribution in numerical fashion. Then, timing analysis of the circuit was performed in BFS. Every time a *SUM* operation was needed, the two sample sets of the two operands were added, yielding a sample set for the result. In case of a *MAX* operation, 1,000 samples were selected randomly from the sample sets of the two operands. This was done by selecting a sample from each set and outputting the larger of the two in the set corresponding to the result of the *MAX* operation. At the endpoints, we performed a *MAX* operation to yield the sample set corresponding to the final timing of the circuit. Figure 3.2 shows the distribution of the samples from the simulation-derived sample set and the delay distribution derived from our SSTA using the same assumptions about the input arrival times. Our tool captures efficiently the worst-case point of the delay and correlates well with the distribution exhibited by the simulation results. The results shown are for one of our benchmark circuits, `b06`, however other circuits exhibit similar results.

3.2 Minimum Sigma Propagation

In this section we describe our procedure for deriving the lowest sigma bound as a function of wire delays for a gate-level circuit and detail the relevant algorithm. Our goal is to derive, given a circuit and its SSTA model, a delay assignment at the circuit's wires which achieves the minimum sigma, at the virtual sink node, n_f. We call this, the Min-Sigma Slack Assignment (MSSA). Similarly to the Zero-Slack Assignment (ZSA) algorithm [56], delay assignment refers to deriving a required delay bound for a given wire, which can then be converted into a wirelength bound for physical design algorithms.

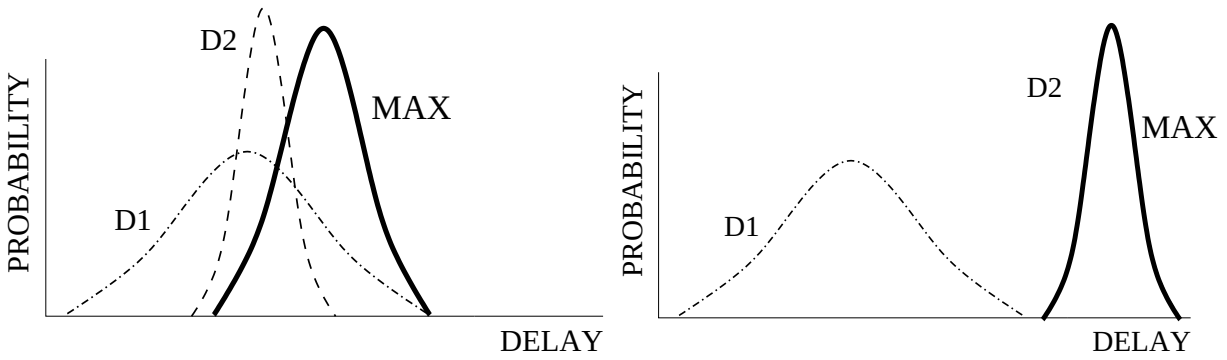
3.2.1 Motivation and Intuition for MSSA

MSSA provides, by definition, the distribution with minimum sigma that can appear to the endpoints of the circuit and to any internal node. An optimization tool for sigma can use MSSA in order to decide how hard the sigma constraints are. MSSA results can also signal the infeasibility of a sigma constraint.

MSSA aims at propagating delay distributions with minimum sigma at all circuit's endpoints. In order to do so, two questions must be answered. The first is how to shape the delay distribution of a gate's output so as the sigma of its delay distribution is minimized. The second is how this procedure can be generalized to circuit granularity, *i.e.* how to minimize the sigma at the circuit's endpoints.

We answer the first question by examining the way delay distributions are calculated at any gate output. There are two steps for this calculation. The first step is to compute the *MAX* of the delays of all gate's inputs and the second step is to add the propagation delay of the gate. We cannot intervene in the second step, as the propagation delay is a value extracted by the technology library. However, the result of the *MAX* operation can be manipulated. In fact, we employ the idea that if two delay distributions do not overlap, then the *MAX* result depends only on the "dominating" distribution, *i.e.* the distribution whose samples are always larger than the samples of the other distribution. This is graphically depicted in Figure 3.3. Thus, by applying delay to the distribution with the minimum sigma, we can minimize the sigma of the *MAX*.

We answer the second question by the observation that minimizing sigma of the delay of one gate, cannot cause an increase in the sigma of another gate. Moreover, minimizing sigmas at one logic level of the circuit cannot cause an increase in sigma in a latter logic level. Thus,



(a) MAX depending on both distributions (b) MAX dominated by one distribution

Figure 3.3: MAX of two normal distributions

by finding the minimum sigma for every single gate of the circuit, then the minimum sigma will also appear at the endpoints.

In the next sections we describe in detail our MSSA algorithm.

3.2.2 MSSA for a Single Gate

Our SSTA delay model focuses on gate delays, *i.e.* we assume that wires affect the gate load and delay, but do not themselves possess a sigma contribution. Based on this delay model, and given a gate with statistical delays at each of its inputs, we pose the question of how to propagate a statistical delay with minimum sigma at this gate's output, given that we can skew the Arrival Time (AT) of its input delays by introducing wire delay.

Given that we can identify the input distribution with minimum sigma, and that the statistical *ADD* operation performed at the gate's input pin can only monotonically increase sigma, one obvious way in which to propagate the distribution with minimum sigma is to sufficiently delay the input pin of the former with enough wire delay so that it doesn't overlap during the statistical *MAX* operation performed at the gate's output pin.

Figure 3.4 illustrates a 3-input *NAND* gate example, where the distribution at node 2, *i.e.* the minimum sigma node, with the addition of the appropriate wire delay, results in the narrowest sigma at the output X. However, there is still the question of finding the minimum delay offset which will yield the minimum sigma.

Answering this question is not easy due to the mathematical complexity of the Gaussian approximations for the statistical *MAX* (or *MIN*) operators [16]. To the best of our knowledge,

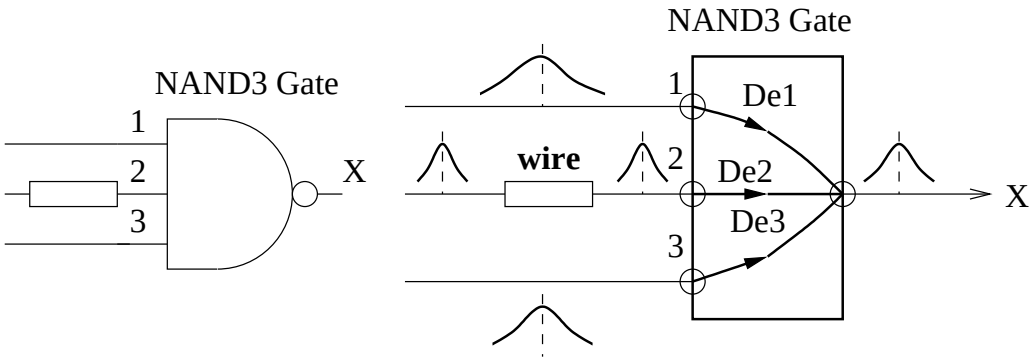


Figure 3.4: 3-input gate MSSA delay assignment example

the mathematical expression for deriving the standard deviation of the statistical MAX operation cannot be expressed analytically due to the presence of the Gaussian Integral. However, by using numerical integration methods [26], it is possible to explore the relationship between the sigma of the statistical MAX operation as a function of a mean value for two or more distributions. Plots of $\sigma(MAX)$ for two distributions with one mean value constant while sweeping the other mean over an interval are illustrated in Figure 3.5, for three different mean and sigma values.

Our numerical analysis indicates firstly that $\sigma(MAX)$ will converge to the minimum sigma of the two distributions, and can never be smaller, and secondly that there exists a mean value (or effectively an offset), whereby $\sigma(MAX)$ will assume its final value, even though the two distributions will be overlapping. This is an important observation, as it relaxes wire offsets for achieving the minimum sigma value. The opposite requirement, *i.e.* that distributions do not overlap would imply a very significant delay at each wire assignment.

Hence, it is indeed possible, given SSTA results at a gate's inputs, to calculate a wire delay assignment at one of them, which would minimize sigma at the gate's output. One subtle detail is that the effect to the inserted wire load has to be accounted for. In the case where a circuit net forks to multiple gate inputs, *i.e.* output fanout is present, the delay assignments should be applied to the appropriate leg of the fork, feeding the relevant gate input of a successor gate.

Overall, the sigma at the gate output will depend on: (i) the narrowest distribution at the gate's inputs, (ii) the inserted wire load to skew this distribution which may affect output sigma, and (iii) the process parameters of the gate which will affect the ADD operation before the MAX at its output.

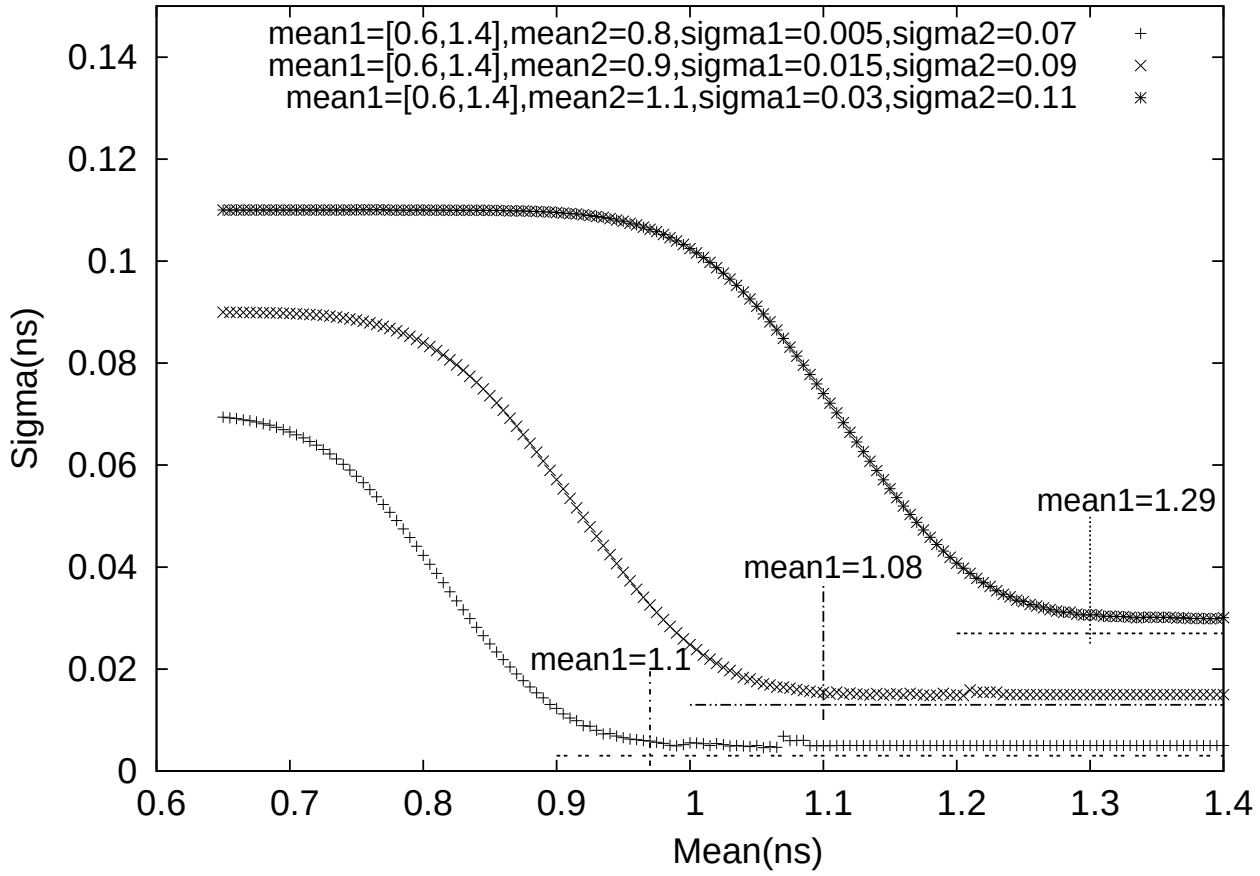


Figure 3.5: σ(MAX), as a function of mean

3.3 Minimum Sigma Slack Assignment

We derive a slack assignment for minimum sigma by a forward traversal of a circuit's timing graph, propagating minimum sigma at every gate. The Minimum Sigma Slack Assignment (MSSA) algorithm, shown in Algorithm 3.1 takes as input the circuit's Timing Graph (TG) and must be run after an initial SSTA analysis. The algorithm operates in block-based, breadth-first fashion, similar to the SSTA computation, traversing the timing graph forward, level by level (Line 3), checking whether new SSTA node results have propagated at all inputs of a gate (Line 9), by using array *SSTA* to indicate result progression. For each gate, with all its input *SSTA* nodes available, the minimum sigma input is selected by function **Min_Sigma** (Line 10). **Min_Offset** computes the wire delay offset that must be appended to the selected input, so as to minimize the sigma of the statistical MAX operation at the gate output (Line 11). Function

SSTA_tent_net performs local, tentative SSTA for a gate. It takes as input a given net along with a delay perturbation, and taking into account the wire load implied by this net delay, computes arrival times and sigma at the gate's output. The input node which yields minimum sigma at the output of the gate is found with Function **Min_Sigma_input** (Line 13). The new timing is committed to the timing graph by adding the offset to the selected input using Function **SSTA_inc_net** (Line 13). The final AT and sigma are stored into two MSSA arrays, *i.e. Arrival_MSSA* and *Sigma_MSSA* respectively. The timing graph assignment is flagged as final (Lines 15), thus indicating that new results are valid for successor level gates. Upon the algorithms' completion, the MSSA delay assignment is stored in array δ_{MSSA}, while arrays *Arrival_MSSA* and *Sigma_MSSA* contain the ATs and the MSSA bounds respectively.

Algorithm 3.1 - Minimum Sigma Slack Assignment (MSSA)

```
 1: MSSA(TG)
 2: SSTA[PIs] ⇐ 1;
 3: for currentlevel = 1 to MaxLevel(TG) do
 4:    LG ⇐ LevelGates(TG, currentlevel);
 5:    repeat
 6:       for all (gate_i ∈ LG) do
 7:          gateoutput ⇐ OutputNet(gate_i);
 8:          gateinputs ⇐ InputNets(gate_i);
 9:          if (∀ net_n ∈ gateinputs: SSTA[net_n] = 1) then
10:             minSigmaInput ⇐ Min_Sigma(gateinputs);
11:             δ_MSSA[minSigmaInput] ⇐ Min_Offset(minSigmaInput, gateinputs);
12:             done[gate_i] ⇐ 1;
13:             (Arrival_MSSA[gateoutput], Sigma_MSSA[gateoutput] ⇐
                SSTA_inc_net(gate_i, minSigmaInput, δ_MSSA[minSigmaInput]);
14:             for all fanoutnet ∈ Fanout(gate_i) do
15:                SSTA[fanoutnet] ⇐ 1;
16:             end for
17:          end if
18:       end for
19:    until (∀ gate_i ∈ LG: done[gate_i] = 1);
20: end for
21: return δ_MSSA, Arrival_MSSA, Sigma_MSSA;
```

After selecting a locally optimal sigma value for each gate node, MSSA will derive the globally optimal sigma based on minimum sigma propagating wire assignments, albeit with an effect on the mean. Nets with multiple fanouts must be handled as separate graph edges, *i.e.*

each fanout leg is handled individually, according to the sigma which is to be propagated to its successors.

Reconvergent fanout portions do not affect the algorithm's operation. The net fanout part of a reconvergent portion is handled as multiple nodes, whereby the successor gate of each leg will dictate whether wire delay is added to this fanout portion, while a reconverging gate will ensure minimal sigma propagation at its outputs. Thus, as the minimum sigma problem has an optimal substructure, *i.e.* locally optimal solutions combine to achieve the global optimum, both net fanouts and reconvengent paths do not affect the quality of the algorithm's results.

3.3.1 MSSA Superfluous Constraints

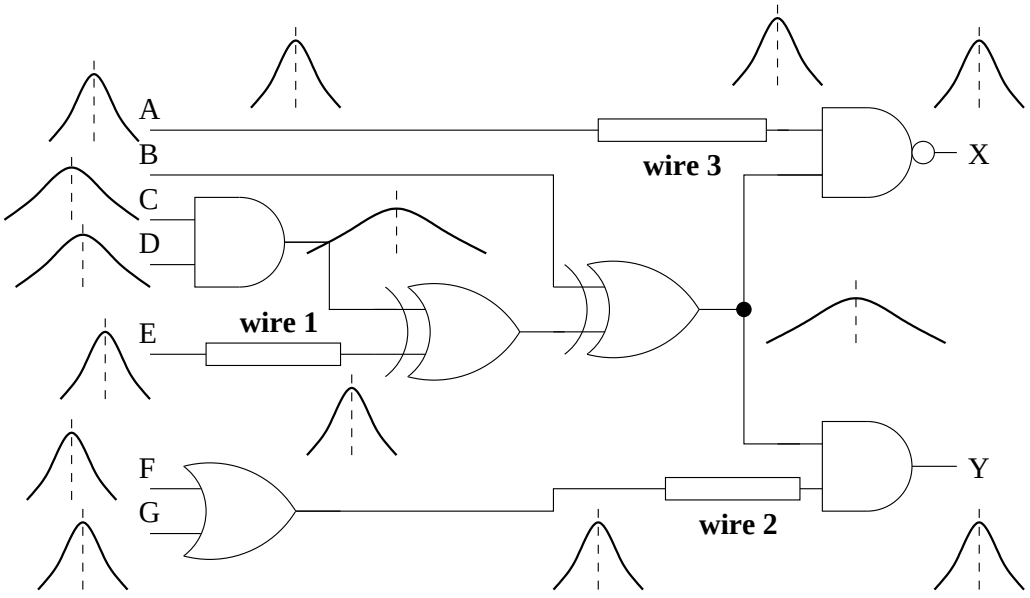


Figure 3.6: MSSA example with a superfluous constraint

A contrived, yet intuitive example illustrating a wire assignment of the MSSA algorithm is shown in Figure 3.6. A total of three wire bounds have been introduced: *wire 1* propagates the sigma of input E to the output of the first XOR gate, *wire 2* propagates the sigma of the OR gate's output to Y and *wire 3* propagates the sigma of input A to output X.

An aspect of the MSSA algorithm wire assignments illustrated by the example of Figure 3.6, is that some of the MSSA constraints may be superfluous, as MSSA only performs a single pass. In this case, as wires *2* and *3* essentially dominate the sigma of outputs X and Y, the

wire *1* constraint is superfluous with respect to obtaining minimum sigma at the outputs, and it merely adds unnecessary delay to the circuit.

Hence, such superfluous wire constraints will impact the MSSA circuit's mean. Eliminating them requires a second pass of the Timing Graph, in the reverse direction, *i.e.* from the sink node to the source node, whereby any wire delay assignment found, which is dominated by another of a successor level is removed.

3.3.2 Runtime Issues

MSSA is based on a BFS traversal of the timing graph. Thus, its timing complexity is *O(V+E)*, where V is the number of timing graph nodes and E is the number of edges. Each node is visited only once and the operations made on each node depend on the type of node. When MSSA visits a node corresponding to the input of a gate, it performs no operations. If the node corresponds to the output of the gate, then the aforementioned numerical method is performed. This, however, is a fast process. The selection of the input with the smallest sigma can be done in linear time with the number of inputs. It should be noted, that a typical gate will have no more than four inputs, the average being less than three inputs. Then, the amount of slack that needs to be applied to this input, so that its delay dominates other inputs' delays, is derived easily with a simple calculation. Thus, all the operations in MSSA are fast, making MSSA an efficient algorithm in terms of runtime. In Chapter 7, this analysis is confirmed by experimental runtimes.

In the next section we show how the results from MSSA can be utilized in our novel statistical slack allocation algorithm, TSZSA.

3.4 Target Sigma Propagation

In this section we describe our methodology for propagating a designated value for sigma to the outputs, instead of propagating the minimum sigma. This calls for a different algorithm than MSSA, which nonetheless must use the information from MSSA.

3.4.1 Motivation and Intuition for TSZSA

MSSA typically requires an excessive amount of slack to be added to the mean delay so that the selected distributions dominate the ones with larger sigma. It is unlikely though, that a designer

will need to constrain the sigma of the circuit's delay so much and to be willing to accept such a large penalty on the mean delay. The more typical case is that both mean and sigma will be values that the designer needs to be bounded from above. Thus, we have developed our Target Sigma Zero Slack Allocation (TSZSA) algorithm, which tries to find a slack allocation that guarantees that the mean and the sigma of the delays of the circuit's outputs meet target mean and target sigma constraints.

It is clear that TSZSA will require less slack to be added to selected nodes in order for the sigma of the delay of a gate's output to be reduced by only some amount instead of reaching its minimum. TSZSA was conceived with this idea in mind; as offsets are applied to inputs of a gate with relatively small sigma, the sigma of the gate's output delay is decreased until it reaches a minimum value. Thus, if a target for the sigma is known, then the amount of offset that must be applied to an input can also be derived. The questions that a TSZSA algorithm needs to answer is how to decrease the sigma of the delay at the output of a gate and how to propagate the distributions with reduced sigmas to the circuit's outputs.

The first question is answered in a similar way as in MSSA. If a target is set for the sigma of a gate output's delay, then the feasibility question can be immediately answered. If the target sigma is greater than the minimum sigma achieved by MSSA, then the target is feasible, otherwise TSZSA can conclude that the target sigma is unrealistic. In the feasibility case, there has to be an offset applied to the input with the smallest sigma that makes this distribution dominant in the calculation of the inputs' *MAX*, reducing the sigma of the *MAX* delay. In fact, it is not necessary that the input with the smallest sigma must be selected. Another input with relatively larger sigma, but requiring less offset might also be enough. Thus, by selecting an input, which given an offset, can reduce the sigma of the gate's output delay to the target sigma, we can derive the exact amount of offset needed and fix the sigma of the gate's output.

The second question can be solved by means of dynamic programming. Starting from the circuit's endpoints, we can first check if the target sigmas are feasible. If they are, we can determine the sigmas that are required at all the gates that drive the endpoints, which, if were in place, then a slack allocation could be found in order to fix the sigmas at the endpoints. At this point, the information from MSSA must be used again to check if the required sigmas at the drivers are generally feasible. By means of dynamic programming, we can traverse the circuit from the endpoints to the startpoints, setting targets for the sigma at each internal node. Then, with a forward traversal, we can meet the target sigmas for all internal nodes using a similar approach as in MSSA, the difference being that that target sigma is not the minimum

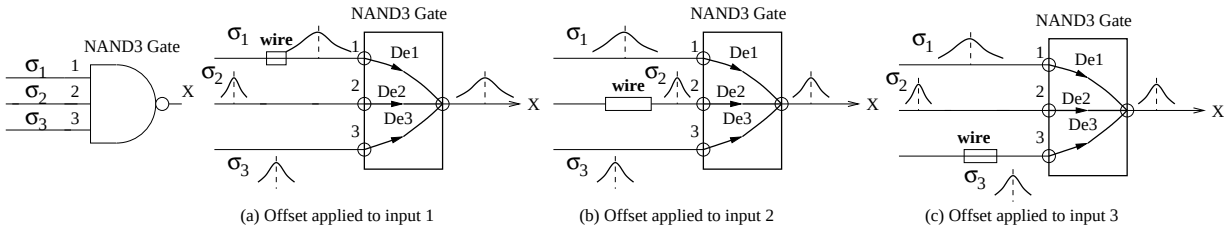
possible sigma.

In the following sections we describe how we solve these problems in practice.

3.4.2 TSZSA for a Single Gate

The foundation for developing our TSZSA algorithm is to find a way to decrease the sigma at the output of a single gate. As explained previously, this is done by selecting one input with relatively small sigma and applying to its delay distribution an amount of offset. There are two cases that need to be considered. The first case is that at least one of the inputs currently has a delay distribution, which if appropriately delayed, will cause a decrease in the sigma of the output. The second case is that none of the inputs can decrease the sigma of the gate's output with its current delay. In the first case, we select the best input and derive the offset that must be applied to it. The best input is the one requiring the minimum offset and is selected with a procedure detailed later in this section. In the second case, we assume that the delay of each input has a sigma that could, potentially cause a decrease in the gate output's sigma, if an appropriate offset was applied. Our task is to find the offset and an appropriate target for this input's sigma. This is done in two phases. First, we assume that this input is infinitely delayed and find the maximum sigma that it can have, in order for the sigma at the output of the gate to meet the target. Having fixed the maximum allowable sigma, we relax the infinite offset assumed in the first place, in order to find the minimum offset required for this input. We then select the best input as in the first case and we transfer the problem to fixing the sigma of the selected input, following the dynamic programming paradigm.

In order to make the best choice for the input, we scan all the inputs, taking into consideration both cases previously described and calculate the minimum offset that needs to be applied to any of them so that the sigma at the gate's output meets the target. This is graphically depicted in Figure 3.7. As inputs will generally have different delay distributions, different amounts of offset will be required for each one. It will be the case that some of the inputs will have too large sigma and their minimum sigma, as indicated by MSSA is too large too. These inputs will not be considered as suitable candidates for decreasing the output's sigma. In the case of Figure 3.7, *input 1* is rendered unsuitable, as no matter how much offset is applied to it, the sigma at the output is not decreased. For the remaining inputs, *input 3* requires the least amount of offset, so this is the one selected.



(a) Offset applied to input 1 (b) Offset applied to input 2 (c) Offset applied to input 3

Figure 3.7: Applying TSZSA offset to gate inputs

3.4.3 TSZSA Algorithm

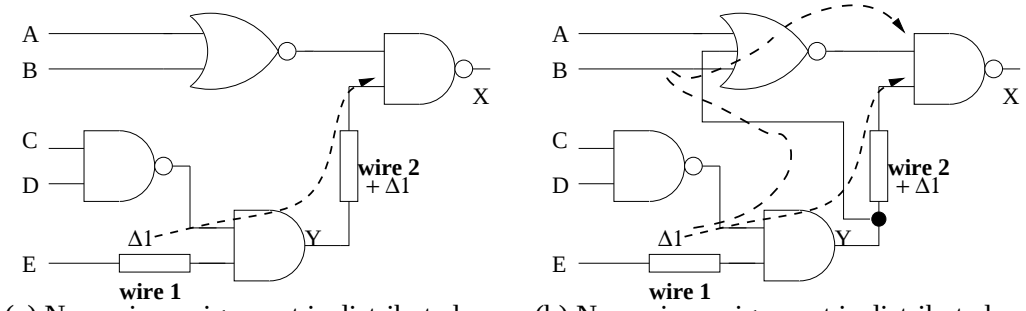
Having derived a minimum sigma bound based on wire assignments, we now present an algorithm which meets a sigma target, greater than the minimum sigma bound, while also preserving a target slack value at the circuit's POs, *i.e.* the virtual sink node. The TSZSA algorithm's inputs are the slack and sigma targets and the output is the wire delay assignments.

As it is impossible to know a priori, whether a required target sigma is achievable, the MSSA algorithm must first be run on the circuit, when minimum sigma bounds are stored at every node, to indicate feasibility. TSZSA uses MSSA assignments to relax sigma constraints, according to the target sigma, while eliminating superfluous sigma constraints, described in Section 3.3. This requires a backwards timing graph traversal, as both the target slack and the target sigma apply at the outputs, and their importance decreases towards the inputs, in accordance with the circuit's structure. At each gate, during this backward traversal, the best local wire delay assignment is selected, *i.e.* the wire delay on one of the gate's input nets of minimum length, which guarantees the target sigma at the gate's output. The target sigma of the non delay assigned wires is set to their current sigma, to relax their constraints backwards and avoid superfluous constraints.

3.4.4 TSZSA Wire Delay Propagation

One subtle detail which needs to be accounted for when assigning wire delays at a given node is whether any successor level's wire bounds are affected, *i.e.* may subsequently be reduced, as the introduction of the new wire assignment modifies their arrival time, hence less delay is now required for a successor level.

Figure 3.8 illustrates the two possible cases, which emerge while TSZSA traverses backwards. After a wire bound is added at *wire 2* to guarantee a target sigma at X, another wire bound is added at *wire 1*, as only G can meet the target sigma now required at node Y. The backwards



(a) New wire assignment is distributed unevenly to the next level

(b) New wire assignment is distributed evenly to the next level

Figure 3.8: Optimizing successor level bounds - possible cases

level of Y is 2, whereas the level of X is 1. Now after adding *wire 1* at level 2, the arrival time of the next constraint, *wire 2*, is affected. In the first circuit, Figure 3.8 (a), *wire 1* of delay *1*, increases the arrival time at Y by $+$ *1*, thus this delay should be be subtracted by *wire 2*'s delay. If this delay was not subtracted, the delay at *wire 2* would be superfluous increasing circuit delay and reducing available slack. This is not the case in Figure 3.8 (b), where the relative delays of the two paths leading to gate X must be preserved. We distinguish between these two cases and remove any superfluous wire delay assignments.

3.5 Target Sigma Zero Slack Assignment

TSZSA, illustrated in Algorithm 3.2, operates in a backwards, breadth-first search fashion, *i.e.* from the virtual sink node of the Timing Graph (TG), towards the virtual source node (Lines 5 to 33). The target constraints for each node are stored in arrays *Slack* and *TSigma* (Lines 3 and 4), whereas array *SSTA* indicates result progression beginning from the POs. All arrays begin initialized to zero. Array *Sigma_MSSA* is the array of lowest sigma bounds obtained by the MSSA algorithm's execution. For each gate of the current level, the slack is computed (Line 13). The **Min** operation is required as a net may fanout to multiple gates and may already have been assigned slack from another gate. Slack is computed as the minimum of the currently assigned value, and the difference of gate delay subtracted from the slack at the gate's output. Function **ComputeNetDelayforSigmaTarget** (Line 14) performs iterative wire delay analysis, at gate input j, in order to achieve the target sigma at the gate's output. It takes as input the MSSA sigma values, provided by *Sigma_MSSA*, to assess that the target sigma specified is achievable. This process is performed for all gate inputs and results are

Algorithm 3.2 - Target Sigma Slack Assignment (TSZSA)

```
 1: TSZSA(TG, targetsigma, targetslack, Sigma_MSSA)
 2: SSTA[POs] ⇐ 1;
 3: Slack[POs] = targetslack;
 4: TSigma[POs] = targetsigma;
 5: for currentlevel = MaxLevel(TG) to 1 do
 6:    LG ⇐ LevelGates(TG, currentlevel);
 7:    repeat
 8:       for all (gate_i ∈ LG) do
 9:          gateoutput ⇐ OutputNet(gate_i);
10:          if SSTA[gateoutput] = 1 then
11:             Clearqueue(Q);
12:             for j = 1 to NoOfInputNets(gate_i) do
13:                Slack[j] = Min(Slack[j], (Slack[gateoutput] - GateEdgeDelay(j,
                   gateoutput)));
14:                (δ[j], Sigma[j]) = ComputeNetDelayforSigmaTarget(gate_i,
                   TSigma[gateoutput], j, Sigma_MSSA[j]);
15:                AscendingOrderEnqueue(Q, δ[j], Sigma[j], gateinputs[j]);
16:             end for
17:             (δ_TSZSA[net_TSZSA], Sigma[j], net_TSZSA) ⇐ Head(Q);
18:             done[gate_i] ⇐ 1;
19:             SSTA_inc_net(gate_i, net_TSZSA, δ_MSSA[net_TSZSA]);
20:             Slack[net_TSZSA] = Min(Slack[net_TSZSA], (Slack[net_TSZSA] - δ_TSZSA[net_TSZSA]));
21:             TSZSA_tighten_level(currentlevel + 1, gate_i, net_TSZSA, δ_MSSA[net_TSZSA]);
22:             for j = 1 to NoInputNets(gate_i) do
23:                if j ≠ net_TSZSA then
24:                   TSigma[j] = Min(TSigma[j], StdDev[j]);
25:                end if
26:             end for
27:             for all faninnet ∈ Fanin(gate_i) do
28:                SSTA[faninnet] ⇐ 1;
29:             end for
30:          end if
31:       end for
32:    until (∀ gate_i ∈ LG: done[gate_i] = 1);
33: end for
34: return δ_TSZSA;
```

sorted by wire delay in a queue, by function **AscendingOrderEnqueue** (Line 15). The

optimal solution is the wire assignment which achieves the target sigma while adding minimum wire delay to one of the gate's inputs. This solution will reside at the head of the queue and is selected (Line 17). The gate is flagged as complete (Line 18), the result of adding the wire delay is committed locally to the Timing Graph by using function **SSTA_inc_net** (Line 19) and the slack of the node is updated, taking into account the added wire delay (Line 20). Function **TSZSA_tighten_level** (Line 21), as explained in Section 3.4.4, checks the impact of the addition of the newly added wire delay to wire delays of the successor level, potentially reducing them to their minimum delay possible without altering their target sigma. If **TSZSA_tighten_level** makes any modifications to successor level wire bounds, it will update the Slack values of both successor level wire bounds, and of the current wire bound. Then, the sigma target of other side inputs is set to their current sigma, *i.e.* their current standard deviation based on the zero wire delay SSTA analysis, unless it is set to a smaller value by another gate, to which this net fans out. This is the purpose of the **Min** operation (Line 24). Finally, the algorithm flags the gate's inputs (Lines 32 to 33), proceeds to consider other gates at this level and then to predecessor levels. Upon the algorithm's completion, the TSZSA constraints are stored in array δ_{TSZSA}.

At the end of the TSZSA algorithms' execution, there are two possibilities, (i) there will either be available slack at the circuit's startpoints, which implies that the original target slack was sufficient to satisfy the sigma constraints, or (ii) the slack at the startpoints will be negative, meaning that it was not possible to assign the allocated slack to achieve the sigma target. In the first case, where slack is left over at the startpoints, the remaining slack should be allocated to circuit's wires, but without significantly altering the target sigma. We solve this issue by formulating a linear programing problem, which aims to distribute the slack without affecting the relations between the delays of the inputs of each gate. This is explained in detail in Section 3.6. In the second case, the sigma constraint conflicts with the mean constraint. Depending on how tight the slack is, there can be cases, where all of the available slack is allocated to satisfy sigma constraints, and some nets end up with zero wire delay. These cases are also unrealistic.

3.6 LP slack assignment

Since we aim at using the TSZSA in the framework of a physical algorithm, we need to make sure that the derived allocation can provide realistic wire bounds for use in a physical tool like

placement. Thus, we have to distribute any remaining slack to nets in such a way that the physical algorithm can create the nets according to their slack. This means that nets must not be assigned too small or too large slacks. The former would call for too short nets, making, *e.g.* placement of the gates connecting the nets too constrained and the latter would require too large nets which would require aggressive lower bounds on the distance between the connected gates. TSZSA guarantees a reasonable slack allocation by formulating all the aforementioned constraints in a linear programming (LP) problem.

The constraints for the LP problem can be divided into separate categories.

- **Absolute constraints on the maximum length of each wire.** This type of constraint ensures that the physical algorithm will not be required to create too long wires.
- **Absolute constraints on the minimum length of each wire.** This type of constraint aims at avoiding aggressive upper bounds on the physical distance among gates, which would be the case if wires were requested to be too short.
- **Relative constraints for each pair of gate inputs.** This type of constraint ensures that the relative order of delay distributions are preserved after slack allocation. This is especially important, as TSZSA is based on selecting a suitable input of a gate and specifically applying offset to it, in order for the sigma of the gate's output to be decreased.
- **Absolute constraints on the mean delay of the endpoints.** This essentially corresponds to the available slack that is to be distributed. The final mean delay of each endpoint must not violate the upper bound on the mean delay.
- **A description of the timing graph in form of linear programming.** This description provides the information of the connections in the timing graph. The delays at gate outputs are modeled as the sum of the delays at the gate inputs plus the propagation delay. The delay at a gate input is modeled as being at least as much as the delay of the driver plus the delay of the wire connecting them.
- **The objective function.** We set the objective function to maximize the total length of all wires. This, coupled with the lower and upper bounds on the delay of each wire will provide a reasonable slack allocation if the problem is feasible.

We now describe the LP formulation in detail.

3.6.1 LP formulation for Statistical Slack Assignment

We have defined the LP problem in such a way that the allowable delay for each wire is maximized. Thus, Equation 3.9 shows the problem definition.

$$\begin{aligned} max \quad & \Sigma w_i \\ w_i \quad & \leq \quad UpperBound \\ w_i \quad & \geq \quad LowerBound \\ d(g_i) \quad & \leq \quad MeanConstraint \end{aligned} \tag{3.9}$$

where w_i is the delay of a wire in the circuit, $UpperBound$ is the maximum delay for each wire, $LowerBound$ is the minimum allowable delay for each wire, $MeanConstraint$ is the maximum allowable mean delay for each endpoint and $d(g_i)$ is the delay of a timing node in the circuit. Additionally, the TSZSA constraints are added in the LP formulation, as shown in Equation 3.10.

$$\begin{gathered} \text{for each } ((d(g_i), d(g_j)) \text{ TSZSA constraint:} \\ d(g_i) \geq d(g_j) \end{gathered} \tag{3.10}$$

Equation 3.10 adds the relative TSZSA constraint to the LP problem, *i.e.* adds the constraint that a timing node is slower than another timing node. This type of constraint is especially important for preserving the offsets added by TSZSA. It is not harmful for the offset to be augmented by the slack allocation. However, since TSZSA has found the minimum offset that must be applied to a gate input, a slack allocation which effectively decreases the assigned offset is not acceptable.

The relations between timing nodes and wires are described in Equation 3.11.

$$\begin{aligned} &\text{for each input:} \\ &\qquad d(g_i) \geq d(g_{driver}) + w_i \\ &\text{for each output:} \\ &\qquad d(g_i) = d(g_{slowest\_in}) + d_{propagation} \end{aligned} \tag{3.11}$$

Equation 3.11 determines that the delay of an input is the sum of the delay of its driver plus the delay of the wire that connects the input to its driver. The delay of w_i is the one which we require to maximize in the LP cost function. Equation 3.11 also determines that the delay of an output is the delay of the slowest input for the given gate, plus the propagation delay for

the gate.

Thus, the above formulation allows for a well-defined LP problem, which can be solved with standard LP solvers. The difference from a traditional zero-slack assignment algorithm, used for reference later in this thesis, is that the aforementioned LP formulation incorporates the TSZSA constraints, thus the resulting assignment not only is a zero slack assignment, but also one that guarantees a target sigma at the outputs. We have used the GNU GLPK solver [25] to solve the LP problem and produce the slack bounds on wires to use later in the placement algorithm.

3.6.2 Runtime Improvement Through Hierarchical LP

GLPK, which uses the revised simplex method [55] can find optimal solutions of the LP problem. However, the runtime can be prohibitive, spanning multiple hours even for circuits of a few thousand standard cells. Thus, we have developed a hierarchical approach to tackle the runtime problem and enhance scalability of our algorithms.

Although part of the constraint set derived by TSZSA is of relative nature, the constraints involved typically require information from a relatively small portion of the circuit. Most constraints can be derived independently to each other, thus the application of a hierarchical approach is possible. In order to employ our hierarchical approach, we allocate slacks first to all wires, in a greedy and fast way, which, in the general case will not be optimal in terms of sigma optimization. We do this, to form an initial solution, which may be suboptimal, but will provide directives for the LP optimization process. Next, we employ a recursive bisection approach, at the circuit graph level, which divides the problem of slack allocation into smaller problems of the same characteristics as the initial problem. We keep bisecting until the segments are small enough so that the problem can be solved locally and fast by GLPK.

The main idea is to derive circuit portions, which have similar structure to the original circuit, *i.e.* primary inputs, primary outputs, internal nodes, startpoints and endpoints. All startpoints, be it primary inputs or sequential elements, will have input arrival times, as the startpoints of the original circuit do. All endpoints, be it primary outputs or sequential elements, will have bounds on their mean and sigma of delays, as is the case with the original circuit. The problem on the circuit portion is essentially the same as in the original circuit and can be solved by an appropriate LP formulation using the same approach as the one we use for the whole problem. The problem is now reduced to creating the circuit portions.

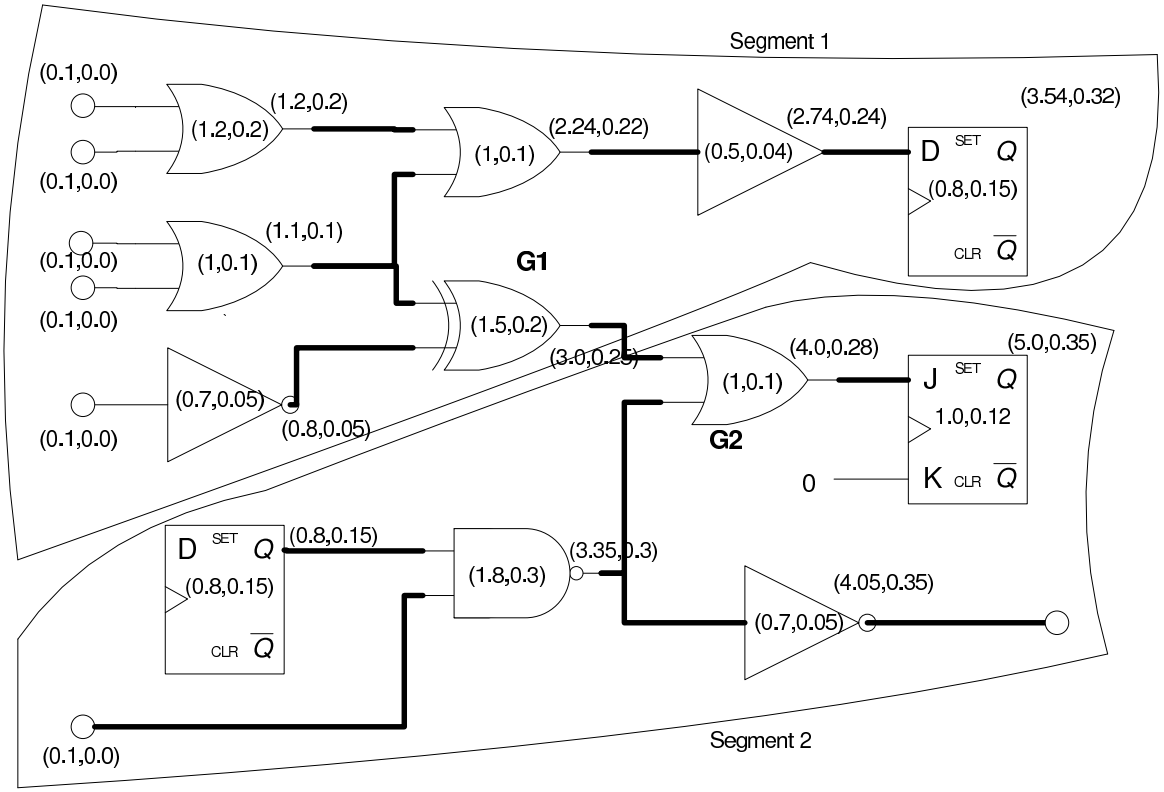


Figure 3.9: Hierarchical support for LP

We use a min-cut bisection approach in order to minimize the number of nets spanning among segments. This step is easily done using any min-cut bisection algorithm like Kernighan-Lin, Fiduccia-Mattheyses or more advanced like hMetis. An example of a bisection is shown in Figure 3.9. The original circuit has been bisected into two segments, `Segment1` and `Segment2` with only one wire spanning between the two segments, the one connecting gates `G1` and `G2`.

Each segment is then treated independently. Given the statistical timing analysis of the original circuit, the arrival times and bounds on each timing node of the circuit are known. For example, `Segment1`, can be handled as an independent circuit with five primary inputs feeding three input gates, three internal gates, one endpoint in the form of a sequential element and one additional, artificial endpoint, the output of `G1`. The bounds on the statistical delay of this output are defined by the previously done SSTA and greedy slack assignment, which in this case dictate that the maximum (μ, σ) should be (3.54, 0.32) for the sequential element and (3.0, 0.25) for the artificial endpoint. Similarly, `Segment2` has one primary input, one input in the form of a sequential element, one internal gate, one primary output (the inverter's output),

one endpoint in the form of a sequential element and one primary input feeding gate `G2`. In this case, the arrival time at `G2` is known to have a value of (3.0, 0.25). If these two segments are small enough for the LP problem to be solved efficiently, then we formulate the two problems, one for each segment, and solve them independently. Otherwise, we keep bisecting forming smaller circuits. The solutions of the two LP problems on the two segments are then combined to form the solution for the whole circuit.

However, there is a point of interest exactly at the wire connecting the two segments. The slack of this wire cannot be found by either of the two LP problems. The slack for this wire is determined by the initial slack assignment and will not be altered, unless any of the smaller problems prove to be infeasible. In this case, we merge the two segments, do a second bisection and prohibit this problematic wire to span between any two segments. In the new bisection, this wire will belong entirely to a segment and its slack will be optimized by the solution of an LP formulation.

We guarantee that the hierarchical algorithm will, in the end find a solution if the original problem is feasible by adding a full merging approach. If, after a number of unsuccessful trials, a solution cannot be found for the smaller segments, they are merged recursively and LP is formulated on the merged segments. This, will give greater freedom of optimization, but will also require larger runtime. In the extreme case, all segments will be merged back to the initial problem and the LP will be formulated on the whole circuit. Thus, in the worst-case scenario, although LP will fail to benefit from the hierarchical approach, it will still search for a solution on the initial problem and will, eventually, find a solution if this is possible.

Having described our novel methodology for deriving statistical wire bounds, we now present our constructive statistical-based placement algorithm, *SCPlace*.

Chapter 4

SCPlace

In this chapter we describe SCPlace, our placement tool which can perform statistical optimization for delay, through placement, on a circuit. As discussed in Section 2.3.7, although statistical approaches are being developed both in industry and in academia, there is lack of support from large-scale optimization tools like placers. SCPlace is our answer to this problem. SCPlace is a hybrid constructive-iterative placer, which incorporates (i) the statistical wire bounds produced by TSZSA (cf. Section 3.4), and (ii) the LP-based slack assignment for the TSZSA's bounds. We tackled the inherent deficiencies of constructive placers by incorporating iterative heuristic approaches in the inner loop of SCPlace's optimization, enhancing SCPlace's solution space search mechanism. We addressed the non-uniform density and routability issues by introducing density screens during the placement process, as described in Section 4.7, which ensure firstly that congestion does not exceed a maximum within the density screen area, and secondly that sparsely populated areas are placement candidates. SCPlace selects legal cell locations by default, thus placement legalization, as a post-placement step is not necessary.

We start the discussion with the description of a rather naive placer we developed, aimed at tackling the problem of statistical delay optimization. In the context of *SSAPlace*, we will show in Section 7.3 that statistical timing cannot be optimized using traditional placement approaches, employed by contemporary placement tools, but alternative approaches, like the ones employed by SCPlace, must be targeted for. Next, we describe SCPlace and then we proceed to more technical details, *e.g.* the requirements of SCPlace for statistical optimization, its interface with other tools, its optimization objectives and the implementation details.

4.1 Motivation for SCPlace

We have developed SSAPlace in order to highlight the inefficiency of monolithic iterative approaches for optimizing for double-sided constraints like the ones required for statistical optimization. Although we do not provide a formal proof for the unsuitability of iterative approaches for the problem we are targeting, experimental results, presented in Section 7.3 on SSAPlace, confirm our intuition. In fact, SSAPlace provided the motivation for developing our slack-based statistical optimization framework. SSAPlace is a simulated annealing algorithm which accepts a move if it improves the mean and sigma of an output. Alternatively, it accepts moves which do not negatively affect the mean delay of the slowest output, but improve sigma locally. A move may also be accepted if it improves the mean or wirelength without increasing the local sigma. Pseudocode for SSAPlace is given in Algorithm 4.1. The greedy condition for accepting a move is performed in Line 9. This directly requires an improvement in either mean, sigma or wirelength.

Algorithm 4.1 - SSAPlace Algorithm

```
 1: SSAPlace()
 2: it ⇐ num_placed * 2
 3: for i = 0 to it do
 4:    cell ⇐ Placed[random]
 5:    Locations ⇐ PossibleLocations(cell)
 6:    new_location ⇐ Locations[random]
 7:    stored_location(cell) ⇐ Location(cell)
 8:    Location(cell) ⇐ new_location
 9:    if Improved(mean) || Improved(sigma) || Improved(wl) || HillClimb-
       ing(mean,sigma,wl) then
10:       Store(mean,sigma,wl)
11:    else
12:       Location(cell) ⇐ stored_location(cell)
13:    end if
14: end for
```

Experimental results (cf. Section 7.3) showed that SSAPlace cannot effectively manipulate the circuit's statistical delay, as local improvements in sigma do not necessarily propagate to the circuit's outputs the way improvements on mean do. Thus, we have shifted our attention to the development of SCPlace which manipulates both mean and sigma through the use of statistical wire bounds, instead of relying to local, greedy optimizations.

4.2 Description and Intuition for SCPlace

SCPlace uses the TSZSA wire bounds as guidance for local decision-making as to where to place cells. In Section 2.3 we discussed that constructive approaches are better suited for placement problems which make use of two-sided constraints. This is the case in statistical optimization and thus, we have developed SCPlace in the general framework of a constructive placer. We have used the cluster-growth model, which dictates that, after placing a few seed gates, then clusters are formed around them with gates with which they share common constraints. The cluster-growth model can be of great value to the problem of placing gates, which have relative timing constraints, in appropriate locations. An illustrating example is the case of the drivers of a gate's inputs, whose inputs share a relative timing constraint that guarantees the sigma value of the gate output's delay. By forming a cluster for these specific gates, they can be placed in appropriate locations and they can also be moved as a cluster in another location if the optimization requires so.

Constructive placers are known to be prone to make inefficient decisions early in the placement process due to incomplete information stemming from gates which have not been placed yet. We address this problem by employing a reconstruction step, described in Section 4.7, which operates as soon as one such deficiency is uncovered. Reconstruction utilizes the notion that each gate does not have only one optimal location. In the case that an early, unfortunate placement of a gate hampers the placement of other gates, then reconstruction replaces gates by examining some of the optimal locations for each gate.

Another deficiency of constructive placers is that they lack global information on gates which do not share a relation to the gates currently being placed. This could lead some gates to occupy locations which may be optimal, taking into account their connections, but seriously reduce the probability that other, unrelated gates find enough room to be placed in the same locations. We overcome this problem by incorporating an iterative process, described in Section 4.7, which shuffles the placement of the already placed gates. We use a simulated annealing algorithm, working on the already placed gates in order to find alternative locations for them, minimizing global wirelength in the process. This process, enables the constructive approach to escape local minima, which may not be possible if the placed gates are regarded as fixed from the point they are placed until placement is complete.

Any placement algorithm needs to be scalable with circuit size, as undoubtedly, circuit sizes are going to be increasing as integration progresses. The constructive nature of SCPlace

ensures that, each gate is generally visited only once and placed at its, almost, final location. The bottleneck for SCPlace is the derivation of TSZSA wire bounds, which is done efficiently by utilizing the hierarchical approach described in Section 3.6.2. We have coupled this approach with a divide-and-conquer approach which we employ during constructive placement greatly enhancing the scalability of SCPlace. This is described in detail in Section 4.8.

We further enhance the probability that SCPlace does not get stuck in local minima by utilizing the complete information on the already placed gates through re-assigning slacks using TSZSA. We employ this process only if after a full pass not all gates have been successfully placed. TSZSA then, regarding placed wires as fixed, re-assigns slacks in the remaining, unplaced wires, greatly enhancing the chances that suitable locations are found for the unplaced gates in the next iteration. We describe this idea in Section 4.7.7.

In the next section we discuss the additional requirements that are needed for SCPlace, being a statistical placer, compared to a non-statistical placer.

4.3 Requirements for Statistical Placement

In order to perform statistical placement optimization, a set of statistical constraints are necessary. SCPlace utilizes constraints on both the mean and the sigma of the required statistical circuit delay. These can be either minimum or maximum constraints, as is the typical case in a non-statistical placer. SCPlace can additionally accept statistical constraints on specific gates of the circuit. Due to the implementation of TSZSA, which derives relative statistical constraints among gates, the designer may also impose a number of relative constraints at their discretion. These can then be forwarded to the LP formulation described in Section 3.6 and be incorporated into the statistical slack assignment, performed by TSZSA, which is then passed to the core of SCPlace.

In addition to statistical constraints, SCPlace requires a statistical model for variation and a statistical model for delay of standard cells. The use of these models is detached from the main placement engine of SCPlace. This means that as research in this area advances, SCPlace can employ diverse or more accurate models. Currently, SCPlace incorporates its default models, but the adaptation of other models is straightforward.

For the statistical delay of standard cells, a number of technology libraries must be provided to SCPlace. These libraries must correspond to different operating conditions. SCPlace extracts the delay values from the technology libraries and fits them automatically into a nor-

mal distribution, as explained in Section 3.1. In the case that only one technology library is available, then SCPlace will assume zero standard deviation for the delay, which will essentially turn SCPlace into a non-statistical timing-driven placer.

For the variation model, SCPlace supports a number of different correlation functions, the definition of which can be provided by the designer, depending on the technology used. Currently, SCPlace assumes only spatial correlations, which have the geometric notion that correlation between devices is inversely proportional to their physical distance. Other correlation functions are in place and can be activated at the designer's discretion. These include correlation due to gate similarity and correlation due to the depth of each gate in the paths it belongs to. SCPlace can also be easily extended to handle hard-wired correlation functions which can define specific amounts of correlation for specific regions of the layout, due to any fabrication technology characteristics.

Finally, a model for wire delays is needed. SCPlace employs the bounding-box technique for the estimation of wire delays. According to this technique, the bounding box for all terminals of a wire is created and the length of the net is estimated as half the length of this bounding box. This is illustrated in Figure 4.1. Bounding-box estimation is one of the most commonly used techniques in both industrial and academic placers. Once the length of the net is determined, its delay is calculated using the Elmore RC delay model [68]. Given the unit capacitance c_u and the unit resistance r_u, the delay d of a wire of length l is estimated as $d = \frac{r_u c_u l^2}{2}$.

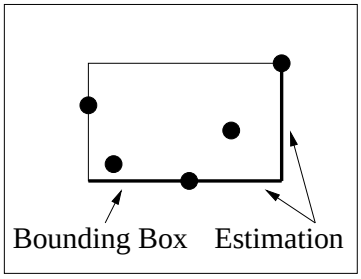


Figure 4.1: Bounding box estimation

4.4 SCPlace Interface

SCPlace conforms to the industry standards for technology file and circuit model definitions which renders it suitable for use in any industrial flow. SCPlace supports the gate-level *Verilog*

circuit description language. The technology description of standard cells is given to SCPlace using the two standard library description formats, *LEF* and *LIB*. The description of final placement is produced by SCPlace using a *DEF* file.

The netlist, provided in Verilog format, and the placement, provided in *DEF* format are the two files needed by a typical fabrication flow in the next step, which is routing. We have verified the usability of SCPlace's placements by routers, by forwarding the placements of SCPlace into a state-of-the-art industrial router using the the industry's standard file formats.

4.5 Optimization Objectives

SCPlace is a timing-driven placer, albeit timing is expressed statistically. Thus, SCPlace operates using statistical timing constraints and performs statistical optimization. Additionally, SCPlace must conform with a set of physical requirements. We now describe the full list of constraints SCPlace can handle.

- **Statistical timing constraints.** SCPlace accepts a global timing constraint for the mean of the circuit's delay and a constraint for the standard deviation (sigma) of the circuit's delay. These are then forwarded to the TSZSA algorithm which will derive bounds for all circuit's wires. If the constraints are feasible, then the bounds will be transformed into slack which directly correspond to wire lengths. SCPlace's task then is to find locations for all cells that satisfy the suggested wire lengths. Statistical timing constraints can also be introduced locally, *i.e.* at internal nodes of the circuit, rather than only on the circuit's endpoints. In this case, TSZSA will also derive bounds taking into account the local constraints. SCPlace then, will proceed as usual working on the suggested wire lengths, as calculated by TSZSA.

- **Wire length constraints.** A set of constraints for maximum or minimum lengths for specific wires is supported by SCPlace. These are formulated as a LP problem. Fixed wire lengths are also supported by SCPlace. They too are inserted into the LP problem, the only difference being that they cannot be optimized.

- **Layout area.** SCPlace works on a given layout area, which must have enough space to accommodate all cells plus sufficient white space. The layout area does not have to be rectangular, or continuous. That means that there may be blockages where no standard

cells can be placed. SCPlace can identify all valid locations and will try to utilize all available locations in order to find the best place for each standard cell.

- **Density screens.** In order to avoid over-congested areas, SCPlace accepts a grid of any granularity, which defines the maximum density allowed at each region of the layout. At all stages of placement, SCPlace checks the density in the region where it intends to place a cell and does not place it, if it violates the density constraint.
- **Design-rule constraints.** The most important design-rule constraint that SCPlace satisfies is that of no cell overlaps. SCPlace keeps track of the occupied layout locations, as the placement progresses, and does not create overlaps by placing a standard cell over another cell at any point in time. Thus, a legalization step is not necessary after placement, which could violate the timing constraints. Moreover, SCPlace aligns all cells on a manufacturing grid, which is standard procedure in any industrial placer.

4.6 The SCPlace Flow

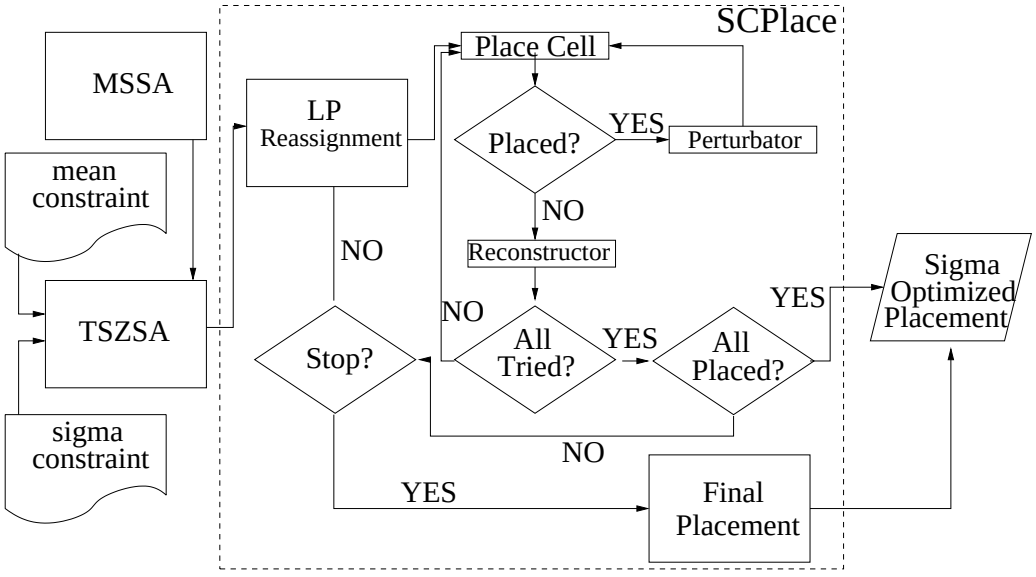


Figure 4.2: SCPlace's flow

The SCPlace flow is illustrated in Figure 4.2. It begins with the MSSA, TSZSA assignment, whereby TSZSA assigns the initial placement wire bounds for a given statistical constraint,

i.e. (mean, sigma) pair. The LP slack assignment will initially distribute any remaining excess slack. At this point, SCPlace's core will enter the constructive placement process.

Pseudocode for the SCPlace is given in Algorithm 4.2. SCPlace starts by assigning slacks to all nets (Line 2) based on the (μ, σ) constraint. Then, it sorts all cells according to the number of their connections (Line 3). For a number of iterations, which depends on the number of cells, SCPlace visits all cells in the sorted array and tries to place them (Lines 4 to 21). For every cell visited, if it it not already placed (Line 8), SCPlace tries to place it satisfying all its constraints (Line 9). If the cell is successfully placed, then all its neighbours are added to a queue (Line 10) and all these cells are tried to be placed (Lines 11 to 14), making sure that no violations are created at any time. If the cell was not placed, then the *Reconstructor* algorithm is called (Line 16). After one full sweep of the sorted list, the *Perturbator* algorithm is called (Line 20) in order to shuffle the placement for the next iteration. After the number of iterations has passed, the placement is finalized (Line 22).

Algorithm 4.2 - SCPlace Constructive Placement

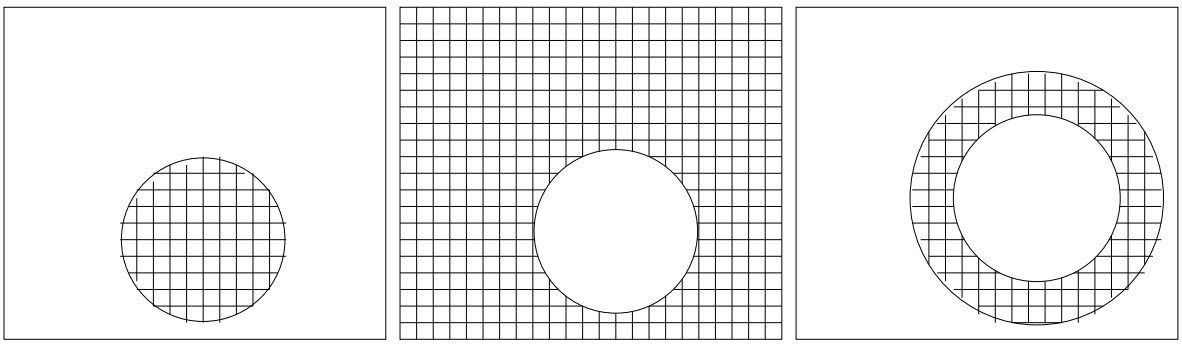
```
 1: SCPlace(μ, σ)
 2: SlackAssignment(μ, σ)
 3: SortedCells ⇐ SortConnectivity(cells)
 4: it ⇐ num_cells / 100
 5: for i = 0 to it do
 6:    for j = 0 to num_cells do
 7:       cell ⇐ SortedCells[j]
 8:       if !Placed[cell] then
 9:          if Place_Constrained(cell) then
10:             NeighboursQueue ⇐ Neighbours(cell)
11:             for all neighbour_i ∈ NeighboursQueue do
12:                Place_Constrained(neighbour_i)
13:                AddNeighbours(NeighboursQueue, neighbour_i)
14:             end for
15:          else
16:             Reconstructor(cell)
17:          end if
18:       end if
19:    end for
20:    Perturbator()
21: end for
22: SCPlace Finalization()
```

4.7 Implementation Details

In this section, we describe in detail the specific implementation aspects of SCPlace.

4.7.1 Constructive Process

SCPlace follows a constructive approach for cell placement, which means that a set of cells are placed first at random locations and then, remaining cells are placed in locations considered best. The best choice of location for a cell is dictated by its constraints. These are always related to some other cells and are always referring to the distance to other cells. The basic idea is to consider a list of possible and legal locations for each cell and then select one at random. A legal location is any location which does not cause overlaps and aligns the cell to the manufacturing grid. With respect to the constraints, there may upper bounds, lower bounds, or a combination of upper and lower bounds. We treat each case differently.



(a) Upper bound constraint (b) Lower bound constraint (c) Upper and lower bound constraint

Figure 4.3: Upper and lower bound constraint types

In the case of an upper bound on the distance to another cell, the valid locations for a cell lie within a disk whose radius is exactly the upper bound on distance. This is illustrated in Figure 4.3a. Any location in the disk is valid, so any location can be chosen. In Figure 4.3a, the disk is shown segmented by the rows and the manufacturing grid, so the valid locations, from which one will be chosen, have become apparent.

In the case of a lower bound on distance, the allowable region at which a cell may be placed is outside the disk whose radius is exactly equal to the lower bound. This is shown in

Figure 4.3b. The segmentation of the allowable area by the rows and the manufacturing grid, allows for creating a list for all valid locations.

In the combined case of a lower and an upper bounds, we form a ring, as shown in Figure 4.3c. This case is slightly more constrained than the simple cases of upper or lower bounds, but the allowable locations can be similarly extracted.

Since cells can connect to more than one cell, it is not unusual that the constraints originate from more than one sources. This allows for two cases. Either there is an overlap for the allowable regions derived from all constraints, or there is no such overlap. In the first case, the cell may be placed in any location in the overlap region, while in the second case, the placement of the cell is infeasible. The feasible case is illustrated in Figure 4.4. The second case is the fundamental problem with any constructive approach; unfortunate placement of some cells may over-constrain the placement of other cells or even make it infeasible. This issue is overcome with our *Reconstruction* and *Perturbation* mechanisms described in Sections 4.7.2 and 4.7.3.

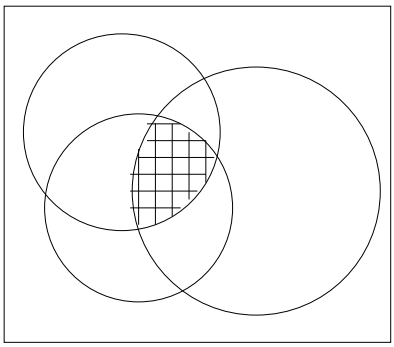
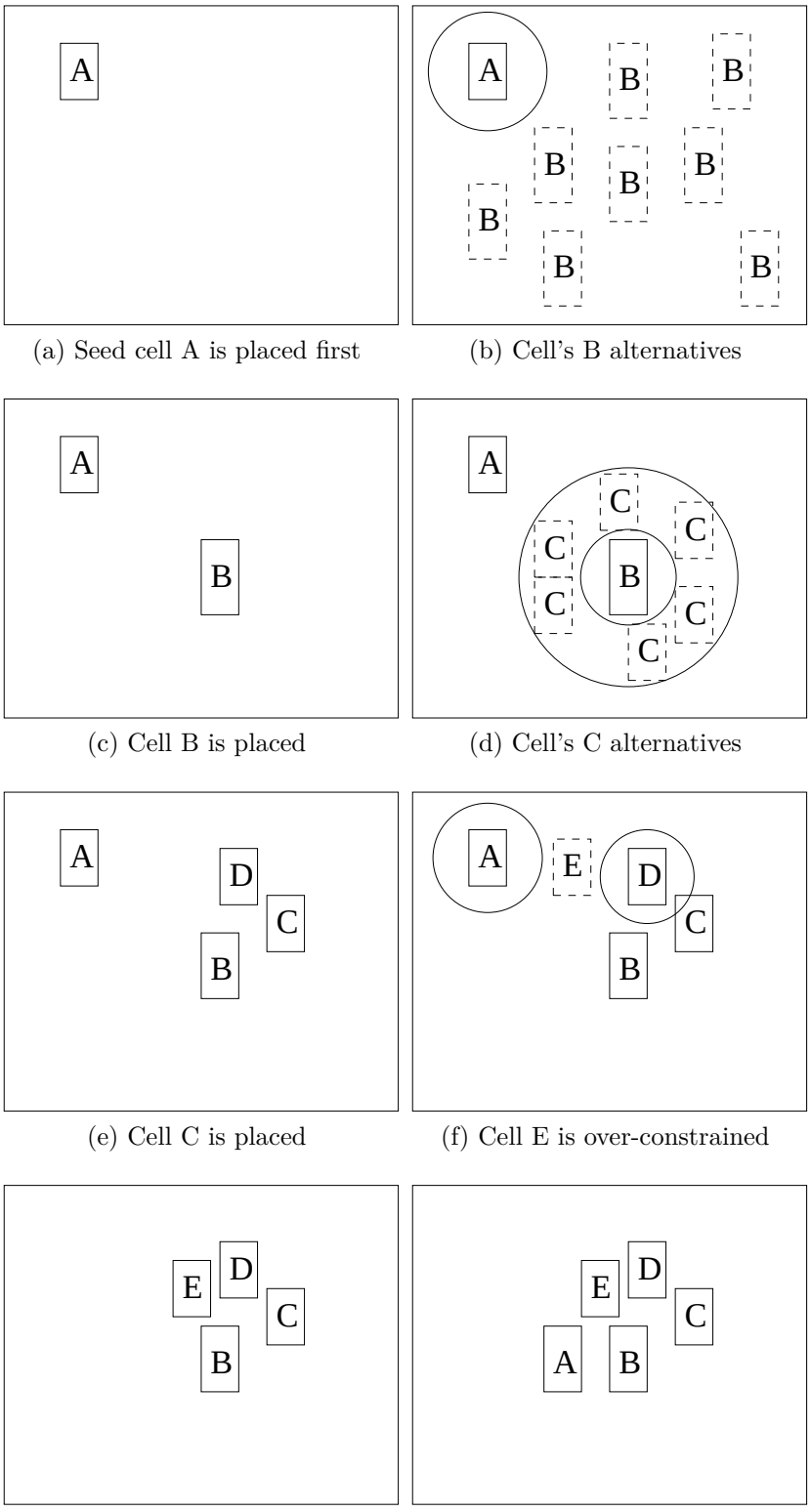


Figure 4.4: Multiple physical distance constraints

4.7.2 Reconstruction

Cell reconstruction attempts to improve on unfortunate decisions made early in the constructive placement due to incomplete information of the potential placement of other cells. The problem, along with the solution reconstruction provides is illustrated in Figure 4.5.

In Figure 4.5, five cells are placed constructively. First, seed cell A is placed at a random location. Then, cell B, which has a constraint with cell A bounding the minimum distance between the two cells must be placed. Figure 4.5b shows a list of possible locations for cell B. In the absence of any other constraints, since no other cells have been placed, cell B can be placed in any of these locations. This is shown in Figure 4.5c. Then, cell C, which shares



(a) Seed cell A is placed first

(b) Cell's B alternatives

(c) Cell B is placed

(d) Cell's C alternatives

(e) Cell C is placed

(f) Cell E is over-constrained

(g) Cells D and E replaced

(h) Cell A is placed

Figure 4.5: Reconstruction example

a constraint with cell B bounding both their minimum and their maximum distance must be placed. The possible locations are shown in Figure 4.5d and the final placement of cell C is shown in Figure 4.5e, where another cell, D has also been placed. Next, cell E must be placed. Unfortunately, cell E shares a constraint with both cells A and D, bounding the maximum distance from both cells. Since cells A and D are too far apart, cell E becomes over-constrained, as shown in Figure 4.5f. This is where the reconstruction step takes place. We identify this problem and start reconstructing the placement starting from the problematic cell . All the neighbours of cell E are removed from placement, which in this case are A and D. Next, the most constrained neighbour of cell E is placed, which in this case we assume it is D. The amount of constraints each cell must abide with depends on the number of the cells, with which is shares constraints, are already placed. Next, the problematic cell is placed, which is now possible, since cell A is absent. Finally, all the remaining neighbours of the problematic cell, which in this case is only A are placed again in the placement. In this case, cell A can be placed resulting in the placement of Figure 4.5h. Reconstruction thus, has corrected the unfortunate placement of cell A, which was placed in a bad location due to incomplete information at the time it was being placed. Reconstruction uses the information of the already placed cells and places all the cells in better locations, if this is possible, giving the constructive approach a "global" view of the placement progress. Pseudocode for the reconstruction process for an identified problematic cell is given in Algorithm 4.3.

Algorithm 4.3 starts by adding all the neighbours of the given cell to an array (Line 2). Lines 3 to 7 remove all the neighbours from the placement and store their current locations in case they are needed if the replacement is unsuccessful. Then, the most constrained cell from the neighbours array is placed first to serve as a seed cell (Line 8). Then, the problematic cell is placed (Line 10). At this point the main loop of the algorithm starts (Lines 11 to 14), where all the neighbours are iterated in search for locations that satisfy all the relevant constraints. The function **PlaceWithConstraints** in Line 13 searches for a location for a neighbour cell that does not violate any of its constraints. After all neighbours have been iterated, it is possible that some of them have not been placed, *i.e.* there was no location that satisfied all their constraints. This case is detected in Lines 15 to 22. In this case the neighbour cells are placed in their stored locations (Lines 17 to 19) and the algorithm returns with error (Line 20). If all cells have been successfully placed, then the algorithm returns with success (Line 24). Returning with failure means that no improvement was made locally, but this problematic case can be resolved later, at the next iteration of the main loop, or through the simulated annealing

Algorithm 4.3 - SCPlace Reconstructor Algorithm

```
1: Reconstructor(cell)
2: NeighboursArray ⇐ GetNeighbours(cell)
3: for all neighbour_i ∈ NeighboursArray do
4:    /* Unplace all the neighbours */
5:    placed[neighbour_i] = 0
6:    stored_location[neighbour_i] = location[neighbour_i]
7: end for
8: PlaceMostConstrained (NeighboursArray)
9: /* Place the problematic cell in a random position */
10: Place_Constrained(cell)
11: for all neighbour_i ∈ NeighboursArray do
12:    /* Try to place all neighbours in good positions */
13:    PlaceWithConstraints(neighbour_i)
14: end for
15: for all neighbour_i ∈ NeighboursArray do
16:    if placed[neighbour_i] == 0 then
17:       for all neighbour_i ∈ NeighboursArray do
18:          location[neighbour_i] = stored_location[neighbour_i]
19:       end for
20:       return failure
21:    end if
22: end for
23: /* If all neighbours were successfully placed, return with success */
24: return success
```

shuffling which is performed by the *Perturbator*.

4.7.3 Perturbation

Constructive placement is relative with respect to the placement of the seed cell. A valid placed cluster in many cases possesses rotational symmetry, *i.e.* the cluster still possess valid positions if all cells are rotated. However, these alternative solutions are not automatically taken into account by the cell by cell placement process, and not considering them may prevent the algorithm from either satisfying all the wire bound constraints or escaping a local minimum. The *Perturbation* step explores equivalent, alternative valid placement solutions. To ensure that this exploration moves the placement towards better solutions, the *Perturbation* step has been implemented as a Simulated Annealling algorithm. Essentially, *Perturbation* is an

iterative step encompassed in the constructive framework of SCPlace, making SCPlace a hybrid approach to placement. The iterative nature of *Perturbation* enables SCPlace to overcome the known deficiencies of constructive placement, like inefficient decision-making which may steer the placement in local minima. By iterative optimization, the solution space is searched efficiently, enabling SCPlace to escape any local minima. The *Perturbation*'s cost function is total wirelength, while respecting all wire bounds. Thus, this step improves the existing placement, incorporating a hill-climbing capability, and provides a more global view, compared to local cluster placement. The *Perturbation* step always respects the wire bounds imposed by TSZSA and, thus, maintains the correct-by construction property of the algorithm. Pseudocode for the perturbator is given in Algorithm 4.4.

Algorithm 4.4 - Perturbator Algorithm

```
 1: Perturbator()
 2: total_wl ⇐ WL(Placed)
 3: best_wl ⇐ total_wl
 4: sbest ⇐ current
 5: it ⇐ num_placed * 2
 6: for i = 0 to it do
 7:    cell ⇐ Placed[random]
 8:    Locations ⇐ PossibleLocations(cell)
 9:    new_location ⇐ Locations[random]
10:    stored_location(cell) ⇐ Location(cell)
11:    Location(cell) ⇐ new_location
12:    if (WL(Placed) < total_wl) || (HillClimbing) then
13:       total_wl ⇐ WL(Placed)
14:    else
15:       Location(cell) ⇐ stored_location(cell)
16:    end if
17:    if total_wl < best_wl then
18:       best_wl ⇐ total_wl
19:       sbest ⇐ current
20:    end if
21: end for
22: return sbest
```

The initialization phase of Algorithm 4.4 consists of defining the current and the best wirelength based on the current placement (Lines 2 to 3). The current placement is also saved as the best solution (Line 4) and the number of simulated annealing iterations is defined (Line 5).

The main loop of the algorithm is described in Lines 6 to 21. According to the simulated annealing paradigm, a randomly placed cell is chosen (Line 7) and all the valid locations, *i.e.* locations which do not violate any constraint, for this cell are determined (Line 8). After randomly selecting a new location (Line 9), the current location is stored for backtrace (Line 10). The chosen cell is assigned to the new location (Line 11). If the new placement is better than the previous one, or in the case of hill climbing (Line 12) the new wirelength is stored (Line 13). Otherwise, the previous placement is restored (Line 15). If the new solution is better than all the solutions encountered so far, then this solution is saved as the new best solution (Lines 17 to 20). Algorithm 4.4 returns the best solution found (Line 22).

4.7.4 Finalization

After constructive placement has finished, there may be some cells that were left unplaced because their placement at any location would cause a constraint violation. In order to ensure that the algorithm always provides a full legal placement, even at the cost of a sigma or a mean violation, a greedy placement procedure is called for the unplaced cells. Algorithm 4.5 describes this procedure. The main idea is to traverse all the unplaced cells and place them at locations that yield the minimum violation of their constraints. The working of the algorithm is described below.

Algorithm 4.5 - SCPlace Finalization Algorithm

```
 1: SCPlace Finalization()
 2: UnplacedArray ⇐ GetUnplaced()
 3: for all cell_i ∈ UnplacedArray do
 4:    CellConstraints ⇐ GetConstraints(cell_i)
 5:    ConstraintsArray ⇐ NULL
 6:    for all constraint_i ∈ CellConstraints do
 7:       current_location ⇐ GetLocation(cell)
 8:       ConstraintsArray ⇐ AddConstraint(constraint_i)
 9:       location = MinViolation(cell, ConstraintsArray, current_location)
10:       PlaceCell(cell, location)
11:    end for
12: end for
```

Algorithm 4.5 starts by initializing an array containing all the unplaced cells (Line 2). All unplaced cells are traversed in the outer loop (Lines 3 to 12) and for each cell a location is found

based on its constraints (Lines 6 to 11). The location which yields the minimum constraint violation based on the constraints seen so far is found with function **MinViolation** (Line 9). The algorithm works in a greedy basis, finding the best location for each cell based on its constraints, thus guaranteeing that all the cells will be placed on the layout.

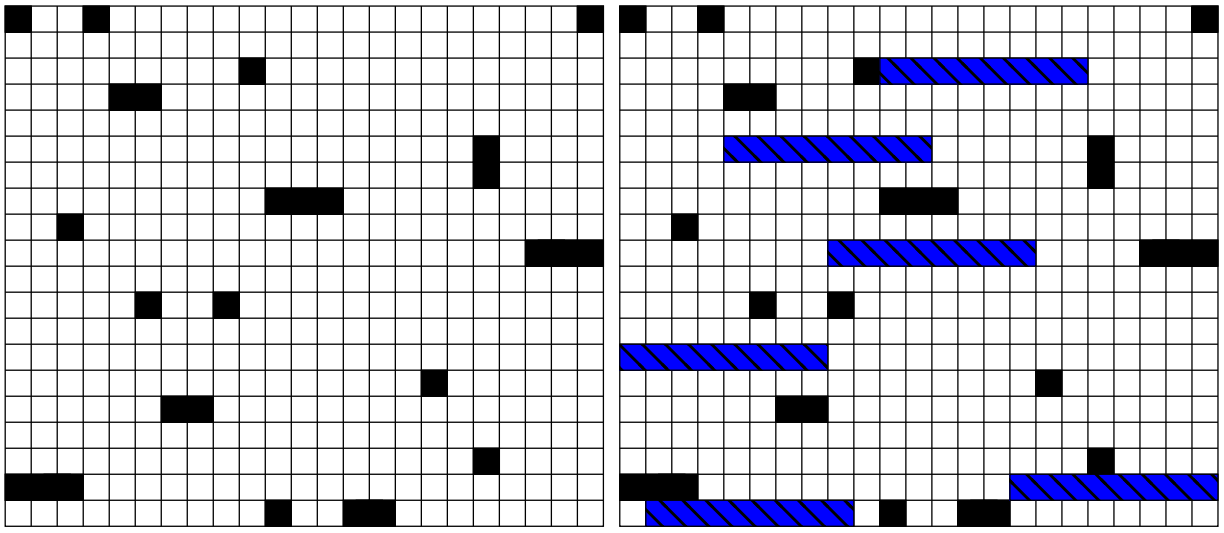
4.7.5 Routability

To ensure that SCPlace produces a routable placement we introduce density screens during the placement process, which ensure both that the local density is kept under control, *i.e.* under a maximum congestion constraint, typically 70%, and that sparsely populated cell regions are utilized. Density screens are exploited at SCPlace's core, during the location selection of seed cells. The density screen size does have a significant effect on the placement. A small number of large density screens may produce large areas with high congestion, whereas a large number of small ones may over-constrain the placer and prevent it from achieving a good solution. A rule of thumb which we used for computing the physical size of the density screens is to make their size proportional to the number of clusters of cells which should lie within close proximity, *i.e.* those which share wire bound constraints.

4.7.6 Legalization

SCPlace guarantees that no overlaps are created at any stage of the placement process. In order to do so, information about the locations of the already placed cells and the free locations must be maintained. The manufacturing grid is used for this purpose. The rows of the layout and the vertical grid lines form virtual rectangles, called sites. Any cell must be aligned to the manufacturing grid, which means that any cell must be aligned with a site. SCPlace maintains a two-dimensional array which holds the status of each site; it may be occupied by part of a cell, or it may be free. This is shown in Figure 4.6a. The occupied sites are shown in black, while the free ones are shown in blank. Any cell, must be placed in adjacent blank sites, otherwise it may overlap with another cell, which is not allowed. Thus, whenever a candidate location is considered for a cell, checks are made to make sure that there is enough free space to accommodate the cell, which means that there are enough empty sites. A subset of the list for the candidate locations for a cell is shown in Figure 4.6b. The cell, spanning multiple sites, can be placed in any of the valid locations shown in Figure 4.6b.

Every time a cell is placed on a new location, all the sites that lie below the cell are marked as occupied. SCPlace can also move cells to different locations in the layout, through the reconstruction and the perturbation functions. In this case, every time a cell is removed from a location, the sites it used to occupy are unmarked so as to be ready to be used by another cell. This means, that there will always be up-to-date information on the locations of all placed cells, enabling SCPlace to maintain its correct-by-construction property with respect to cell overlaps.



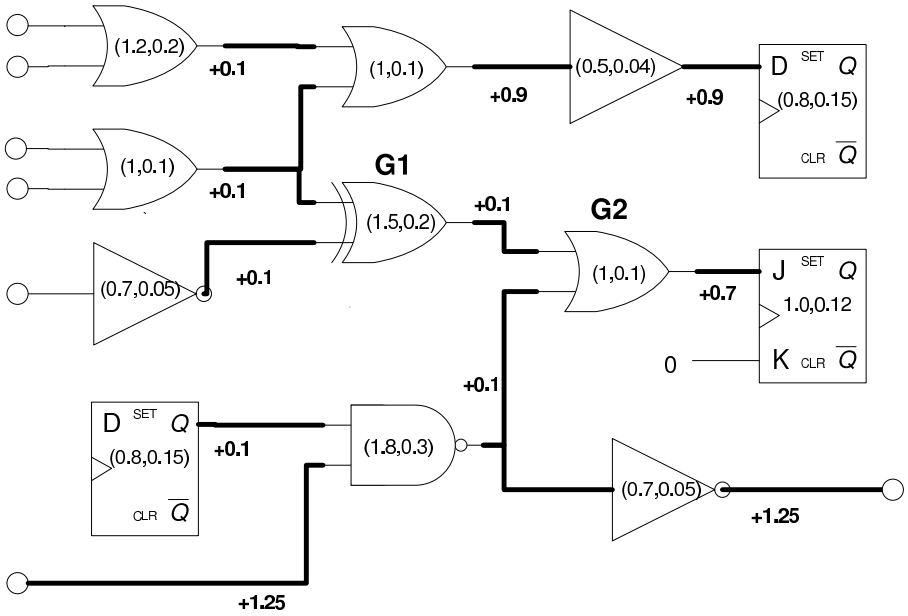
(a) Free and occupied location status (b) Possible placements of a cell

Figure 4.6: Placement of a cell on a valid location

4.7.7 Slack Reassignment

Although the perturbation process can help SCPlace escape local minima through shuffling the locations of cells, more can be done to help correct any unfortunate cases in slack assignment. Since the solution offered by LP is not necessary the only best solution, but is chosen among a set of optimal solutions, it may happen that a cell which is over-constrained by a specific slack assignment may not be so hard to place with an alternative slack assignment. This is illustrated in Figure 4.7.

In this example, the initial slack allocation is shown in Figure 4.9a. Here, gate `G1` has upper bounds, of 0.1, on the delay with its neighbouring gates, which may be hard to meet.

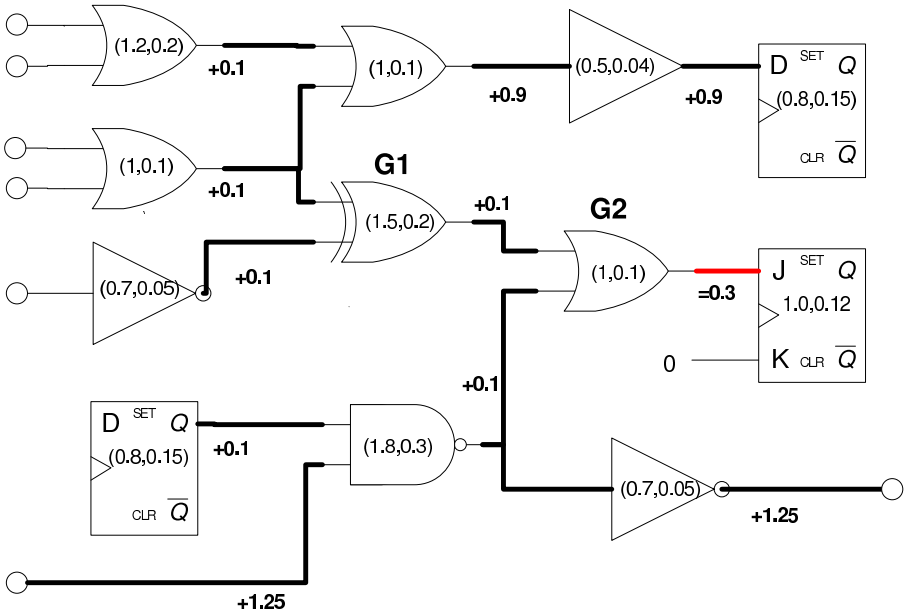


(a) Initial slack assignment

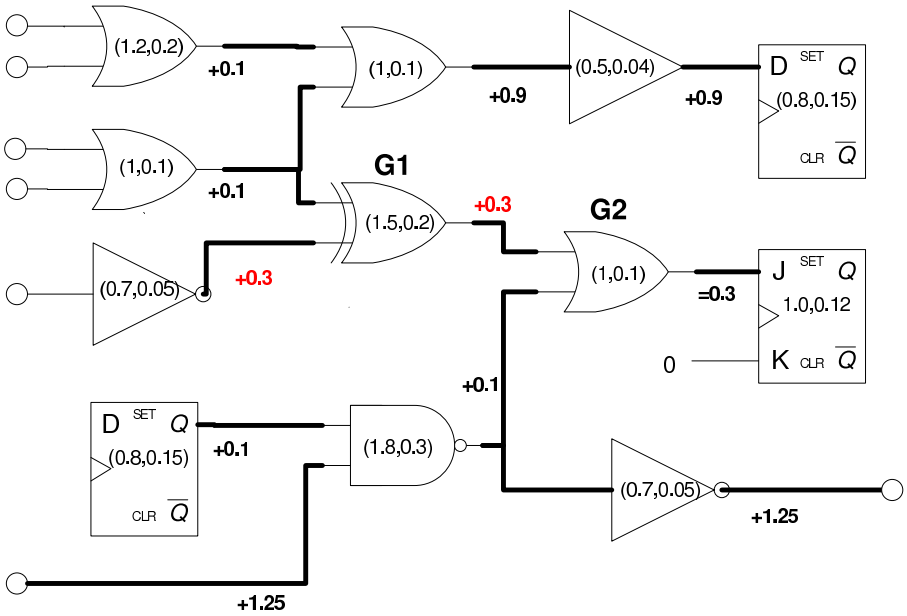
We assume that after some steps of constructive placement, gate `G2` and its fanout gate (the sequential element) are both placed. Thus, the delay of the wire connecting `G2` to its fanout is fixed and meets its upper bound which is 0.7. This is shown in Figure 4.8b. Since gate `G2` has not used all its available slack, it is possible, that through slack reassignment, gate `G1` gets additional slack in order to enhance its chances of being placed successfully. This is shown in Figure 4.7c, where the unused slack from gate `G2` has been distributed to its transitive fanin, providing an alternative, optimal, slack allocation.

If there are cells which have not been placed after one iteration, we formulate the LP problem again, with the same set of constraints as in the initial, unplaced circuit. However, for wires which connect already placed cells we do not allow any optimization; their delays are inserted into the LP problem as fixed values. Thus, LP will try to find alternative optimal solutions with different slack assignments for unfixed nets. In the next iteration, SCPlace will try to place any unplaced cells with different slack constraints, which enables it to search the solution space efficiently and discover solutions that were hidden in the previous iteration.

One important observation is that after any iteration, the number of placed cells will have been increased. This will result in an increased number of fixed nets, and thus fewer unfixed nets. This means that the unknowns in the LP problem will be fewer after each iteration enabling its faster convergence. Thus, it is unlikely that the repeated calls of LP will form a



(b) A wire is fixed



(c) Slacks reassigned

Figure 4.7: SCPlace slack reassignment example

new performance bottleneck for SCPlace.

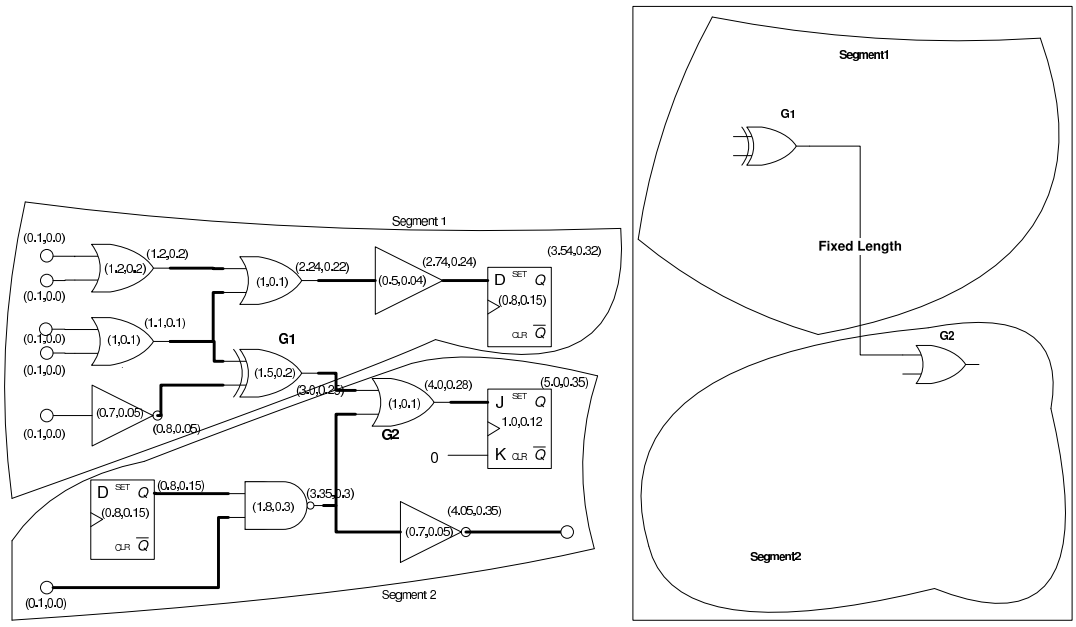
4.8 SCPlace Hierarchical Approach

The core of SCPlace is a relatively fast procedure. A small number of computations is necessary for each cell and each cell is visited only once if it can be assigned to its final location. The runtime bottleneck of SCPlace lies then, at the slack assignment procedure. We have presented the hierarchical approach for TSZSA in Section 3.6. In order for this procedure to be effective, it needs to be incorporated into the framework of SCPlace. In this section, we describe how SCPlace utilizes the hierarchical approach presented in Section 3.6.

As described in Section 3.6, we apply recursive bisection in order to divide the slack assignment problem for the whole circuit into smaller problems with the same characteristics as the initial problem. At every bisection, a number of nets will span between two segments. The length of these nets cannot be optimized by LP, as they cannot be modeled in either of the two resulting LP formulations for the two separate segments. This is the point of integration of the hierarchical approach with SCPlace. SCPlace, following its constructive paradigm, places first the cells forming the nets which span between the two segments. With this action, SCPlace solves two problems at the same time. First, the length of this wire is fixed and does not need to be formulated into any LP problem and second, a convenient choice for seed cells is made. If this fixed length for this wire is proven to cause problems later in the constructive placement process, then it can always be altered through the *Reconstruction*, or the *Perturbation* processes.

Figure **??** shows how the example of Section 3.6 is integrated into placement. After one bisection, two segments are formed with one net spanning between the two segments. Gates `G1` and `G2` are placed in locations which guarantee that the length of their connection meets its bound. These cells are used then as seed cells for the constructive placement which will happen separately in `Segment1` and `Segment2`. It should be noted that the regions for `Segment1` and `Segment2` in Figure 4.8b are only indicative and do not imply any strict boundary for the cells connecting to each segment. However, due to the presence of seed cells, the other cells belonging to the same segments are likely to be placed in these regions.

In the next chapter, we present our novel leakage recovery flow, which uses the TSZSA's statistical bounds and can be applied directly after SCPlace.



(a) Circuit-level bisection (b) Seed cells are placed

Figure 4.8: Hierarchical SCPlace

Chapter 5

Post-Placement Statistical Leakage Optimization

In this chapter, we present our implementation of a combined statistical leakage and area recovery flow [49], which preserves the achieved statistical timing yield. The flow operates post-placement, on a layout which has met a combined statistical constraint, *i.e.* a (mean, sigma) pair, and recovers leakage through gate resizing, while aiming to keep the timing yield constant. The algorithm is based on our TSZSA statistical slack assignment, described in Section 3.5 whereby the required timing yield constraint can be mapped to delay bounds for circuit gates and wires. The latter are exploited for leakage recovery by selecting candidate gates for downsizing, while preserving timing yield. Incremental SSTA within the leakage optimization loop enables fast and accurate size selection per candidate gate.

5.1 Statistical Leakage Optimization

EDA tools, like placers, which use global timing constraints, *i.e.* bounds on the maximum delay for the whole circuit, do not try to make all paths of the circuit exactly meet the constraints. However, all paths must have delays better than the constraint, otherwise the circuit would be failing. This leads to the observation that there will be paths that are not critical, *i.e.* paths which can afford some extra delay without violating the timing constraint. In the case of statistical timing constraints, the same intuition applies; there will be some paths, which can afford an increase in their mean or sigma of delay without violating any statistical constraint.

In the context of a power optimization process, *e.g.* a leakage minimization algorithm, this means that there are some gates that can be replaced with slower gates, having better leakage properties not being too slow to violate the timing constraints.

We use this intuition in order to optimize the statistical leakage of a circuit. We identify gates that can be replaced with slower ones by calculating their statistical slack after placement. Our TSZSA algorithm fits well with this idea. By setting the mean and the sigma of the finalized delay, after placement, as statistical targets, TSZSA can derive a slack allocation that, translates the slack of each path into slack for each gate. If these slacks were applied to the gates, then all paths would be made critical, matching exactly the statistical delay constraint. Having derived the slack for each gate, then a greedy optimization algorithm can be applied, which replaces each gate with positive slack with a slower gate which emits less leakage current. The important property of TSZSA slack allocation is that the slacks assigned to each gate do not have dependencies among them. This means that consuming all the available slack for a gate does not affect in any way the slack assigned to another gate, leaving room for the application of a greedy approach. Since the set of alternative cells is finite, it is unlikely that the slack of each gate will be spent in its entirety after a gate substitution. In order to make effective use of the remaining slack, we employ incremental SSTA, which updates the timing and slack of the cells, close to the one substituted. We also enhance the optimization potential of our flow by performing slack re-assignment after one full pass of the circuit and starting over again. The unused slack may be distributed to other cells which might be further optimized using the additional slack.

An algorithm of this type could also be used to trade-off delay for leakage. By setting larger mean delay than the one reported after placement, even the most critical gates could afford some extra delay allowing for more optimization. This way, a designer willing to pay a small penalty on the delay, could get a circuit emitting less leakage current.

5.2 Statistical Leakage Optimization Requirements

There is a set of requirements for our statistical leakage optimization flow to be effective. These are models for statistical timing and leakage, physical information, *i.e.* a placement and wire modeling, a set of constraints and a library from which alternative gates for each gate can be extracted.

5.2.1 Physical Information

Our leakage flow operates in in the post-placement phase. This means that the circuit must have already been placed and all cells have valid locations on the layout. Additionally, a model to estimate wire delays is needed. This can be any model employed by contemporary physical tools, prior to routing. By default, our flow estimates wire delays by using the bounding-box technique and Elmore delay assumptions, in a similar way as SCPlace does (c.f. Section 4.3).

5.2.2 Timing Analysis and Leakage Model

Our flow aims at optimizing for both statistical leakage and statistical timing. Thus, models for SSTA and statistical leakage analysis are needed. For SSTA, we employ our SSTA engine, described in Section 3.1.

It has been shown [65], that the leakage of a gate, in the presence of variations, can be approximated by a log-normal distribution. The properties of log-normal distributions are explained in Appendix C.2. Log-normal distributions correlate well with the behaviour of leakage current due to the fact that leakage current exhibits an exponential dependence on the gate length. We follow this log-normal model, and model the leakage power consumption of each circuit gate as a log-normal distribution.

A log-normal distribution can be adequately expressed by its first two moments, *i.e.* the mean and variance of the corresponding normal distribution. We evaluate the log-normal leakage distribution parameters of each standard cell, by analyzing the cell's leakage consumption per available corner, temperature and voltage, and then fit and extrapolate these leakage values onto a log-normal distribution, by using the related normal distribution, as follows.

Given a few random samples, x_i, of a log-normal distribution, random samples $ln(x_i)$ will map onto a normal distribution whose μ and σ can be easily evaluated [42]. Thus, the first two moments of the corresponding log-normal distribution can be calculated using the formulas below:

$$\begin{aligned} E(X) &= e^{\mu+\frac{\sigma^2}{2}} \\ VAR(X) &= (e^{\sigma^2-1})(e^{2\mu+\sigma^2}) \end{aligned} \tag{5.1}$$

Figure 5.1 shows the extrapolated statistical leakage PDF for a library cell, whose reported leakage values at five different corners are {1109, 6075, 7294, 15202, 34948, 43531} (pW).

Thus, given the statistical leakage power consumption for all library cells, the total leakage

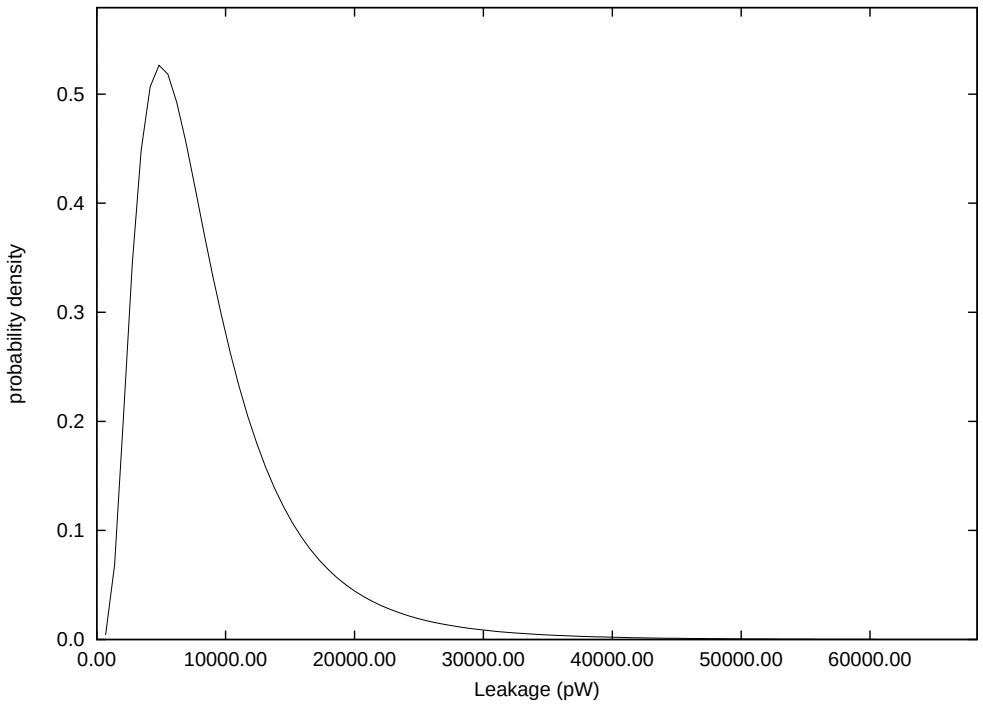


Figure 5.1: Statistical leakage extrapolated PDF example

power consumption may be calculated by summing the current log-normals of all of the circuit's constituent cells. The sum of two log-normal random variables is generally not log-normal, but as has been shown [11], that the first two moments of $Z \sim lnN(\mu_z, \sigma_z) = X \sim lnN(\mu_x, \sigma_x) + Y \sim lnN(\mu_y, \sigma_y)$ may be reasonably approximated to the first two moments of a log-normal, by using equations 5.2.

$$\begin{aligned} \sigma_z^2 &= ln\left[\frac{e^{2\mu_x+\sigma_x^2}\left(e^{\sigma_x^2}-1\right)+e^{2\mu_y+\sigma_y^2}\left(e^{\sigma_y^2}-1\right)}{\left(e^{\mu_x+\frac{\sigma_x^2}{2}}+e^{\mu_y+\frac{\sigma_y^2}{2}}\right)^2}+1\right] \\ \mu_z &= ln\left[e^{\mu_x+\frac{\sigma_x^2}{2}}+e^{\mu_y+\frac{\sigma_y^2}{2}}\right]-\frac{\sigma_z^2}{2} \end{aligned} \tag{5.2}$$

5.2.3 Timing and Leakage Constraints

Any optimization tool needs a set of constraints in order to be able to steer the optimization to the direction the designer wants. Our leakage flow can accept constraints for both timing and leakage, although they are not strictly required. Timing constraints need to be of statistical

nature. This means that they must define a maximum on the mean, on the sigma, or both of the circuit's statistical delay. In the absence of any timing constraints, our flow sets the initial statistical delay of the circuit as the statistical timing constraint. In this case, it is assumed that our flow is expected to optimize for leakage without any negative impact on the delay of the circuit.

A constraint on leakage can guide our flow, as to how much optimization is required. This can be statistical in nature, *i.e.* can be on any of the first two moments of the log-normal distribution describing the total statistical leakage. The more common constraint would be on the first moment, *i.e.* the expected value of leakage. In the absence of any constraints on leakage, our flow assumes that the best optimization possible is required. In this case, we optimize for leakage as aggressively as possible, making sure that the statistical delay constraint is not violated.

5.2.4 Gate Substitution

Our leakage flow operates as a gate resizing/substitution flow. It replaces each gate, which can afford additional delay without violating the timing constraint, by slower gates which are less leaky. Thus, a set of alternative, equivalent gates is needed, for each gate, in order for this substitution to take place. A set of alternative gates is commonly provided from a typical technology library as gates with different sizes and drive strengths. Additionally, different gates with the same functionality (*e.g.* the carry-out of a two-bit adder is equivalent to an *AND* gate) can also be used.

5.3 Statistical Leakage Optimization Interface

Our flow operates directly after placement has been completed. Thus, it must be able to accept industry-standard formats from a placer and create the same files for use in the next step of the fabrication flow.

First, a description of the technology library must be given in standard library description formats, *LEF* and *LIB*. We use the *LEF* file to extract physical information, like the unit capacitance and resistance of wires. We use the *LIB* file to extract timing and leakage information for the standard cells. For statistical optimization, a set of *LIB* files must be provided, each corresponding to a different operating corner, in order for our flow to derive

the statistical behaviour of single circuit elements as described in Sections 3.1 and 5.2.2. In case only one *LIB* file is provided, our flow essentially becomes non-statistical in nature, performing static timing analysis and optimizing for non-statistical leakage. Placement must be provided in the industry-standard *DEF* format. The circuit description must be provided by the industry-standard *Verilog* description language.

In the next section we describe in detail the optimization objectives that can be applied to our leakage optimization flow.

5.4 Optimization Objectives

Although certain flows can efficiently increase either (i) the timing yield or (ii) the leakage yield, no approach has been demonstrated which, while keeping timing yield almost fixed, can perform leakage recovery, which would be ideal for today's Deep-Sub Micron (DSM) designs. In many cases, contemporary industrial tools, while optimizing for leakage can incur significant timing yield penalties. Thus, our leakage optimization flow works on a set of objectives and constraints which aim to optimize leakage and area without violating any design rules or any timing constraints. Below we describe in detail the objectives and the constraint set.

- **Statistical Timing Constraints.** We aim to preserve the initial timing of the circuit, thus our leakage flow accepts statistical timing constraints. These, in the general case, refer to the timing of the circuit before optimization. This means that the (μ, σ) constraint will be the actual (μ, σ) of the circuit's timing. We then derive bounds in order to preserve this timing, which means that we identify the gates which can afford to be slowed by a small amount of delay without affecting the overall timing of the circuit. Our leakage flow also accepts a relaxation of the circuit's timing. This means that the timing constraint will allow the circuit to run somewhat slower, or to have an overall delay with greater sigma. In this case, our flow tradeoffs between delay and leakage, using the extra delay allowed by the constraints in order to perform more aggressive leakage optimization.

- **Statistical Leakage Constraints.** In the general case, our leakage optimization flow does not need any constraint on the leakage. It derives the maximum amount of optimization allowed for each gate and uses it to the maximum. Thus, it aims at optimizing leakage in the most aggressive possible way. However, our flow can accept a constraint on the statistical leakage. If this constraint is tighter than the minimum leakage possible

for the circuit, then our flow will optimize leakage as much as possible and return with a leakage violation, as referred to the constraint. If the constraint is less tight than the minimum possible leakage, then our flow will keep optimizing until leakage meets the constraint. This means that not all gates will be optimized to the maximum, allowing them not to become too timing-critical.

- **Don't touch constraints.** It is not uncommon that during post-placement optimization, the designer wants to forbid optimization tools from optimizing certain gates or specific modules. This is commonly done by specifying *don't touch* gates, *i.e.* gates that the designer does not want to be altered in any way. Our leakage optimization flow accepts a list of gates that should not be optimized. Although these gates may be found to have positive slack, *i.e.* they could have been optimized for leakage, our flow respects the *don't touch* constraint. Currently we do not support the transfer of slack from *don't touch* gates to other gates to allow more aggressive optimization. However, the TSZSA algorithm, which assigns the statistical slacks to gates can be extended to handle this case too. This will lead to assigning more slack to the non-*don't touch* gates allowing more leakage recovery, if this is possible.

- **Density Screens.** Our leakage flow guarantees congestion bounds by using density screens throughout the layout. Horizontal and vertical, virtual lines, divide the layout area into regions. The density in each region can be defined by the designer. A typical value for density is 70% for each region and 65% for the whole circuit. Although a post-placement flow, which does not move standard cells cannot affect density too much, it still can, since it changes standard cells, and thus changes the dimensions of standard cells. Our leakage flow checks before each cell is replaced by an alternative cell, if the change incurs a violation in the density of the region it belongs to. If it does so, our flow rejects this substitution. By ensuring that density is not violated after the post-placement optimization, we guarantee that our flow will not render unroutable a placement which was initially routable.

- **Overlap Constraints.** Our leakage optimization flow does not create any overlap violations. Overlaps can easily occur during post-placement optimization which changes standard cells. Replacing a standard cell with an alternative, can mean that the width of the cell is changed, which can potentially cause an overlap with its adjacent cells. Our

flow maintains at all times up-to-date information about the free space around all cells making sure that changing the width of a standard cell does not create any overlaps. Should one replacement cause any overlaps, then it is rejected regardless of the amount of leakage it might recover.

- **Design-rule Constraints.** Additional second-order design rules are satisfied by our leakage flow, provided that they were satisfied in the initial placement. This means that our flow does not have the ability to correct any design-rule violations that were present in the initial placement, but guarantees that it will not introduce any new violations. The most important rules that our flow guarantees is that all standard cells, after they are replaced with alternatives, are aligned with the manufacturing grid. This ensures that no movement is necessary after optimization, which could potentially jeopardize any strict timing or density constraints.

5.5 Leakage Optimization Flow

Figure 5.2 shows our implemented flow, which includes the TSZSA and LP slack reassignment steps.

The first step of the optimization process is derive wire bounds for the (mean, sigma) constraint, which is performed by running TSZSA on the placed netlist. Since we aim to preserve timing yield, *i.e.* fix the mean and sigma values, we regard the SSTA results as hard constraints for both mean and sigma.

The next step is to calculate the slack of each circuit gate, *i.e.* the amount of delay each gate can afford without violating either the mean or the sigma constraint for the overall circuit. This task is performed by the application of TSZSA and the LP slack reassignment. The TSZSA wire bounds are forwarded to the LP slack reassignment, which identifies a slack assignment for all wires. The slack for a gate is then set to the minimum slack of the wires, the gate's output drives. Even if all gates are downsized in a way that all their slack is consumed, the following properties will hold: (i) the mean of the circuit's delay will not be violated, (ii) the sigma of delay at each gate will not exceed its initial value, resulting in (iii) the sigma of the circuit's delay will not increase.

This last property is the key contribution of TSZSA to our flow, since without the relative constraints guaranteeing the propagation of the desired sigma, the slack assignment would not

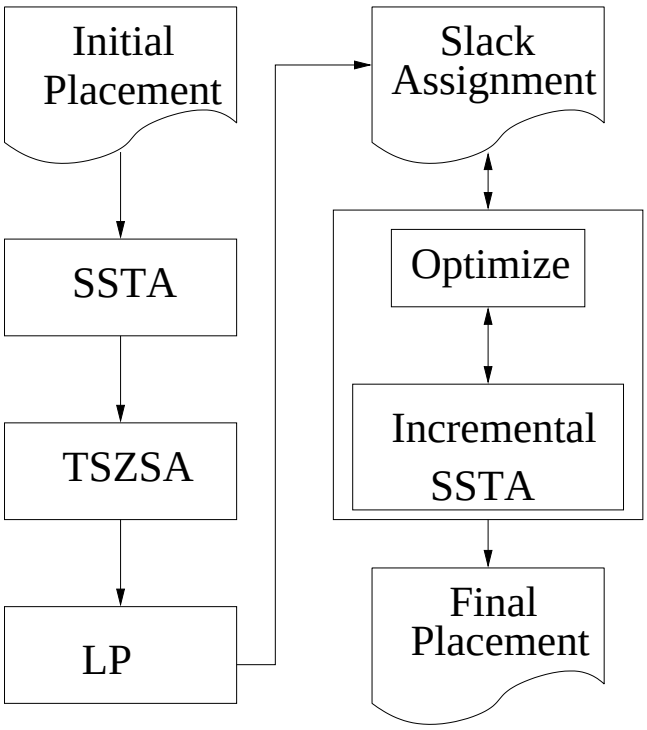


Figure 5.2: Leakage reduction flow

guarantee any bound on sigma. Using our approach, we have divided the problem of recovering leakage, while meeting a statistical delay constraint into two separate problems, which are solved by two independent algorithms. TSZSA ensures that the mean and sigma is maintained, and the gate resize algorithm recovers leakage using the sigma-safe slack allocation from TSZSA.

The gate sizing process then examines all gates with positive slack, in breadth-first-manner starting from the circuit's inputs, and attempts to downsize them, trading off the available excess slack for leakage. Algorithm 5.1 illustrates the steps performed for each gate.

As shown in Algorithm 5.1, for each gate, the first step is to extract the transitive fanin and transitive fanout cone originating from the selected gate (Line 3). These cones are used in the incremental SSTA step. For each gate, all alternative gates are identified and stored in decreasing order of size in queue (Lines 4 to 5). Thus, the last gate in this queue will correspond to the most aggressive downsizing. Moreover, as smaller gates also exhibit larger delay, if one gate fails the delay check, then all remaining smaller gates will also fail the check, thus they do not need to be explicitly checked.

The core of the sizing algorithm is Lines 6 to 16. We selectively replace the selected gate

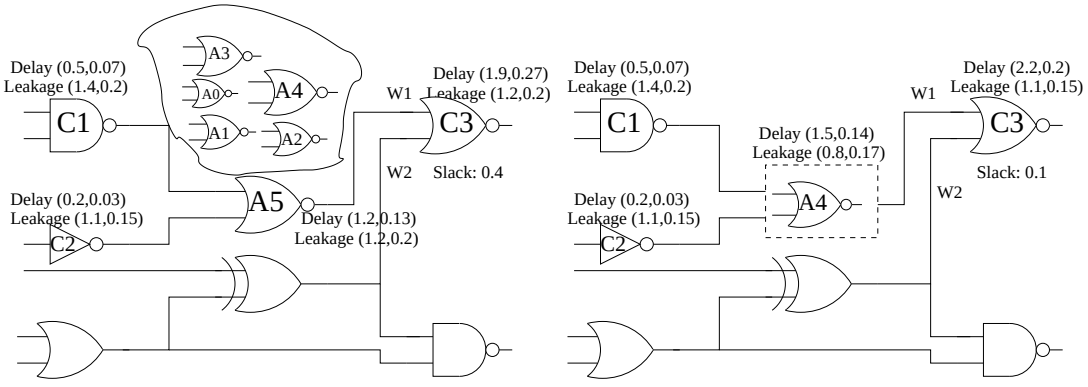
Algorithm 5.1 - Gate Sizing Algorithm

```
1: Gate Sizing(netlist)
2: for all (gate_i ∈ netlist) do
3:    cone ⇐ FindCone(gate_i)
4:    alternatives ⇐ FindAlternatives(gate_i)
5:    Sort(alternatives)
6:    for all (alternative_j ∈ alternatives) do
7:       ReplaceGate(gate_i, alternative_j)
8:       IncSSTA(cone)
9:       for all gate_cone ∈ cone do
10:         if violated(delay(gate_cone)) then
11:            RestoreSize
12:            break
13:         end if
14:      end for
15:   end for
16: end for
17: return
```

by all alternatives, one at a time (Line 7). After each trial, incremental SSTA is performed on the selected gate's cone (Line 8). The delay of each gate in the cone is checked against the delay constraints. If one gate fails, then the trial is considered to be unsuccessful, the selected gate is restored to its previous size, *i.e.* the last successful resize, and the algorithm resumes operation for the next gate with positive slack (Lines 9 to 14). When the algorithm returns, all gates will have been examined for being replaced with smaller gates. The successful trials will have resulted in on-site replacement, resulting in leakage recovery.

Figure 5.3 shows an example of the sizing process. As shown in Figure 5.3a, when we examine gate $A5$, we have five alternative gates to consider, $A0 \ldots A4$. Since all gates are sorted by their area, we start from the largest alternative, *i.e* $A4$. In Figure 5.3b, we have replaced $A5$ with $A4$ and now we perform incremental SSTA from the replaced gate, one level backwards and one level forwards. The cone on which incremental SSTA is performed contains the gates $C1, C2, C3$ and $A4$. This trial resize is successful, as the mean delay of $C3$ is increased by a smaller amount than its slack. Additionally, the small increase in the sigma of $A4$ delay's, is not propagated to the output of gate $C3$. If this increase in sigma could be propagated through $C3$, then TSZSA would have enforced a smaller slack on gate $A5$, thus making the selection of gate $A4$ unacceptable. The algorithm proceeds by examining all alternative gates

in order, until one sizing causes a violation in either the mean or the sigma of delay, or all alternatives are examined. In the example of Figure 5.3, gate $A3$ may have caused a sigma violation, thus the accepted solution is the replacement of $A5$ with $A4$. The resulting sizing scheme, results in the same sigma for the delay, and a trade of available slack for a reduction in the distribution of leakage consumption ((1.2,0.2) down to (0.8,0.17)).



(a) Sizing with alternative gates (b) Sizing trial

Figure 5.3: Re-Sizing example

The optimization process ends when all gates with positive slack have been downsized. The optimization flow is of $O(n.p)$ complexity, where n is the number of gates and p is the number of alternative sizes for each gate. Each gate is visited only once and at most p trials are made for each gate.

5.6 Optimization Flow Details

In this section we discuss implementation details of the flow. Specifically, we describe the slack re-assignment process, how the alternative gates are selected and sorted, why is incremental SSTA important and how it is performed, how the best candidate gate for replacement is chosen and why slack reassignment is beneficial for finding the best optimization.

5.6.1 Statistical Slack Assignment

Our statistical leakage optimization works by downsizing gates which can afford an amount of delay without violating the statistical timing constraints. In order to derive the maximum amount of delay each gate can afford, a slack assignment procedure is needed. We use the

TSZSA algorithm, introduced in Section 3.4.4, which can derive a statistical slack assignment which guarantees a statistical (μ, σ) delay at the circuit's outputs.

Our leakage optimization flow can operate under two modes. The first is to optimize slack maintaining the current statistical timing and the second is to optimize slack under a relaxed statistical timing constraint. The first mode corresponds to leakage recovery in a transparent way referred to timing. The second mode corresponds to trading off timing for leakage. The choice of modes lies with the designer and the specific application of the circuit with respect to the criticality of timing and leakage consumption.

In the first mode, TSZSA accepts as (μ, σ) constraints the (μ, σ) delay that the initial placement has. Using the approach presented in 3.4.4, TSZSA, in conjunction with the LP formulation, finds the timing slack of each gate so that the initial statistical timing of the circuit is not affected. If then all gates are delayed by their designated statistical slack, then all gates will become timing-critical, making in turn all paths timing-critical. This means that all paths of the circuit will have a delay approximately matching the (μ, σ) of the timing constraint. However, no timing constraints will be introduced. Our flow will utilize the slacks indicated by TSZSA to make all gates as close to timing-critical as possible, recovering leakage in the process.

In the second mode, TSZSA accepts a (μ', σ') constraint with μ' being greater than the μ of the initial placement's delay, or σ' being greater, or both. This allows greater flexibility to TSZSA. In this case, all gates can have positive slack, as all paths have smaller delay than the one suggested by the constraint. This will lead to more slack being assigned to gates, which will allow for more aggressive optimization to recover more leakage. Thus, the designer is provided with a mechanism to tradeoff between delay and leakage and to choose the best set of constraints which suit the desired application in the best way.

5.6.2 Gate Sorting

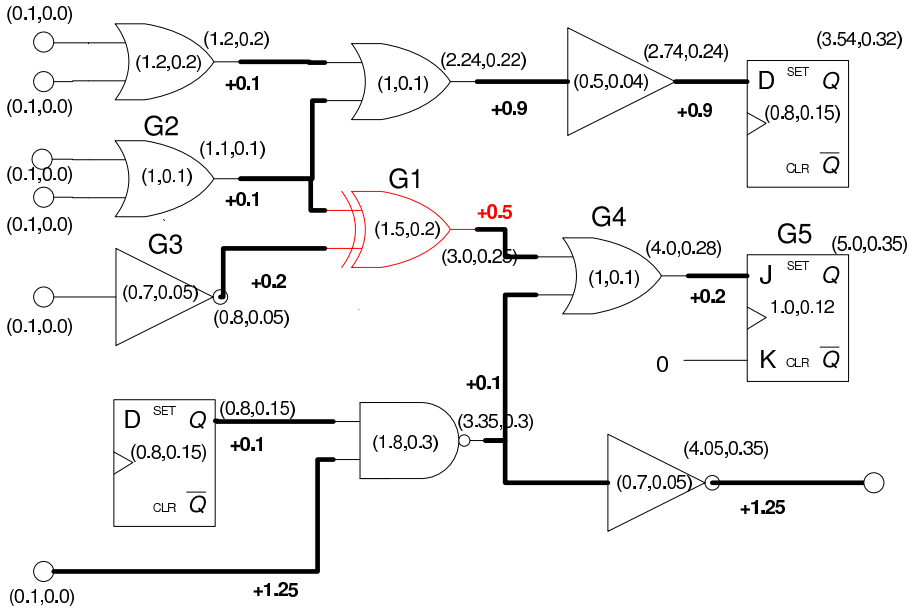
In order to substitute a gate with another one, the set of gates with exactly the same functionality must be found first. To do so efficiently, we maintain a list of equivalent gates for all library gates. For example, an *AND* gate will be available with different cells at different drive strengths. Additionally, other gates which can function as an *AND* gate, with specific conditions on their inputs, will also be listed. An example would be a half adder with the *carry out* having the same function as an *AND* gate.

Every time a gate is found to have positive slack, its alternative gates are retrieved. Then, the alternatives are sorted with respect to their delay, the fastest taking the first position in the list. All the gates are then tried in the order of the sorted list to replace the given gate. The first option will give a fast gate with high leakage, the second a little slower gate with less leakage and the last gate will give a slow gate with small leakage. All gates are tried until a gate violates the slack assigned to the gate to be replaced. When this happens, the trials stop, as the next gate in the list will be even slower violating the slack further. Thus, sorting the gates before the trials eliminates the need to try gates which are not likely to meet the timing constraint.

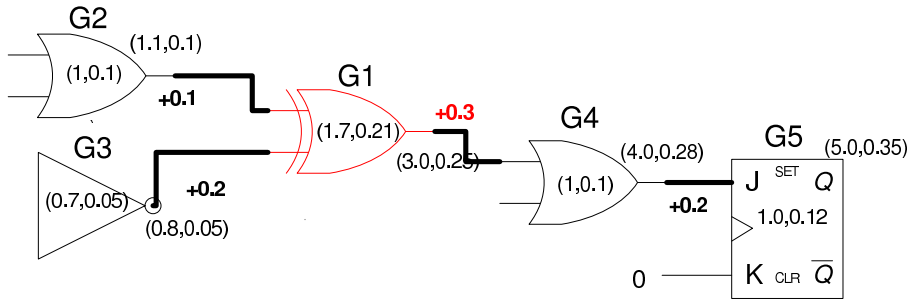
Immediately after a substitute gate fails the timing constraint, we have the best candidate for replacement. This is the gate which was tried last and met the timing constraint. In the worse case, where no substitute gate with less leakage is fast enough to meet the timing constraint, this will be the initial gate. This is an extreme case of slack that will be left unused. A more common case will happen when the best substitute gate consumes some of the slack assigned, but not all of it. This will happen very often, as it is unlikely that a substitute gate will be found, whose delay meets exactly the timing constraint. In this case, the unused slack cannot be used by any other gate, as it is only assigned to the gate that has just been optimized. This highlights the need to incrementally update the timing graph and readjust slacks in order to utilize as much slack as possible.

5.6.3 Incremental SSTA

We use incremental SSTA after every replacement in order to keep the timing graph up-to-date and to readjust slacks in order to improve efficiency. This is illustrated in Figure 5.4. In Figure 5.7a, gate `G1` is set to be downsized, an action which will improve its leakage. It is very likely that the best downsize will not utilize all the slack assigned to `G1`, which currently is 0.5ns. Indeed, the best downsize option uses only 0.2 of its slack, as shown in Figure 5.6b. Any other downsize for this gate would cause a timing slack violation, thus no further optimization is possible. We perform incremental SSTA at this point, updating the timing information on gates `G2`, `G3`, `G4` and `G5`. We update the timing graph one level backwards and two level forwards. This is shown in Figure 5.5c After the timing graph has been updated, the unused slack of gate `G1` is distributed to its neighbouring gates in a greedy way. Currently, it is distributed evenly to the gates whose timing has been updated by incremental SSTA. Thus, gates `G2`, `G3`, `G4` and



(a) G1 set to be downsized

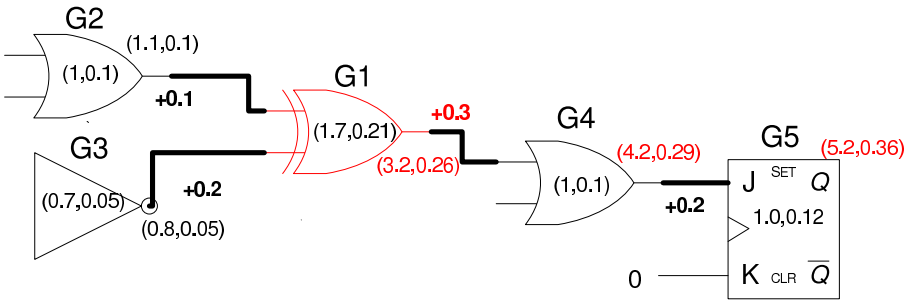


(b) Unused slack after downsize

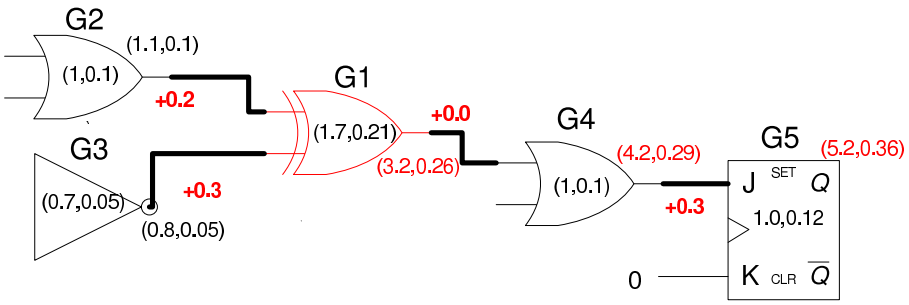
`G5` can benefit from the unused slack from gate `G1` in order to be optimized more aggressively. The new slack allocation is shown in Figure 5.4d.

5.6.4 Slack Reassignment

After one full iteration of optimization, all gates will have been optimized in an optimal way, given their initial slack assignment. However, as it is very unlikely that for all gates, substitute gates which can utilize all their assigned slack can be found, there will be some unused slack spread throughout the circuit. In this case, the unused slack at each individual gate cannot be used, as there are no suitable candidate gates for substitution. However, it is possible that by reassigning slacks there may be gates which could be optimized further, given that they are



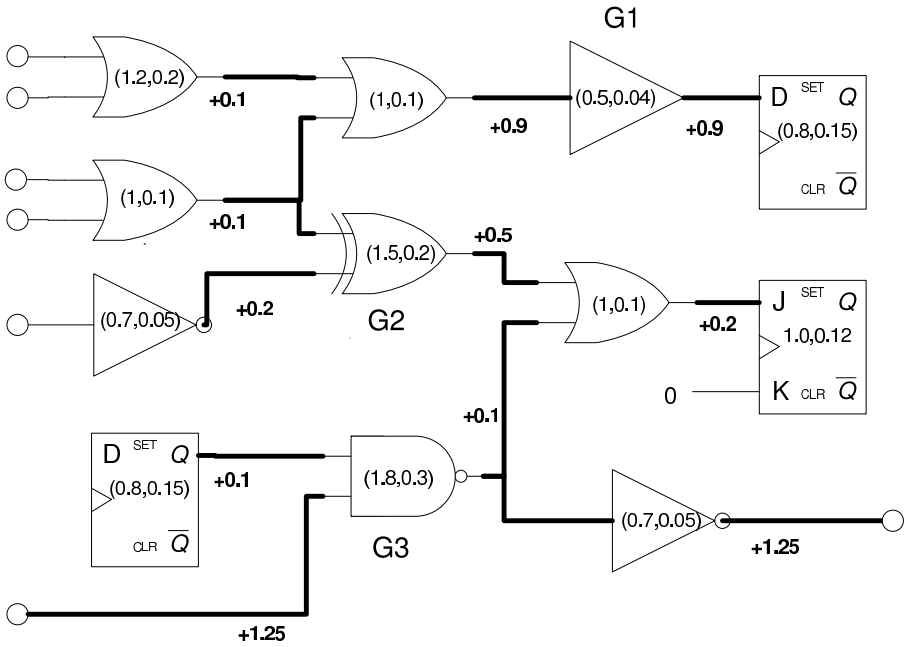
(c) Incremental SSTA



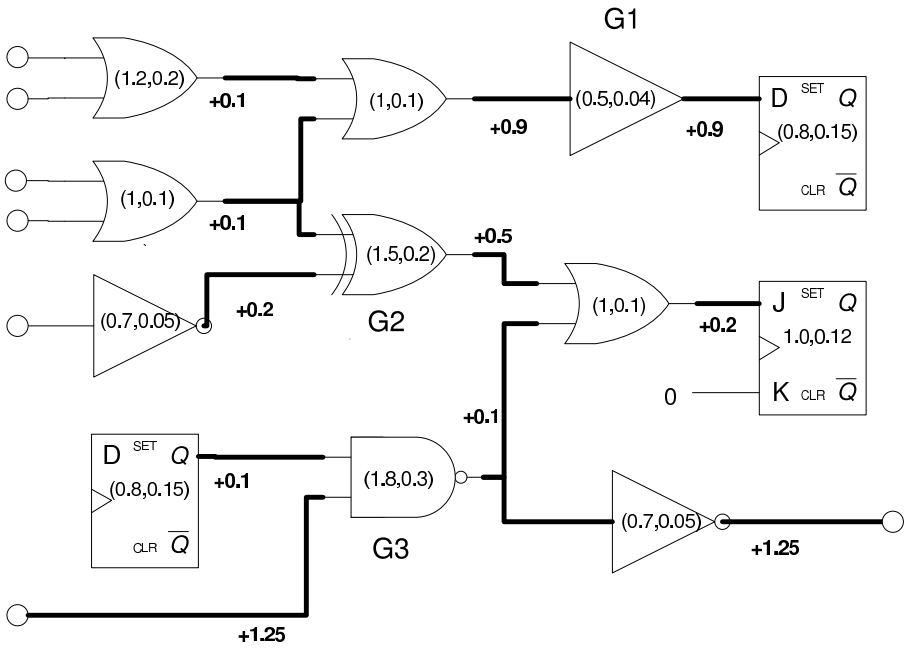
(d) Unused slack distributed

Figure 5.4: Distributing unused slack with incremental SSTA

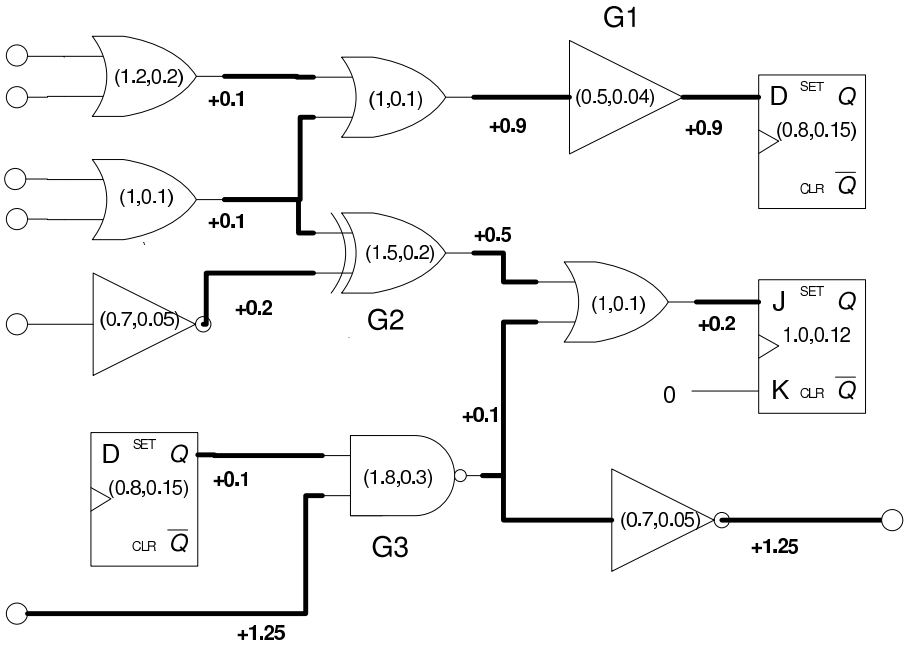
assigned more slack. This slack can be found by removing it from gates which are very unlikely to be further optimized. This is illustrated in Figure 5.5. In Figure 5.8a we assume that only gates `G1`, `G2` and `G3` can be downsized. In Figure 5.7b they have been downsized and their timing has been updated. This has caused an update on the slack of these gates and on the slack of their neighbouring gates, which is illustrated in Figure 5.6c. At this point, one of the inputs of gate `G3` has slack of 0.9ns, while the other input has slack of 0.0ns. This prevents gate `G5` from being optimized. Additionally, gate `G2` has slack of +0.3ns which cannot be used, as no suitable substitute gate exists for gate `G2`. However, gate `G4` does not have any slack and cannot be downsized. This example illustrates how, after one full optimization pass, the final slack distribution may be unfortunate. At this point we perform slack allocation again, in order to find another optimal solution. We formulate the LP problem which finds optimal slack allocations again, by adding artificial bounds on the slack that can be assigned to gates that are very unlikely to be optimized at another pass. This will lead to assigning more slacks to other gates, for which it is more likely that suitable substitute gates can be found. The new slack allocation is shown in Figure 5.5d, where gates `G4` and `G5` have enough positive slack. If there are suitable candidate gates, they can be downsized resulting in more leakage optimization.



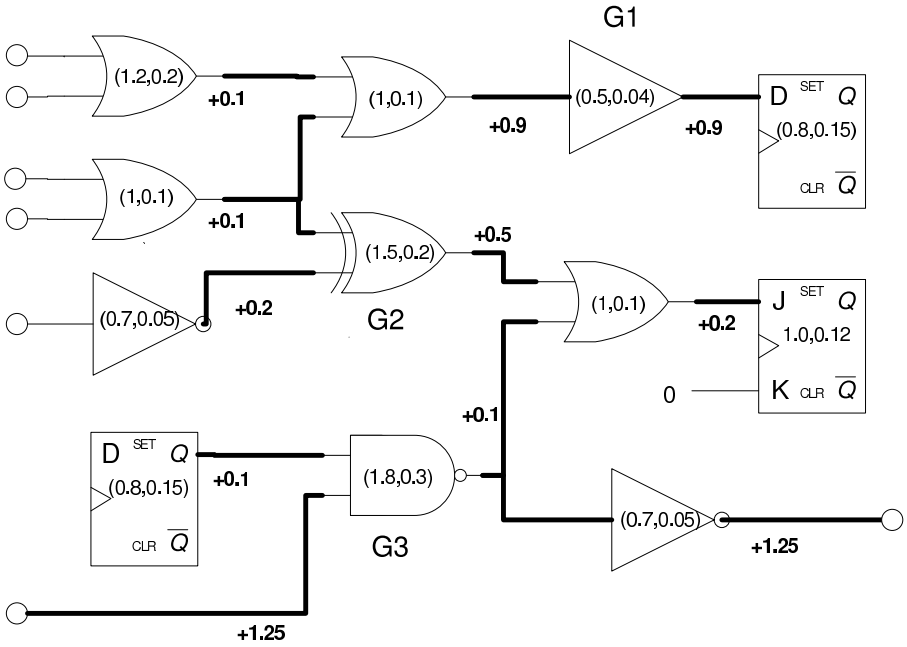
(a) G1, G2 and G3 set to be downsized



(b) G1, G2 and G3 have been downsized



(c) Slack has been updated



(d) Slack has been redistributed

Figure 5.5: Re-distributing slack after one full optimization iteration

5.7 Routability and Legalization

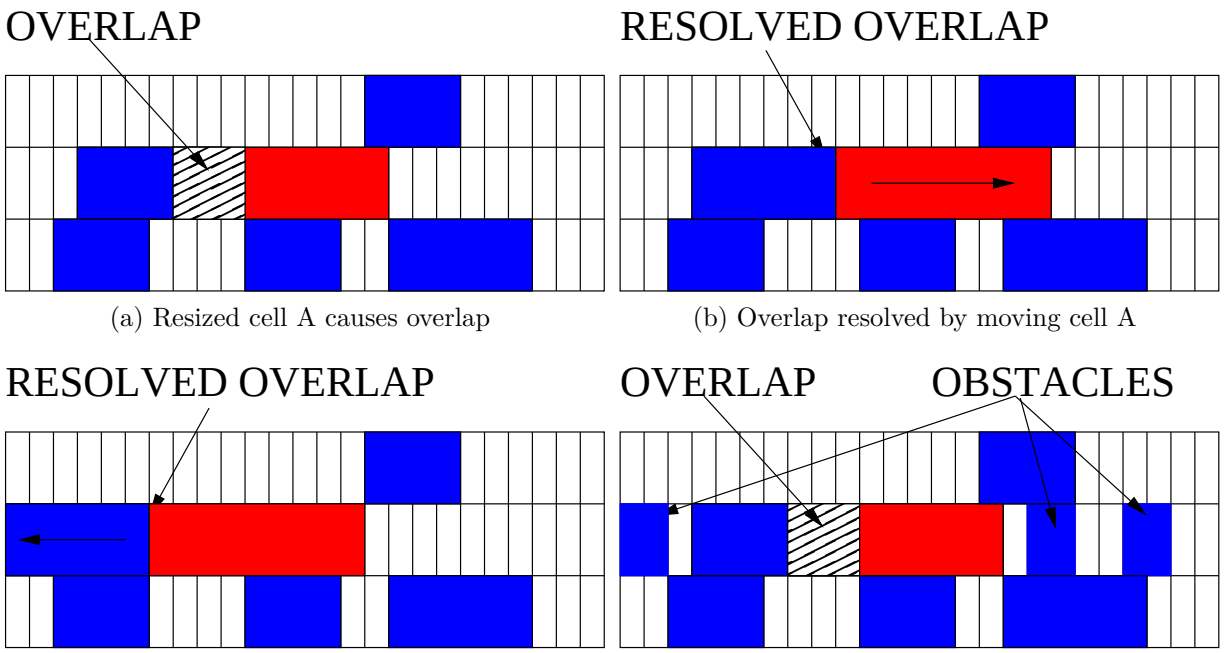
Leakage reduction, as a post-placement optimization tool, must not disturb placement in ways which can affect either its timing characteristics, or its routability and legality. Timing is guaranteed by our flow by the aforementioned methods. We guarantee routability and legalization with the use of density screens and up-to-date information about the occupied locations of the layout area.

Our flow uses downsizing in order to replace cells with "smaller" ones. However this term typically refers to the speed and drive strength of cells, rather than their physical size. This means, that the new cell, which we are substituting another with, may actually be larger than the substituted one, potentially leading to congestion and legalization issues. This can be especially true if we substitute *e.g.* a *AND* gate with an adder.

We tackle the problem of congestion by enforcing density screens throughout the layout area. Similarly to SCPlace, we segment the area into regions, typically rectangular ones and bound the maximum congestion under each region. If not otherwise directed by the designer, we calculate the total congestion of the initial placement and bound the congestion under each region to the same percentage as the total congestion over all the chip. If resizing one cell causes a violation in the congestion of the area the cell belongs to, then this resize option is discarded. Thus, in dense areas, only resizes which select smaller, in the physical sense cells, or cells with the same physical dimensions as the original ones are selected. However, our flow supports global and local density constraints, which can be provided to the tool at the designer's discretion.

In order to cope with any potential legalization issues, we maintain up-to-date information about the occupied sites of the manufacturing grid, in the same way as we do in SCPlace (c.f. Section 4.7.6). Every time a cell is resized, the status of the sites it occupies is updated. If the resize causes a legalization conflict with a neighbouring cell, then one fast check is made before this resize option is discarded. Since every standard cell has fixed height, the only dimension that can change during a resize is its width. An overlap can occur if another cell, or the core boundary, is at its left or at its right. One of these cases is illustrated in Figure 5.6a. If this case arises, we check first, if by moving the resized cell slightly to the left or to the right resolves the conflict. This is shown in Figure 5.6b. If this fails, we try to move the neighbouring cell slightly, as shown in Figure 5.6c. If both heuristics fail, then we discard this resizing as it seems it is causing too much unrest in a dense region, where overlaps will likely be hard to fix. An

extreme of this case is illustrated in Figure 5.6d.



(a) Resized cell A causes overlap

(b) Overlap resolved by moving cell A

(c) Overlap resolved by moving neighbouring cell

(d) Overlap hard to resolve

Figure 5.6: Overlaps after standard cell resizing

5.8 Runtime Issues

The hardest part of our leakage optimization flow is to calculate the statistical slacks for each gate. This is done by our TSZSA algorithm, which, coupled by the hierarchical approach described in Section 3.6.2 can handle large circuits in reasonable time, as will be shown in Chapter 7. Once slacks are known, then the resize process can begin, examining one gate at a time. For each gate, its alternatives are checked in decreasing drive strength order. In the worst case, each gate will have enough slack so that any resize is good, meaning that all alternatives will be examined. However, in a typical technology library, a gate will have no more than 10-15 cells alternatives, often less than 10. For every resize, incremental SSTA is performed in the neighbouring cells, which will typically be around 10, depending on the number of fanouts the cell has. Incremental SSTA only requires the evaluation of a few analytical functions, as shown in Section 3.1. Checking for congestion is straightforward as it requires only a simple

calculation of the previous and the new size of the cell. Our legalization approach, described in Section 4.7.6 is a fast and efficient way of keeping track of occupied area in the layout and does not require any time-consuming operations. Overall, the runtime of our leakage flow is dominated by the slack (re)-assignment and no runtime bottlenecks are likely to appear in other functions.

In the next chapter, we move to asynchronous placement and present our novel constructive placement algorithm, *CPlace*, which can handle asynchronous circuits.

Chapter 6

CPlace

In this chapter, we present CPlace [47], our placement tool which supports both synchronous and asynchronous circuits. CPlace is an evolution of SCPlace [48], presented in Chapter 4, our statistical timing based placer, which supports absolute and relative wire bounds for meeting a (mean, sigma) constraint.

CPlace exploits: (i) absolute delay bounds produced by asynchronous timing analysis, (ii) relative bounds for satisfying QDI constraints, and (iii) LP-based slack re-assignment. We use TSE to derive bounds on wire lengths that guarantee performance of the placed circuit. We identify all isochronic forks of the circuit and bound the absolute difference in the delay of each leg in each isochronic fork in order to guarantee the speed independent assumptions of the circuit. Both types of bounds are then forwarded to the LP formulation, after which, a slack assignment is derived that can be used for the circuit's placement, which we perform by employing a constructive approach.

6.1 Asynchronous Placement Requirements

Current state-of-the-art placers focus on either optimizing the total wirelength of a circuit, or meeting the clock period timing constraints. These goals are generally one-sided, *i.e.* improvement over one step of the optimization process may be evaluated by direct comparison with an absolute value. For example, in the case of timing driven placement, if the delay of the critical path is decreased, while the delay of no other path is increased, then the new placement is considered an improvement over the previous one.

However, in the case of asynchronous placement, two-sided constraints may generally need to be met. Certain wires, *i.e.* legs of isochronic forks, will require both a minimum and a maximum allowable delay constraint. Such bounds may not be known a priori, as they are relative to other wires. However, as far as we know, synchronous placement algorithms do not support relative timing constraints.

Furthermore, conventional STA engines used during timing-driven placement assume that the circuit is acyclic. When cycles are encountered during STA, STA engines will typically break them arbitrarily and analyze the resultant acyclic timing graph. Asynchronous control circuits are cyclic circuits, therefore cannot be effectively analyzed using STA, or timing-driven placed by existing synchronous placers with a given performance goal such as asynchronous period.

Thus, an efficient placer for asynchronous circuits must incorporate a timing analysis engine which can handle cycles, like TSE, and must also be able to tackle the contradicting targets created by the relative and two-sided constraints.

6.2 CPlace's Interface

CPlace has been designed to conform to industry standards with regard to circuit and library descriptions.

The concurrent nature of asynchronous circuits often results in a description format for concurrent systems like the ones described in Section 2.1.4. In other cases, the circuit is described using standard gate-level description languages, like Verilog. In the latter case, the circuit can be directly processed by the core of CPlace. In the former case, a preprocessing step is required in order to transform the circuit description into an industry-standard format.

CPlace supports the Event-Rule specification format, as described in Section 2.1.4 in case the circuit is described as a concurrent system. If the circuit description is given in this format, CPlace uses an external tool to transform the event-rule representation into *Verilog* format. This is done using the tool Petrify [18], which can translate the concurrent specification into gate-level representation and then map the gate representation into a given technology library. CPlace provides Petrify with the specific technology gates and then receives the final gate-level netlist.

For processing by CPlace's core, the circuit must have been mapped into a specific technology, a process which provides its gate-level description. CPlace, like SCPlace, supports the full

set of *Verilog*'s semantics.

The technology library's description must be in *LIB* and *LEF* formats. CPlace creates a final placement which will then be processed by a router. The final placement is described in *DEF* format.

We have verified the interface of CPlace with an industrial state-of-the-art router by passing all CPlace's placements to the router for detailed routing. All placement were successfully routed proving that CPlace can be integrated into an industrial flow without any interface issues with a standard router.

6.3 CPlace Objectives

CPlace, being a placer which can handle asynchronous circuits, must be able to optimize for both traditional, synchronous circuits optimization objectives, but also for performance and correctness objectives referring to asynchronous circuits. Below we list the objectives CPlace is guided from and the constraints it conforms to.

- **Wire length constraints.** Minimization of total wirelength is the traditional optimization objective for academic placers and has also been one of the main objectives for industrial placers before the timing-driven placement era. Although CPlace does not aim at optimizing for wirelength directly, it does support constraints on the lengths of individual wires. These constraints can be either hard, prohibiting CPlace from creating wires larger than a threshold or soft, providing CPlace with wire length directives. If the designer provides CPlace with adequate constraints for individual wires, then CPlace is capable of optimizing the total wire length.

- **Asynchronous timing performance constraints.** Performance constraints refer to how fast the asynchronous circuit can operate. This follows directly from the timing analysis on the timing cycles of the asynchronous circuit, which can provide information about the cycle time or the delay of the circuit. By performing TSE on the asynchronous circuit, as described in Section 2.1.4, CPlace can be provided with directives as to how the circuit can be optimized for faster cycles or smaller delay. This will directly result in timing performance optimization.

- **QDI constraints.** The second set of timing constraints refer to correct operation of the asynchronous circuit. Since CPlace works with QDI assumptions on asynchronous circuits, there need to be in place a set of constraints which guarantee that no isochronic forks are violated, as described in Section 2.1.4. These constraints do not refer to the performance of the circuit, but they guarantee that there are no timing violations which can jeopardize the correct flow of data through the circuit, or in other words, the correctness of computations. CPlace treats this set of constraints as hard constraints, *i.e.* it is not allowed to introduce violations of any kind.

- **Layout area.** Being a placer algorithm, CPlace accepts a total layout area as a hard constraint as to where it is allowed to place cells. The layout area does not have to have a regular shape. This means that it may not be rectangular, or it may contain blockages on which cells cannot be placed. CPlace accepts a geometric description of the layout area which defines the available space in order to place all cells in their best locations.

- **Density screens.** To enhance routability, CPlace does not allow for too dense designs. This is enforced with the use of density screens throughout the layout area. Density screens define the maximum allowable density on various regions of the layout. CPlace accounts for these constraints by routinely checking if the region on which it intends to place a new cell is sparsely populated enough, so that the new cell will not cause a density violation.

- **Legalization.** A common problem among academic placers is that they may produce placements which are not legal. This means that a legalization step is necessary after placement. The first problem with this is that legalization may not be straightforward especially in densely populated regions of the layout. Secondly, the legalization step itself may move cells to locations which cause timing or other violations. CPlace does not suffer from this problems, as it produces a correct-by-construction placement. The occupied and the free locations on the layout are kept track of, at all times. Thus, when CPlace decides for a location for a cell, it first checks if this location causes any overlaps; if it does so, it rejects it and searches for another location. Thus, no overlaps may be introduced at any stage of the placement process.

- **Design-rule constraints.** CPlace accepts a set of common industry design-rule constraints which eliminate the need for a correction step after placement in order for the

placement to be ready to undergo fabrication. Specifically, CPlace places all cells in locations which are aligned with the given manufacturing grid.

6.4 Slack Assignment for Asynchronous Circuits

CPlace is a constructive placer which uses bounds on the wire lengths for guidance. In order to derive bounds for the performance-efficient and QDI placement of any asynchronous circuit, we must identify the critical cycles of the circuit and also examine the presence of isochronic forks.

6.4.1 Wire-Delay Bounds

We need to distinguish between two types of constraints: absolute and relative constraints. Performance of an asynchronous circuit is determined by its cycle time, which is the speed at which its cycles execute. The critical cycle of the circuit dominates the overall performance. Thus, translating the delay of the most critical cycle to an absolute timing constraint is an efficient method of creating performance constraints for the circuit.

On the other hand, in order to satisfy QDI, all wire forks must be isochronic with respect to a given specified margin.

An example of the constraints, both absolute and relative, resulting from a circuit is shown in Figure 6.1.

TSE can identify the critical cycles and their delay in the circuit graph. In the example of Figure 6.1, there are several cycles; depending on the delay of the individual gates, any of them can be critical. The cycle that is most likely to be critical is the one traversing gates $G1 \rightarrow G4 \rightarrow G5 \rightarrow G7$. In this case, TSE will evaluate its period p, which will serve as a performance constraint. We will add a constraint $AT(G7) - AT(G1) \leq p$, which states that the maximum allowable difference in the arrival times between $G1$ and $G7$ is p. Similarly, we create constraints for the remaining cycles, thus producing a complete set of absolute constraints targeting performance.

In order to satisfy QDI operation, we need to bound the difference of delays in each leg of each isochronic fork. We identify isochronic forks in the case of a gate which has more than one fanouts. In Figure 6.1 an example of a gate with more than one fanouts is $G3$. Thus, we need

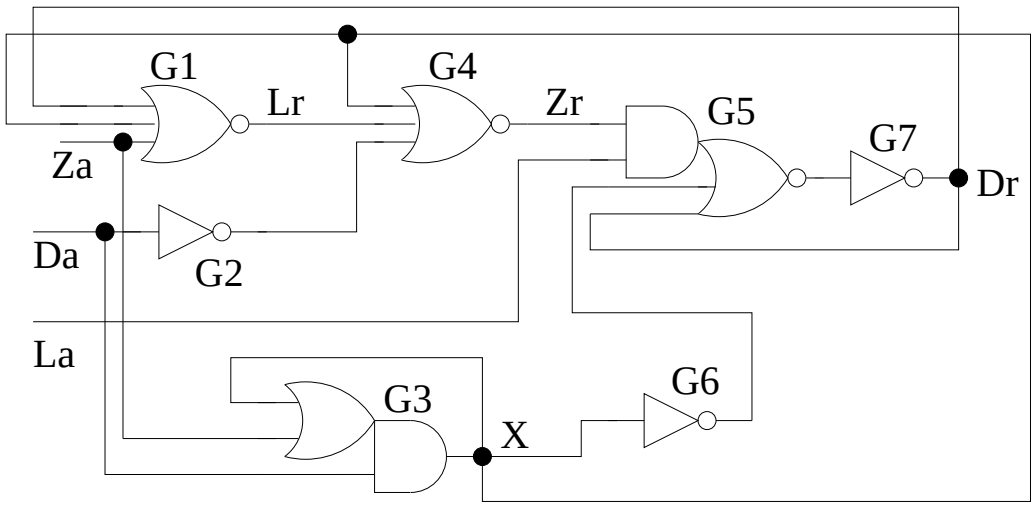


Figure 6.1: Absolute and relative constraints

to create constraints for the four legs of the isochronic fork: $G3 \rightarrow G3$, $G3 \rightarrow G1$, $G3 \rightarrow G6$ and $G3 \rightarrow G4$. These can be implemented the following inequalities:

$$\begin{aligned}
|(d_{G3\rightarrow G3}) - (d_{G3\rightarrow G1})| &\leq Isochronic_Bound \\
|(d_{G3\rightarrow G3}) - (d_{G3\rightarrow G4})| &\leq Isochronic_Bound \\
|(d_{G3\rightarrow G3}) - (d_{G3\rightarrow G6})| &\leq Isochronic_Bound \\
|(d_{G3\rightarrow G1}) - (d_{G3\rightarrow G4})| &\leq Isochronic_Bound \\
|(d_{G3\rightarrow G1}) - (d_{G3\rightarrow G6})| &\leq Isochronic_Bound \\
|(d_{G3\rightarrow G4}) - (d_{G3\rightarrow G6})| &\leq Isochronic_Bound
\end{aligned} \tag{6.1}$$

We define the $Isochronic\_Bound$ to be equal to the delay of a fast inverter for the target technology library, which is a realistic constraint for an automated placement tool. Similarly, we can create relative constraints for any isochronic fork of the circuit. Equations 6.1 are part of the LP problem, thus any resulting slack assignment will meet both the absolute and the relative constraints.

6.4.2 LP Formulation

The set of constraints for the LP problem CPlace formulates are of the following types.

- **Maximum and Minimum Wire Lengths.** The length of each wire which corresponds to its delay in the final slack assignment should be bounded within reasonable limits. An

upper bound is essential to avoid asking from the placer to create too long wires. A lower bound is equally important to allow the placer some flexibility in placing cells instead of over-restricting their maximum distance.

- **Absolute upper bounds for every cycle in the circuit.** This type of constraints guarantees the performance of the asynchronous circuit.
- **Relative bounds for each isochronic fork.** Constraints of this type must make sure that the wires in each fork do not bias the delay towards one leg or another.
- **Timing graph description.** The LP formulation must have information about the connections between the circuit elements and the wires it will try to optimize. Each gate input must be connected to its driver, through a wire whose length is part of the objective function and each gate output must be connected to the gate's inputs.
- **Objective function.** We seek to maximize the total wire length to allow the placer the greatest flexibility. This, coupled with the upper bounds for performance and relative bounds for speed independence, will steer the LP solution towards a performance-efficient slack assignment which respects asynchronous delay assumptions.

The LP formulation is shown in Equations set 6.2.

$$\begin{aligned}
max & & \Sigma w_i \\
w_i & \leq & Wire_bound \\
\forall(i \rightarrow j) & & \text{cycle} \\
AT(j) - AT(i) & \leq & Cycle_Bound \\
\forall\{(g \rightarrow i), (g \rightarrow j)\} & & \text{isochronic fork} \\
|d(g \rightarrow i) - d(g \rightarrow j)| & \leq & Isochronic_Bound \\
AT(g_i) & \geq & AT(g_{driver}) + w_i \\
AT(g_o) & = & AT(g_{slowest_in}) + d_{propagation}
\end{aligned} \tag{6.2}$$

$Wire\_Bound$ is the maximum allowable delay for a wire, $Cycle\_Bound$ is the absolute constraint on the delay of each cycle in the circuit and $Isochronic\_Bound$ is the maximum margin of delay between the legs of an isochronic fork. Equation set 6.2 requires that we maximize the delays of wires, while at the same time, all wires have a maximum delay of $Wire\_Bound$. Additionally, the delay of all cycles must be bounded by $Cycle\_Bound$. For

QDI, we require that all legs in isochronic forks have a maximum margin, in terms of their delay, of $Isochronic\_Bound$. In order to complete the LP problem, we define that the arrival time (AT) of any gate input is the sum of the arrival time of its driver plus the delay of the wire connecting them. Finally, the arrival time of a gate output is the sum of the arrival time at the gate's slowest input plus the propagation delay of the gate.

We have used the GNU GLPK solver [25] to solve the LP problem and produce the slack bounds on wires which are to be used later in the placement algorithm.

6.5 The CPlace Flow

The CPlace flow is illustrated in Figure 6.2. Pseudocode for CPlace is shown in Algorithm 6.1.

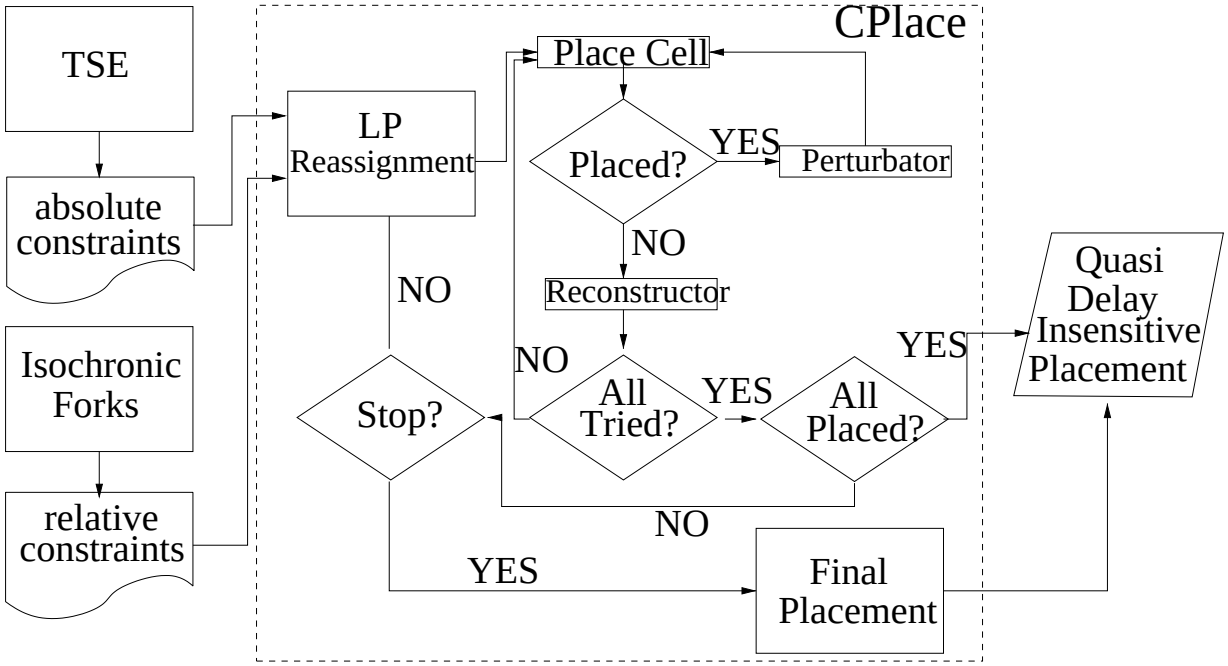


Figure 6.2: CPlace flow

CPlace starts by transforming the given STG-file into a *Verilog* file, if one is provided (Lines 2 to 4). Otherwise, it uses the *Verilog* netlist provided. The first step of CPlace's core is to extract all isochronic forks from the circuit's description. This is done by function **IsochronicForks** using the information from the netlist only. Then, asynchronous slack assignment is done using any global performance constraints d, the list of isochronic forks and the netlist (Line 6). The

Algorithm 6.1 - Constructive Placement for CPlace

```
1: CPlace(d, STG, Netlist)
2: if STG then
3:    Netlist = STGtoNetlist
4: end if
5: IsochronicForks = FindForks(Netlist)
6: SlackAssignment(d, Netlist, IsochronicForks)
7: SortedCells ⇐ SortConnectivity(cells)
8: it ⇐ num_cells / 100
9: for i = 0 to it do
10:    for j = 0 to num_cells do
11:       cell ⇐ SortedCells[j]
12:       if !Placed[cell] then
13:          if Place(cell) then
14:             RelatedCells ⇐ IsochronicForks(cell)
15:             for all cell_i ∈ RelatedCells do
16:                PlaceQDI(cell, cell_i)
17:             end for
18:          else
19:             Reconstructor(cell)
20:          end if
21:       end if
22:    end for
23:    Perturbator()
24: end for
25: CPlace Finalization()
```

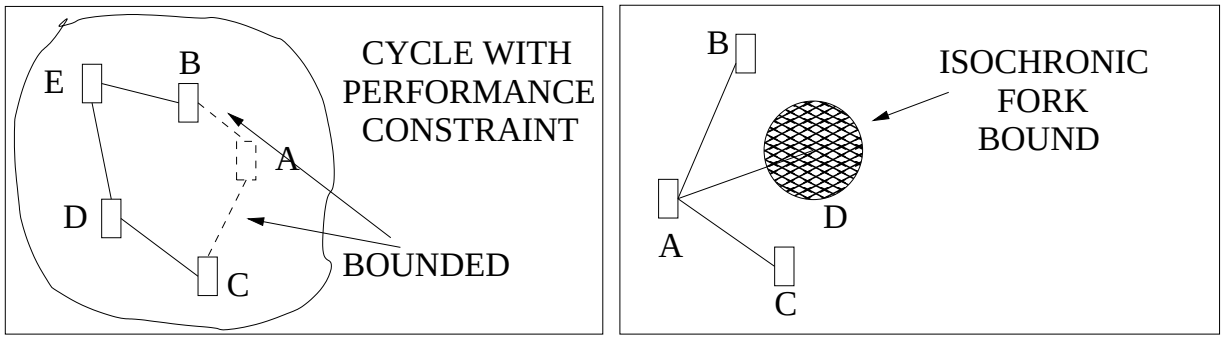
constructive placement begins then, with creating a sorted list of cells according to the number of their connections (Line 7). The list is traversed for a number of iterations (Lines 9 to 24) and at each time a cell is selected to be the next to be placed (Lines 10 to 22). Every time a cell is selected, CPlace tries to place it at a suitable location (Line 13). If this placement is successful, then all the cells which form isochronic forks with the cell just placed are retrieved (Line 14) and suitable locations for their placement are sought (Lines 15 to 17). If the placement of the initial cell is unsuccessful, then the *Reconstructor* is called (Line 19) in order to correct the location of the cell's neighbours. After one full traversal of the list of cells, the *Perturbator* is called (Line 23) in order to shuffle the placement and help CPlace escape any local minima. After the number of iterations has passed, the placement is finalized (Line 25).

6.6 CPlace Implementation Details

In this section we discuss the specific implementation details of CPlace. Although CPlace is an evolution of SCPlace, discussed in Chapter 4, some of the implementation details have been altered in order to fit better with asynchronous circuits.

6.6.1 Constructive Process

The constructive placement process operates by selecting one of the unplaced cells and placing it to a location which meets its constraints. Specifically for CPlace, these constraints are absolute constraints if the cell belongs to a cycle and relative constraints if there is an isochronic fork related to this cell. The first case is shown in Figure 6.3a. Cell A, connecting to both cells B and C, will likely be placed near cells B and C due to the fact that they are all part of a cycle, whose delay must not exceed a performance bound. The second case is illustrated in Figure 6.3b. Here, cell A has multiple fanouts and only cells B and C have already been placed. Cell D is pending and must meet relative constraints for the legs of the isochronic fork $A \rightarrow B$, $A \rightarrow C$ and $A \rightarrow D$. Thus, it is most likely to be constrained into the shaded region so that the length of the wire $A \rightarrow D$ matches the lengths of the other two wires.



(a) Cycle performance bound (b) Isochronic bound

Figure 6.3: CPlace constructive bounds

6.6.2 Reconstruction

The *Reconstruction* step is a similar procedure, as the one described in Section 4.7. It is aimed at correcting random decisions on the placement of seed cells, which may hamper the

placement of other cells later in the constructive placement process. The difference against the *Reconstructor* of 4.7, is that at each step, care is taken that the isochronic forks are fixed. Algorithm 6.2 shows pseudocode for the CPlace reconstructor.

Algorithm 6.2 - CPlace Reconstructor Algorithm

```
1: Reconstructor(cell)
2: NeighboursArray ⇐ GetNeighbours(cell)
3: for all neighbour_i ∈ NeighboursArray do
4:    /* Unplace all the neighbours */
5:    placed[neighbour_i] = 0
6:    stored_location[neighbour_i] = location[neighbour_i]
7: end for
8: PlaceMostConstrained (NeighboursArray)
9: /* Place the problematic cell in a random position */
10: Place_Constrained(cell)
11: for all neighbour_i ∈ NeighboursArray do
12:    /* Try to place all neighbours in good positions */
13:    PlaceQDI(neighbour_i)
14: end for
15: for all neighbour_i ∈ NeighboursArray do
16:    if placed[neighbour_i] == 0 then
17:       for all neighbour_i ∈ NeighboursArray do
18:          location[neighbour_i] = stored_location[neighbour_i]
19:       end for
20:       return failure
21:    end if
22: end for
23: /* If all neighbours were successfully placed, return with success */
24: return success
```

The *Reconstructor* in Algorithm 6.2, works on a cell which could not be placed. It first finds and removes from the placement all its neighbours (Lines 2 to 7), storing their current locations in case the reconstruction is unsuccessful. Then, the neighbouring cell which is most constrained is placed (Line 8) and next, each neighbour is placed taking special care of its QDI constraints (Lines 11 to 14). If the reconstruction is unsuccessful, all cells are restored to their previous locations (Lines 15 to 22). Else, the reconstruction is successful and all cells have been placed (Lines 23 to 24).

6.6.3 Perturbation

In order to escape local minina by shuffling the current placement solution, a *Perturbator* function is used. This is a simulated annealing process, which works on only the already placed cells. It aims at reducing total wire length, without violating any constraints. Thus, at each step, the simulated annealing process ensures that no isochronic forks are violated and no performance constraints are exceeded. The *Perturbator* of CPlace is the same as the *Perturbator* of SCPlace, described in Section 4.7, the only difference being that the type of constraints that simulated annealing must check at each step is not of statistical nature, but of asynchronous nature.

6.6.4 Finalization

At the end of the constructive placement process, there may exist some cells, for which a valid location was not found. CPlace always delivers a full, legal, placement, so in this case, the remaining cells will be placed at valid locations, albeit incurring some constraint violations. The idea of *Finalization* is to cause as little violation as possible. All the remaining cells are sorted with respect to the number of their constraints. Then, for each cell, a location is chosen disregarding one constraint at a time. In the worst case, all its constraints will be removed and the cell will be placed at random. Pseudocode for the *Finalization* process of CPlace is shown in Algorithm 6.3.

Algorithm 6.3 - CPlace Finalization Algorithm

```
1: CPlace Finalization()
2: UnplacedArray ⇐ GetUnplaced()
3: for all cell_i ∈ UnplacedArray do
4:    cell_i−>ConstraintsArray ⇐ GetConstraints(cell_i)
5: end for
6: SortCellsConstraints (UnplacedArray)
7: for all cell_i ∈ UnplacedArray do
8:    while Unplaced (cell_i) do
9:       PlaceCell (cell_i)
10:      RemoveConstraint (cell_i)
11:   end while
12: end for
```

Algorithm 6.3 starts by finding all unplaced cells (Line 2). All the constraints for each cell

are stored (Lines 3 to 5). Then, all unplaced cells are sorted with respect to the number of their constraints (Line 6). The performance constraints are placed before the QDI constraints, so that if a violation must occur, it may affect performance rather than correctness. Then, for each cell, as long as it has not been successfully placed (Lines 8 to 11) the finalization process tries to place it (Line 9) and then removes one constraint (Line 10) in order to try again with fewer constraints if the placement trial did not succeed. At the end of the finalization process, all cells will have been placed, although some of them may have end up in random locations.

6.6.5 Routability and Legalization

To guarantee routability, CPlace employs the use of density screens throughout the design. These are formed by n horizontal and m vertical lines, separating the layout in nm regions. The density of each region is bounded to 70%. This is a hard limit on the amount of standard cells that each region can accommodate. If CPlace is faced with the option of exceeding the density of a region or not placing the cell, it chooses the latter, hoping that the unplaced cell will be placed later, benefiting from the effects of reconstruction or perturbation. The density of the whole design is also bounded to 65% which means that not all regions must have exactly the same amount of utilization.

6.7 Runtime Issues

Runtime of CPlace is dominated by the solution of the LP problem, which derives the slack assignment used later in the constructive process. TSE, which is essential for identifying the cycles in the circuit to be later transformed into absolute wire bounds, is also time-consuming, but only needs to run once, before placement starts. The derivation of relative constraints due to isochronic forks is very fast, as it simply requires parsing the circuit description. Constructive placement, as in the case of SCPlace is very fast given all the necessary bounds. The auxiliary functions of reconstruction and perturbation are only in place to enhance the performance of CPlace and do not constitute a performance bottleneck in themselves. Detailed runtimes will be reported in Section 7.5, where these estimations are quantified. Overall, a fast timing analysis engine for asynchronous circuits and a fast slack allocation algorithm would greatly enhance the runtime of CPlace.

Chapter 7

Results

In this chapter, we evaluate the findings of our EDA algorithms. First, we evaluate our slack assignment algorithms, *i.e.* MSSA and TSZSA against a well-known, widely-adopted slack assignment algorithm called Zero Slack Assignment (ZSA) algorithm [70]. Our findings will show that our slack assignment algorithms, operating before any physical information, *i.e.* placement, is available, can derive statistical bounds which can manipulate the delay distributions at the circuit's outputs. Instead, ZSA derives a slack allocation under which the uncertainty of delay fluctuates uncontrollably. Next, we evaluate the results of our statistical placement tool, SCPlace, and our statistical leakage reduction flow. Both tools use the statistical bounds of MSSA and TSZSA. We show that our statistical bounds are suitable for integration with large-scale optimization processes such as a placement or a leakage reduction algorithm. We also show that our statistical placer and our leakage reduction flow compare favourably to existing state-of-the-art industrial and academic tools and algorithms.

A separate section is devoted to our placement algorithm, CPlace, which can handle asynchronous circuits. We show how CPlace can make effective use of both relative and absolute wire bounds in order to guarantee performance-efficient and operation-correct placements of asynchronous circuits. We prove that CPlace can indeed meet its targets by evaluating it with a super-set of the current state-of-the-art asynchronous benchmarks.

7.1 Benchmark Set

In this section we describe our benchmark set for both the synchronous flows (SCPlace and statistical leakage reduction) and our asynchronous placer (CPlace).

7.1.1 Synchronous Benchmarks

To validate our results, we have used state-of-the-art benchmarks, taken from the IWLS 2005 benchmark set [35]. This benchmark set is the only widely adopted set in the literature which contains synthesizable circuits. This property is a requirement for our flows, as is for standard industry flows, which require fully synthesizable circuits that can be mapped to a given technology. Using this kind of benchmarks, we can also compare directly against industrial flows. For this reason, we have rejected other benchmark sets, such as the IBM benchmark set [1],[2] which are not synthesizable and are only suitable for wirelength minimization.

Circuit	Cells	Area	Rows
b01	55	486.24	14
b02	27	257.94	10
b03	176	1625.55	24
b04	544	5275.07	44
b05	497	4384.91	40
b06	64	521.36	14
b14	4373	37759.64	118
b15	6445	50488.50	136
b17	18182	136529.37	222
b18	51277	407548.76	382
b19	91931	685848.53	495
b20	7844	61057.29	148
b21	8083	63194.32	151
b22	12016	92007.42	182
aes	26293	65043	153
des3	52174	176194	251

Table 7.1: Synchronous benchmarks

Table 7.1 shows the characteristics of the synchronous benchmarks. The benchmark set ranges from small circuits, `b01` to `b06`, to larger benchmarks, `b14` and above. Three of the benchmarks consist of more than 50,000 standard cells, which is a reasonable enough circuit size to test the validity of our tools. In order to prove the scalability of our tools, we needed larger, synthesizable benchmarks, which, unfortunately, are not available as a benchmark set. However, we have collected a number of real, reasonably sized circuits from various sources, including the Opencores [60] online resource which provides the description of real and often

industry-sized circuits. We have used the collection of these circuits in order to prove the scalability of our tools, measure their runtimes and evaluate the importance of the integration of our hierarchical flows in our proposed algorithms. The set of the larger benchmarks is shown in Table 7.2. `leon2` is a 32-bit CPU microprocessor core, based on the RISC architecture and instruction set. `vga_lcd` is a combined VGA and LCD controller. `b19_10` and `b19_20` are the `b19` benchmark instantiated 10 and 20 times respectively. These particular benchmarks were created with the sole purpose of showing the scalability of our tools.

Circuit	Cells	Area	Rows
leon2	273626	830847	545
vga_lcd	220014	596046	462
b19_10	801990	1611006	758
b19_20	1605740	3220019	1602

Table 7.2: Synchronous large benchmarks

7.1.2 Asynchronous Benchmarks

To validate our asynchronous placer's, CPlace, results we have used the most widely adopted state-of-the-art asynchronous benchmark set. We have enriched the benchmark set with an asynchronous version of the DLX processor, created with the desynchronization approach [19]. In view of the relatively small size of the asynchronous benchmarks, we have also created larger ones, by instantiating multiple times smaller benchmarks. The resulting asynchronous circuits, ranging up to 64,000 standard cells help prove the scalability of CPlace.

Table 7.3 shows the characteristics of the asynchronous benchmarks.

Circuit	Cells	Area	Rows
c3dec2	11	42	4
ccc	7	16	3
chu133	7	16	3
chu150	10	25	3
converta	15	47	4
half	10	38	4
mmu	25	85	6
mp_forward_pkt	12	42	4
nak_pa	16	52	4
nowick	9	20	3
rcv_setup	7	20	3
rpdft	13	41	4
sbuf_read_ctl	10	30	3
sbuf_send_ctl	17	61	5
seq_mix	22	73	5
trimos_send	30	132	7
var1	8	36	4
vbe10b	36	136	7
vbe5b	12	36	4
vbe5c	10	28	3
vbe6a	29	148	7
wrdatab	32	100	6
xyz	8	32	3
seq_mix_10000	10560	34944	112
half_10000	10240	39322	119
mmu_10000	12800	43418	125
ccc_15000	14336	33587	110
converta_64000	61440	19331	83
dlx_desync	15793	52277	137

Table 7.3: Asynchronous benchmarks

7.2 Slack Assignment Results

In this section, we present the evaluation of our slack assignment algorithms. MSSA is evaluated for its property to propagate a statistical distribution which has the minimum possible sigma, *i.e.* uncertainty, for the delay of the circuit's virtual sink. TSZSA is evaluated for its property to assign slacks to selected internal nodes of the circuit so that their delay distributions meet targets on both mean and sigma. These targets are calculated during runtime of TSZSA and are aimed at ensuring propagation of appropriate delay distributions to the circuit's outputs. We show that TSZSA can indeed calculate correctly the required values of mean and sigma for the delay of internal nodes which directly leads to meeting the designer-enforced bounds, *i.e.* targets on both mean and sigma of the circuit outputs's delay distributions.

We first demonstrate the results of MSSA with respect to its aim of minimizing the sigma of the circuit delay's distribution. Table 7.4 shows the delay distribution for the virtual sink

IWLS Circ.	Zero Wire Delay		MSSA	
	μ(ns)	σ(ps)	μ(ns)	σ(ps)
b01	0.21	20	0.37	10
b02	0.20	20	0.36	10
b03	0.35	30	1.31	10
b04	0.40	20	0.65	10
b05	0.66	40	2.87	10
b06	0.24	20	0.39	20
b14	1.63	100	5.37	7
b15	1.18	100	4.41	8
b17	1.23	120	4.86	6
b18	3.58	690	15.85	8
b19	5.22	870	17.45	8
b20	1.67	140	5.74	20
b21	1.78	130	5.7	20
b22	1.75	170	5.82	20
aes	0.95	10	3.52	6
des3	1.08	220	4.14	20
leon2	4.9	790	17.6	20
vga_lcd	3.5	670	15.4	20
b19_10	6.8	1010	21.2	20
b19_20	7.2	1090	22.7	20
Avg.	*2.22*	*313*	*7.78*	*13*

Table 7.4: Comparison of zero wire delay with MSSA

that MSSA can achieve for all benchmarks. In order to discuss MSSA's results with respect to the final mean and sigma, we compare against the zero wire delay (ZWD) model, which assumes zero delay for all wires. ZWD model is useful in terms of extracting the "hard" lower bound on the delay of the circuit, which is the delay of only the gates. Although unrealistic as a model, as any physical algorithm will introduce delay to account for the actual delays of wires, ZWD can be very useful for estimating the delay overhead each algorithm requires. It can also provide guidelines as to how much improvement is possible after the delays of wires are accounted for. Physical algorithms can abandon their optimization process if their results are close the the ZWD estimate, even if constraints are not met, in which case they may conclude that the constraints are unrealistic.

The results of Table 7.4 show that MSSA can indeed propagate a delay distribution with minimum sigma to the virtual sink. The reduction in sigma is 24× on average, compared to the delay reported by ZWD. Moreover, the absolute values of sigma are comparable for all circuits, showing that the the minimum possible sigma does not depend on the size or the structure of the circuit. Furthermore, the absolute value of the sigma is comparable to the sigma of a typical 2-input gate for the technology used. This shows that indeed, the delay of a gate with

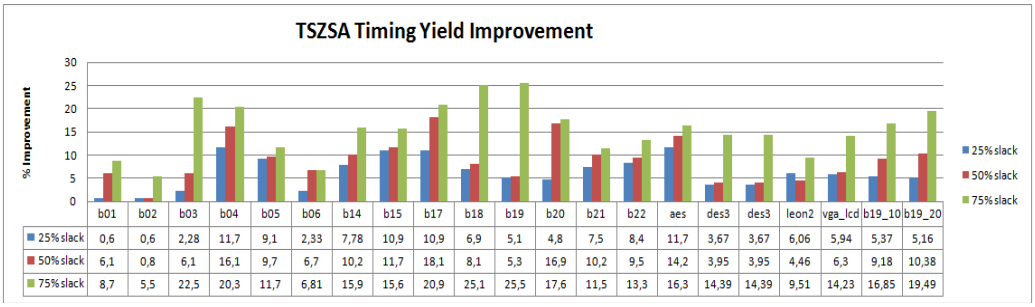
relatively small sigma is selected from MSSA and its delay distribution is propagated to the virtual sink by appropriate slack allocation. However, this approach comes at the the price of a significant increase in the mean delay. Compared to ZWD, MSSA needs about 3.5× more statistical mean, which means that MSSA alone is insufficient as a slack allocation algorithm.

IWLS Circ.	*25% slack*					*50% slack*					*75% slack*				
	ZSA		**TSZSA**		**Yield**	**ZSA**		**TSZSA**		**Yield**	**ZSA**		**TSZSA**		**Yield**
	μ(ns)	σ(ps)	μ(ns)	σ(ps)	**Impr.**	μ(ns)	σ(ps)	μ(ns)	σ(ps)	**Impr.**	μ(ns)	σ(ps)	μ(ns)	σ(ps)	**Imp.**
b01	0.25	20	0.24	20	**0.6**%	0.31	20	0.3	17	**6.1**%	0.38	30	0.37	17	**8.7**%
b02	0.24	20	0.23	20	**0.6**%	0.29	20	0.29	16	**0.8**%	0.36	30	0.36	16	**5.5**%
b03	0.42	30	0.42	20	**2.28**%	0.51	20	0.49	17	**6.1**%	0.62	50	0.63	16	**22.5**%
b04	0.51	20	0.48	18	**11.7**%	0.63	20	0.59	20	**16.1**%	0.77	60	0.76	20	**20.3**%
b05	0.81	30	0.79	20	**9.1**%	1.02	20	1.01	12	**9.7**%	1.22	80	1.28	12	**11.7**%
b06	0.30	20	0.28	20	**2.33**%	0.37	20	0.34	20	**6.7**%	0.48	40	0.48	20	**6.81**%
b14	2.10	140	2.03	90	**7.78**%	2.61	110	2.48	90	**10.2**%	3.18	240	3.21	70	**15.9**%
b15	1.52	130	1.44	80	**10.9**%	1.85	110	1.78	67	**11.7**%	2.21	210	2.27	51	**15.6**%
b17	1.61	170	1.55	90	**10.9**%	2.01	160	1.95	69	**18.1**%	2.53	270	2.59	53	**20.9**%
b18	4.35	910	4.2	504	**6.9**%	5.14	620	5.08	310	**8.1**%	5.91	1190	6.12	198	**25.1**%
b19	6.69	1010	6.4	697	**5.1**%	8.27	610	8.09	390	**5.3**%	10.9	1470	11.04	209	**25.5**%
b20	2.12	180	2.03	130	**4.8**%	2.53	110	2.42	72	**16.9**%	2.89	310	2.97	70	**17.6**%
b21	2.27	180	2.2	110	**7.5**%	2.74	120	2.65	81	**10.2**%	3.22	300	3.4	60	**11.5**%
b22	2.28	210	2.27	109	**8.4**%	2.87	140	2.82	78	**9.5**%	3.45	320	3.63	59	**13.3**%
aes	1.20	50	1.17	50	**11.7**%	1.47	40	1.42	31	**14.2**%	1.74	90	1.76	23	**16.3**%
des3	1.41	390	1.39	240	**3.67**%	1.78	310	1.81	172	**3.92**%	2.19	490	2.21	160	**14.39**%
leon2	6.2	910	6.1	510	**6.06**%	7.4	980	7.45	540	**4.46**%	8.7	1240	8.8	510	**9.51**%
vga_lcd	4.4	720	4.39	380	**5.94**%	5.4	820	5.43	410	**6.3**%	6.3	1110	6.32	390	**14.23**%
b19_10	8.5	1210	8.51	650	**5.37**%	10.3	1150	10.4	480	**9.18**%	12.1	1350	12.2	400	**16.85**%
b19_20	8.7	1240	8.72	670	**5.16**%	11.4	1210	11.4	510	**10.38**%	12.8	1510	12.82	430	**19.49**%
Avg.	*2.8*	*381*	*2.73*	*220*	**6.40** %	*3.45*	*330*	*3.41*	*170*	**9.19** %	*4.1*	*519*	*4.16*	*139*	**15.59** %

Table 7.5: Comparison of ZSA and TSZSA results for given slack, illustrating yield improvement

We now show how TSZSA, making effective use of the information derived by MSSA can manipulate the delay distributions in the circuit so as to meet targets on both mean and sigma. We want to compare the results of TSZSA against a state-of-the-art slack allocation algorithm. We have chosen ZSA, which either in its pure form, or enhanced by process-specific heuristics is widely used by contemporary industrial algorithms which require slacks on either wires or gates [70]. Since we perform statistical optimization, we need to account for both mean and sigma. One way to compare would be to examine mean and sigma separately. However, this would mitigate the very essence of statistical optimization, which tries to optimize for both values simultaneously. Thus, we have employed the metric of timing yield. Timing yield is the probability that, given a delay distribution and a constraint, the delay of a random sample from the delay distribution meets the constraint. Greater timing yield means that more circuits are likely to meet their timing constraints. Yield on the other hand, detached from any statistical notion, is a common metric in industry referring to the probability that a particular chip meets

its constraints. Thus, statistical timing yield is the natural expansion of the same metric to statistical flows.



	b01	b02	b03	b04	b05	b06	b14	b15	b17	b18	b19	b20	b21	b22	aes	des3	des3	leon2	vga_lcd	b19_10	b19_20
25% slack	0,6	0,6	2,28	11,7	9,1	2,33	7,78	10,9	10,9	6,9	5,1	4,8	7,5	8,4	11,7	3,67	3,67	6,06	5,94	5,37	5,16
50% slack	6,1	0,8	6,1	16,1	9,7	6,7	10,2	11,7	18,1	8,1	5,3	16,9	10,2	9,5	14,2	3,95	3,95	4,46	6,3	9,18	10,38
75% slack	8,7	5,5	22,5	20,3	11,7	6,81	15,9	15,6	20,9	25,1	25,5	17,6	11,5	13,3	16,3	14,39	14,39	9,51	14,23	16,85	19,49

Figure 7.1: TSZSA timing yield gains

Table 7.5 shows the timing yield gains of TSZSA over ZSA. The results are graphically depicted in Figure 7.1. Table 7.5 shows the delay distributions for all benchmarks, achieved by ZSA and TSZSA, for three different slack constraints. The slack constraints were defined at 125%, 150% and 175% of the zero wire delay's mean, which translates into 25%, 50% and 75% allowed slack over ZWD. Essentially, the slack constraint corresponds to a constraint for the delay's mean. For each constraint, ZSA and TSZSA were run, both aiming to distribute all the available slack. TSZSA was also constrained in terms of the resulting delay's sigma. For each benchmark, TSZSA was run multiple times in order to determine the minimum sigma that can be achieved for the given mean constraint. Therefore, it is expected that if the constraint in mean is relaxed, then TSZSA can minimize sigma more aggressively.

The results in Table 7.5 show that TSZSA claims 6.4%, 9.19% and 15.59% better timing yield compared to ZSA for 25%, 50% and 75% slack respectively. The timing yield gains are proportional to the amount of slack allowed, which can be explained by the fact that given more slack, TSZSA can effectively utilize it to apply more aggressive optimization on the delay's sigma. Table 7.5 also shows that TSZSA's resulting delay distributions have about the same mean as ZSA's, which means that TSZSA does not introduce any mean delay violations. However, the delay's sigma is significantly decreased, highlighting TSZSA's ability to manipulate sigma. The reduction in delay's sigma with no overhead in mean, effectively results in better timing yield.

The result averages of Tables 7.4 and 7.5 are graphically depicted in Figure 7.2, where the vertical axis is the sigma value, in *ps*, and the horizontal axis is the mean value, based on the allocated slack. The ZWD point serves as reference. ZSA and TSZSA have three points,

which correspond to 25%, 50% and 75% slack, while for MSSA we only consider the minimum sigma solution. Points for ZSA, MSSA and TSZSA all lie on the right of ZWD point, as all ZSA, MSSA and TSZSA distribute slack, which is considered zero for ZWD. As shown in Figure 7.2, ZSA exhibits unpredictable behaviour on sigma, which is not linked to the amount of slack allowed. TSZSA's results however, appear to correlate on a Pareto type curve, where sigma can be further optimized if more slack is allowed. Furthermore, TSZSA can produce the solution of MSSA if enough slack is allowed.

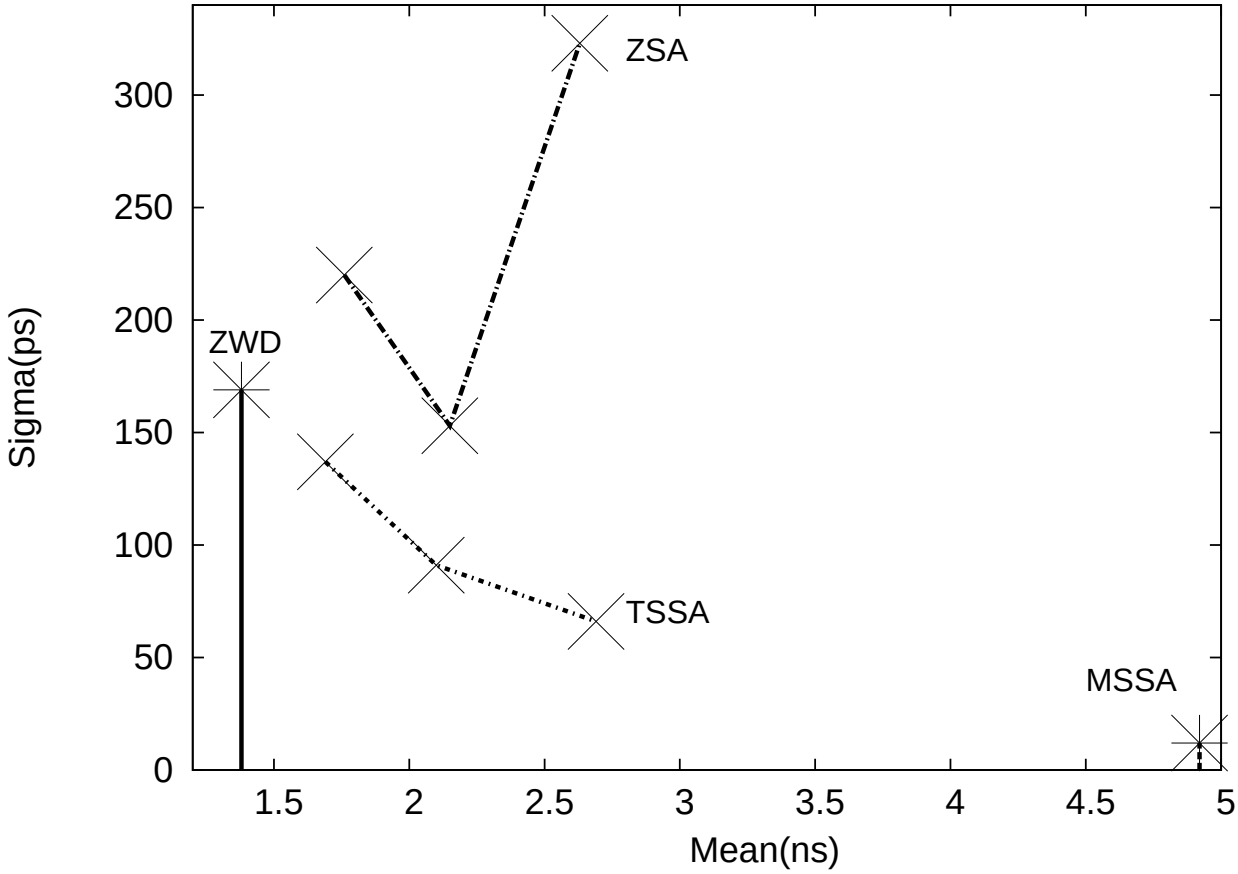


Figure 7.2: ZWD, ZSA, MSSA and TSSA comparison

Figure 7.3 illustrates the shape of the typical average mean, sigma tradeoff curve of the TSZSA algorithm. The leftmost mean point of this curve represents the 25% slack point, shown in Table 7.5, whereas the rightmost point represents the MSSA solution, *i.e.* no limit on slack. The specific circuit generating this curve is `b20`, however all other circuits exhibit a similar tradeoff. Essentially, the trend is an initial sharp drop in sigma with a small amount

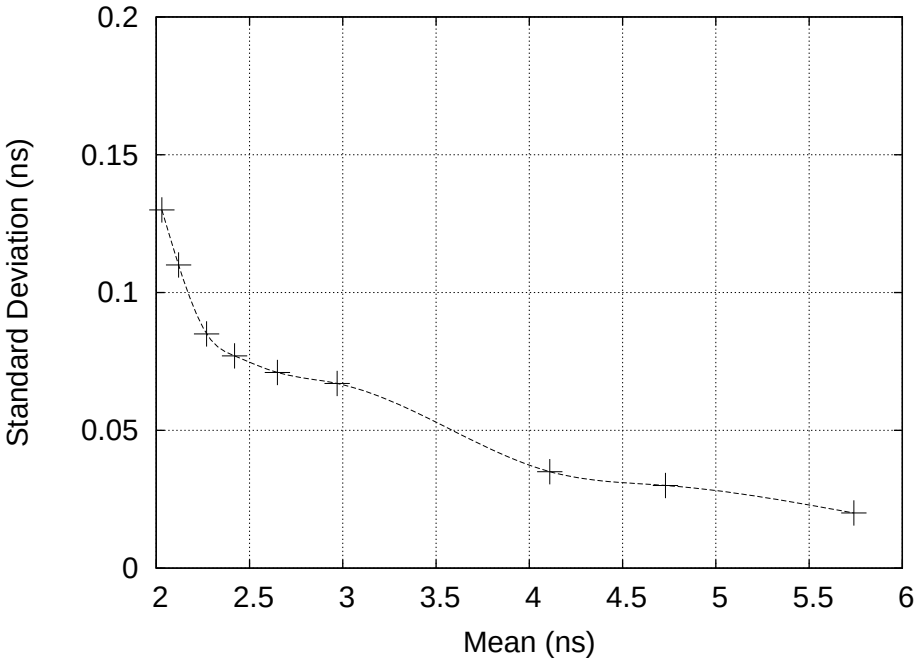


Figure 7.3: TSSA mean, sigma tradeoff curve

of slack. Then, another important point is the flattening of this curve, which for this example occurs for 50% of allocated slack. This happens when TSZSA has reached the point where no further optimization in sigma is possible regardless of the penalty allowed on mean.

To summarize, the TSZSA results indicate that with little change in the mean delay, TSZSA achieves a reduction in sigma, proportional to the available slack, which enables the algorithm to modify the arrival times at the circuit's internal timing nodes appropriately.

7.2.1 Slack Assignment Runtime

Any step of a contemporary EDA flow must provide optimizations in a reasonable time frame. In this section we show runtime results for our slack allocation algorithms. We specifically show that our hierarchical approach, described in Section 3.6.2 enables our slack allocation strategies to handle circuits consisting of several hundred thousand standard cells. Circuits of this size are currently regarded by academia as large enough to test the scalability of physical algorithms. Table 7.6 shows the runtime results for MSSA and TSZSA. All experiments were performed on an Intel(R) Core(TM)2 Duo CPU E8500 running at 3.16GHz with 4GB of RAM. Profiling results were collected with the GNU GPROF [27] tool.

	Runtime (sec)			
	Non-Hierarchical		Hierarchical	
Circuit	MSSA	TSZSA	MSSA	TSZSA
b01	3	15	3	17
b02	3	14	3	15
b03	5	17	5	18
b04	4	15	4	16
b05	5	13	5	15
b06	5	17	5	15
b14	12	302	12	84
b15	10	287	10	79
b17	13	311	13	93
b18	18	15783	18	1305
b19	20	19038	20	1592
b20	14	471	14	117
b21	13	459	13	109
b22	14	502	14	98
aes	12	399	12	102
des3	13	613	13	96
leon2	51	**N/A**	51	12035
vga_lcd	55	**N/A**	55	11593
b19_10	170	**N/A**	170	18301
b19_20	350	**N/A**	350	37598

Table 7.6: MSSA and TSZSA runtime

Table 7.6 shows the runtime for MSSA and TSZSA with both hierarchical and non-hierarchical approaches. The runtime for MSSA is the same for both approaches, as it simply requires a forward traversal of the circuit's timing graph and thus, no gains are expected from a hierarchical approach. In the case of TSZSA however, the segmentation of the whole problem into smaller sub-problems yields significant runtime gains. Table 7.6 confirms that MSSA runs very fast for all benchmarks. The runtime is correlated linearly to the circuit size. The runtime of TSZSA on the other hand, relies on the runtime of the LP solver, which does not scale well with circuit size. It is expected that the GLPK solver, which we employed for solving the LP problem, has complexity no better than polynomial, which calls for the application of our hierarchical methodology. As shown in Table 7.6, the runtime for the larger circuits becomes prohibitive, while for the largest circuits, TSZSA fails to provide a solution without our hierarchical approach. On the other hand, the use of hierarchical approach, can introduce a small runtime penalty on the smaller circuits, but enables TSZSA to derive bounds even for the largest circuits. Runtimes of a few hours are not uncommon in contemporary EDA tools for circuits with a few million transistors, like `b19_20`, which consists of 1,600,000 standard cells.

7.3 SCPlace Results

In this section we discuss the results of SCPlace. Since SCPlace aims at statistical optimization, we employ statistical metrics for evaluating its performance. Thus, as previously, we measure the timing yield gains obtained by SCPlace. Specifically, we investigate the timing yield gains obtained by SCPlace compared to a state-of-the-art industrial flow, a statistical placer which does not use bounds and a state-of-the-art academic placer. We also show that the placements produced by SCPlace are routable. Since we have shown that our hierarchical approach for the derivation of wire bounds is superior to the non-hierarchical one, we have only used the former for the presentation of results in this section.

7.3.1 Timing Yield

IWLS	**Capo [10]**			**Industrial Placer**			**SSAPlace**			**Yield**	**SCPlace**			**Yield**
Circ.	μ(ns)	σ(ps)	HPWL	μ(ns)	σ(ps)	HPWL	μ(ns)	σ(ps)	HPWL	**Imp.**	μ(ns)	σ(ps)	HPWL	**Imp.**
b01	0.23	42	6.96×10^{05}	0.21	39	6.46×10^{05}	0.22	38	6.98×10^{05}	**-0.25%**	0.22	23	6.98×10^{05}	**2.17%**
b02	0.37	45	4.46×10^{05}	0.33	37	5.41×10^{05}	0.35	36	4.51×10^{05}	**-0.59%**	0.32	28	4.49×10^{05}	**2.33%**
b03	0.61	60	1.87×10^{06}	0.56	49	2.23×10^{06}	0.59	49	1.89×10^{06}	**-0.87%**	0.56	27	1.87×10^{06}	**4.95%**
b04	0.82	49	7.07×10^{06}	0.75	47	6.88×10^{06}	0.81	46	7.14×10^{06}	**-3.92%**	0.77	23	7.11×10^{06}	**2.94%**
b05	1.23	129	8.46×10^{06}	1.14	113	8.88×10^{06}	1.17	109	8.53×10^{06}	**-0.23%**	1.20	28	8.54×10^{06}	**10.2%**
b06	0.49	51	2.39×10^{07}	0.44	48	8.72×10^{06}	0.51	45	2.42×10^{07}	**-5.05%**	0.44	24	2.41×10^{07}	**6.81%**
b14	3.58	151	2.83×10^{08}	3.45	142	2.55×10^{08}	3.52	136	3.01×10^{08}	**-0.45%**	3.47	85	2.98×10^{08}	**2.68%**
b15	1.91	138	1.90×10^{08}	1.78	126	1.59×10^{08}	1.80	125	2.05×10^{08}	**-0.21%**	1.75	78	1.97×10^{08}	**3.36%**
b17	1.78	112	6.17×10^{08}	1.65	107	5.11×10^{08}	1.73	99	6.21×10^{08}	**-0.75%**	1.68	44	6.19×10^{08}	**6.55%**
b18	6.03	249	1.61×10^{09}	5.87	234	1.37×10^{09}	6.11	217	1.89×10^{09}	**-1.70%**	5.93	113	1.78×10^{09}	**4.46%**
b19	7.65	342	1.82×10^{09}	7.13	296	1.51×10^{09}	7.45	282	2.21×10^{09}	**-2.22%**	7.21	159	2.05×10^{09}	**3.01%**
b20	2.03	132	6.36×10^{08}	1.84	127	5.66×10^{08}	1.85	124	6.57×10^{08}	**0.13%**	1.85	79	6.54×10^{08}	**2.62%**
b21	2.44	205	5.71×10^{08}	2.43	202	5.02×10^{08}	2.49	193	6.14×10^{08}	**-0.24%**	2.45	112	6.12×10^{08}	**3.92%**
b22	2.83	247	9.38×10^{08}	2.58	239	7.83×10^{08}	2.91	225	1.19×10^{09}	**-4.36%**	2.67	74	9.85×10^{08}	**9.68%**
aes	1.41	93	7.65×10^{08}	1.34	91	7.00×10^{08}	1.42	87	8.32×10^{08}	**-1.36%**	1.34	48	8.04×10^{08}	**5.71%**
des3	2.09	102	6.18×10^{08}	2.04	95	5.82×10^{08}	2.07	91	6.34×10^{08}	**-0.26%**	2.05	54	6.21×10^{08}	**3.51%**
leon2	6.2	125	4.02×10^{09}	6.04	117	3.77×10^{09}	6.1	116	4.5×10^{09}	**-0.62%**	6.11	50	4.35×10^{09}	**3.01%**
vga_lcd	4.9	131	2.44×10^{09}	4.81	123	2.08×10^{09}	4.9	125	2.57×10^{09}	**-1.29%**	4.9	43	2.52×10^{09}	**3.75%**
b19_10	9.2	139	1.85×10^{10}	9.01	129	1.61×10^{10}	9.08	140	2.03×10^{10}	**-1.19%**	9.03	50	2.1×10^{10}	**9.51%**
b19_20	10.3	145	3.8×10^{10}	9.9	159	3.3×10^{10}	10.1	140	4.1×10^{10}	**-2.4%**	9.95	55	4.3×10^{10}	**8.85%**
Avg.	*3.31*	*135*	3.54×10^{09}	*3.16*	*127*	3.1×10^{09}	*3.26*	*121*	3.87×10^{09}	**-1.38%**	*3.2*	*61*	3.97×10^{09}	**5%**

Table 7.7: Placement results comparison

Table 7.7 illustrates the results of SCPlace, which exploits the TSZSA bounds in conjunction with the hierarchical LP-based slack assignment, and compares against (1) a commercial placement tool, (2) Capo and (3) an alternative statistical placer we created for comparison purposes, which does not use wire bounds. The latter, labeled as "SSA Place", is an iterative, Simulated Annealing (SA) based placer, which performs cell swaps to improve wirelength, mean

delay and sigma. The results are graphically depicted in Figure 7.4.

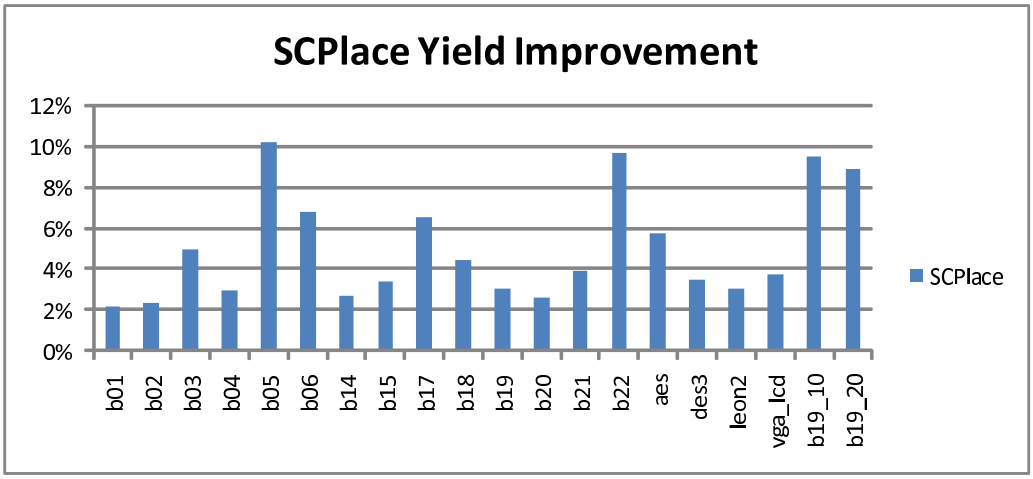


Figure 7.4: SCPlace timing yield improvement

The results of Table 7.7 are post-placement and have been verified against a commercial STA engine, which considers routing delay estimations for a given placement. The correlation of our SSTA engine with Monte-Carlo simulations has been shown in Section 3.1. The slack given for TSZSA is defined by the commercial placer's results, *i.e.* TSZSA is constrained with the mean delay achievable by the commercial placer. Its task is to optimize the sigma without negatively affecting the mean of the circuit's delay. The sigma target was set to 50% of the commercial tool's sigma and was then readjusted accordingly depending on whether it was feasible or not. The results of Table 7.7 indicate for SCPlace an effective yield improvement post-placement over the commercial, non-statistical, Capo and the SSAPlace. SCPlace's yield improvement is of the order of the yield achieved by TSZSA, albeit smaller due to (i) wire delay estimations inaccuracies during TSZSA assignment, (ii) lack of systematic correlation information during TSZSA, (iii) Slack re-assignment by the LP step and (iv) unsatisfied wire bound constraints which are as close as possible to the original constraint, but may not exactly meet it. On average, the yield gained is 5% for the same mean value as a commercial placement tool, which illustrates that the TSZSA's wire bounds, combined with the LP slack assignment are effectively used by SCPlace to produce a legal and valid placement. On the other hand, SSAPlace's results indicate that local sigma optimization does not necessarily imply sigma optimization at primary outputs. This is the case because sigma improvements are not additive, as circuit delay improvements, thus a local sigma improvement at a circuit node can easily be

filtered by a side input's sigma, and it is impossible without a global view to analyze such interactions during placement.

SCPlace's runtime ranges from a few seconds for the smaller benchmarks (`b01-b06`), to a few hours for the larger ones. SCPlace's runtime profile is dominated by the execution time spent for the LP slack assignment step, *i.e.* the LP solver, and the execution time of the SSTA engine.

	Runtime(seconds)					
Circuit	Total	MSSA	TSZSA	Perturbation	Reconstruction	Constructive
b01	27	3	17	4	2	1
b02	23	3	15	0	3	2
b03	26	5	18	0	2	1
b04	28	4	16	5	2	1
b05	29	5	15	4	3	2
b06	24	5	15	0	2	2
b14	134	12	84	17	9	11
b15	134	10	79	19	12	14
b17	141	13	93	15	8	12
b18	1595	18	1305	78	129	65
b19	1954	20	1592	69	201	72
b20	210	14	117	21	42	16
b21	196	13	109	23	39	12
b22	172	14	98	19	27	14
aes	197	12	102	13	51	19
des3	193	13	96	34	29	21
leon2	13859	51	12035	251	1031	491
vga_lcd	13627	55	11593	198	1279	502
b19_10	22897	170	18301	319	3409	698
b19_20	47301	350	36925	702	7592	1732

Table 7.8: SCPlace runtime breakdown

However, none of LP or the SSTA engine is essential in the core of SCPlace, as no timing or slack calculations are needed after the slacks for each wire are known. The core of SCPlace's algorithm is fast and does not require more than a few minutes for any benchmark. Thus, the execution time is dominated by the execution of TSZSA, which requires SSTA, and the number of LP solver iterations, which does not require SSTA, but is slow in itself. Table 7.8 verifies these conclusions. The runtime in Table 7.8 is broken down into the major function calls of SCPlace. These are the SSTA calls, including any incremental SSTA calls, the MSSA and TSZSA calls and the calls to the two auxiliary functions, perturbation and reconstruction. As shown in Table 7.8, the total runtime is dominated by the TSZSA function, which does all the hard work of deriving the bounds which guarantee the desired delay distributions at the circuit's outputs. MSSA is very fast for all circuits and it scales linearly with circuit size. The runtime of the auxiliary functions depends mainly on how often they are called. For some small circuits,

the Perturbation is never called, as all cells can be placed at the first iteration of SCPlace. The Reconstructor is called only when a cell cannot be immediately placed. This means that the runtime required by this function depends on how hard the constraints imposed by TSZSA are. However, the calculations required for each cell during reconstruction are simple enough to allow it to require only a few minutes even for the largest circuits. Constructive placement itself is very fast, as all the hard calculations have been transferred to TSZSA and the auxiliary functions. Thus, the process of finding all candidate locations and choosing one to place any cell requires only a fraction of the total runtime. In total, the results of Table 7.8 show that SCPlace, being a complete placement tool for statistical optimization, can handle circuits of sizes exceeding millions of transistors with total runtime which is considered reasonable for a physical design tool of this caliber.

7.3.2 Routability

In a typical EDA flow, the step following placement is the routing step. Thus, a placer needs to make sure that the placement is routable. Although there is no way to guarantee that a placement is routable before the actual routing information is available, there is one important heuristic which is used by contemporary placers. This is a limit on the congestion imposed by the presence of standard cells. The less congested an area is, the fewer wires need to be routed through this area, making it less likely that the amount of wires requiring routing exceeds the routing capacity of the layout area. As described in Chapter 4, SCPlace guarantees routing by employing density screens which constraint local congestion. We employ density screens in a grid fashion, which bounds the maximum congestion on every segment of the layout, as defined by the overlaid grid. The free space, which SCPlace guarantees at regular space intervals, limits the number of standard cells over a given layout area. This means that the number of nets that arrive or originate from this limited number of standard cells, also does not blow up. The limited number of nets allows the router to effectively route all nets using the available layout area.

We have verified the routability of all SCPlace's placements by performing detailed routing using a state-of-the-art industrial router. For all placements, routing completed successfully without any serious congestion concerns. Figure 7.5 shows an example of a successfully routed benchmark. There are no routing violations, as reported by the industrial tool. For congestion, there is a small area (coloured blue) with relatively larger congestion than the rest of the layout.

However, this amount of congestion is not a problem for the router.

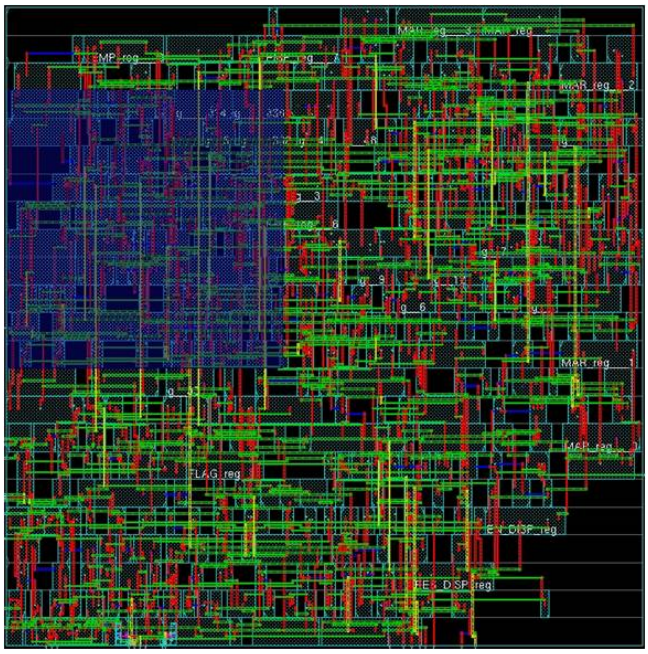


Figure 7.5: **b05** after detailed routing

In the next section we evaluate our post-placement leakage recovery flow which can be applied directly after SCPlace.

7.4 Leakage Recovery Results

In this section we discuss the results of our statistical leakage recovery flow. Our leakage recovery flow can be used after the placement of the circuit is finalized. This means that it can be the next step after SCPlace or any other placer which conforms to industry standards for the description of placement. Our leakage recovery flow works in-place, *i.e.* it only resizes standard cells or replaces them with functionally equivalent ones. The primary aim is to reduce the total statistical leakage of the circuit without affecting the statistical delay of the circuit. Alternatively, our leakage flow can be used to trade-off statistical delay for statistical leakage at the designer's discretion. For comparison purposes, we use a state-of-the-art industrial leakage reduction flow. Specifically, we show that our flow can co-optimize for statistical delay and leakage, while the industrial flow cannot manipulate the uncertainty (sigma) factor of the

delay.

Circuit	Delay (ns)		Leakage (uW)		Area
	Mean	Sigma	Mean	Sigma	(μm^2)
b01	0.23	0.03	0.79	0.05	156
b02	0.2	0.02	0.61	0.04	109
b03	0.34	0.04	2.23	0.21	494
b04	0.38	0.05	7.1	0.47	1478
b05	0.9	0.11	6.75	0.39	1404
b06	0.23	0.03	0.94	0.07	193
b14	3.93	0.27	159	11.2	30580
b15	1.93	0.19	84.5	4.46	19991
b17	1.69	0.16	256	17.3	61764
b18	6.3	0.65	795	53.9	181008
b19	7.71	0.73	2301	107.2	358554
b20	2.33	0.17	309	23.2	58818
b21	2.67	0.22	268	19.1	54326
b22	2.48	0.25	463	35.8	88764
aes	3.3	0.11	310	21.0	63243
des3	2.32	0.28	101	9.4	176172
leon2	6.17	1.02	5397	315.4	859310
vga_lcd	4.90	0.84	4732	286.1	613047
b19_10	9.34	1.29	22956	974.3	1784713
b19_20	10.2	1.37	47839	1543.3	3546980
Avg	*3.38*	*0.38*	*4300*	*171*	*395047*

Table 7.9: Initial placement

The starting point of our experiments is a finalized placement for all benchmarks. Table 7.9 illustrates statistical delay, statistical leakage consumption and area of each benchmark post-placement. All placements were created using a timing-driven state-of-the-art industrial placer which was geared towards timing-efficient placements. This allows for both leakage flows to show their potential as a placement optimized for timing allows more room for leakage improvement. Additionally, using the input from an industrial tool highlights the suitability of our leakage flow to be integrated into an existing EDA flow. The $(\mu + 3\sigma)$ of the initial's placement delay serves as the cutoff point for yield estimation. This means that if the circuit is slower than the initial delay, then a timing yield loss will be reported with respect to the initial statistical delay. Statistical delay and leakage are calculated using our statistical timing analysis engine. The standard deviation of delay in Table 7.9 is considered to be hard constraints for leakage optimization for our flow. This means that our flow is not allowed to optimize any standard cells in such a way that the sigma of circuit's delay is increased.

Circuit	Industrial Flow							Proposed Flow						
	Delay (ns)		Leakage (uW)		Area (μm^2)	Timing Yield Loss	E(X) Leakage Recovery	Delay (ns)		Leakage (uW)		Area (μm^2)	Timing Yield Loss	E(X) Leakage Recovery
	Mean	Sigma	Mean	Sigma				Mean	Sigma	Mean	Sigma			
b01	0.23	0.03	0.56	0.06	138	**0%**	**29.1%**	0.23	0.03	0.56	0.06	139	**0%**	**29.1%**
b02	0.2	0.03	0.44	0.04	96	**2.3%**	**27.9%**	0.2	0.03	0.45	0.04	96	**2.3%**	**26.2%**
b03	0.35	0.04	1.82	0.23	475	**0.3%**	**18.4%**	0.35	0.04	1.85	0.23	489	**0.3%**	**17%**
b04	0.39	0.05	6.07	0.51	1387	**0.3%**	**14.5%**	0.39	0.05	6.16	0.51	1432	**0.3%**	**13.2%**
b05	0.92	0.13	5.17	0.47	1332	**0.9%**	**23.4%**	0.92	0.11	5.38	0.46	1349	**0.2%**	**20.3%**
b06	0.24	0.03	0.69	0.08	169	**0.4%**	**26.6%**	0.24	0.03	0.71	0.08	173	**0.4%**	**24.5%**
b14	3.92	0.43	132	11.6	28943	**2.9%**	**17%**	3.95	0.27	141	11.4	29539	**0.1%**	**11.3%**
b15	2.02	0.27	67	4.59	19021	**3.8%**	**20.7%**	1.98	0.19	78.3	4.55	19702	**0.4%**	**7.3%**
b17	1.86	0.32	229	19.1	57682	**16.9%**	**10.5%**	1.71	0.16	240	18.3	59681	**0.2%**	**6.3%**
b18	6.52	1.03	538	61.3	172390	**4.75%**	**32.3%**	6.32	0.66	550	58.3	174468	**0.1%**	**30.8%**
b19	7.94	1.31	1549	129.3	339051	**6.8%**	**32.7%**	7.73	0.74	1629	119.1	343529	**0.1%**	**29.2%**
b20	2.41	0.38	202	25.8	51803	**12.92%**	**34.6%**	2.33	0.18	221	23.7	53655	**0.23%**	**28.5%**
b21	2.71	0.44	219	21.1	50913	**8.08%**	**18.3%**	2.67	0.23	235	20.4	52976	**0.21%**	**12.3%**
b22	2.53	0.41	309	37.4	79104	**4.46%**	**33.2%**	2.49	0.26	338	34.6	81302	**0.23%**	**27%**
aes	3.51	0.24	179	24.2	55591	**31.2%**	**42.2%**	3.31	0.11	190	22.9	57224	**0.1%**	**38.7%**
des3	2.34	0.62	65	10.1	165598	**9.34%**	**35.6%**	2.33	0.28	78	9.8	170047	**0.12%**	**22.8%**
leon2	6.37	1.82	4284	282	832597	**3.1%**	**20.6%**	6.19	1.05	4501	289	836721	**0.4%**	**16.6%**
vga_lcd	5.23	1.45	4039	217	598639	**6.5%**	**14.6%**	4.93	0.85	4251	230	603498	**0.1%**	**10.2%**
b19_10	9.41	2.14	15610	725	1593198	**3.8%**	**32%**	9.37	1.30	17409	787	1645601	**0.1%**	**24.1%**
b19_20	10.2	2.41	34302	1487	3264208	**4.4%**	**28%**	10.2	1.38	36741	787	3439847	**0.1%**	**23%**
Avg	*3.46*	*0.68*	*3087*	*153*	*365617*	**6.16%**	**25.6%**	*3.39*	*0.4*	*3330*	*121*	*378573*	**0.3%**	**20.9%**

Table 7.10: Timing yield comparison

7.4.1 Leakage Recovery vs Industrial

Table 7.10 illustrates the leakage recovery results and contrasts the results of a commercial, non-statistical, leakage recovery tool with the results obtained using our statistical, (mean, sigma) preserving leakage optimization flow. The mean delay for both flows is identical. These results indicate that our flow achieves comparable leakage recovery to the industrial flow. The average gain in the expected value of leakage is 20.9%, ranging from about 6% to 38.7%. For some of the largest circuits, `b18` and `b19`, we achieve above average leakage gains, *i.e.* 30.8% and 29.2% respectively. Our flow also succeeds at preserving the required mean and sigma, which is not the case in the industrial flow. The use of relative constraints that our flow employs, guarantees that the sigma at the outputs is not disturbed. During leakage optimization only very small fluctuations occur, in both mean and sigma. On the other hand, the commercial tool performs more aggressive gate sizing, resulting in greater optimization for leakage, but fails to control sigma. The timing yield losses for the two approaches are shown in Figure 7.6.

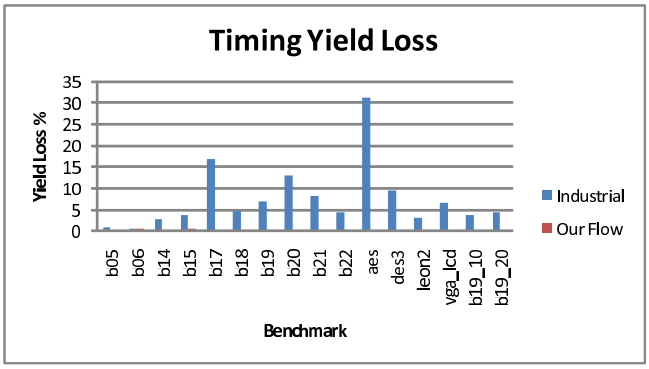


Figure 7.6: Timing yield loss after leakage reduction

7.4.2 Delay-Leakage Tradeoff

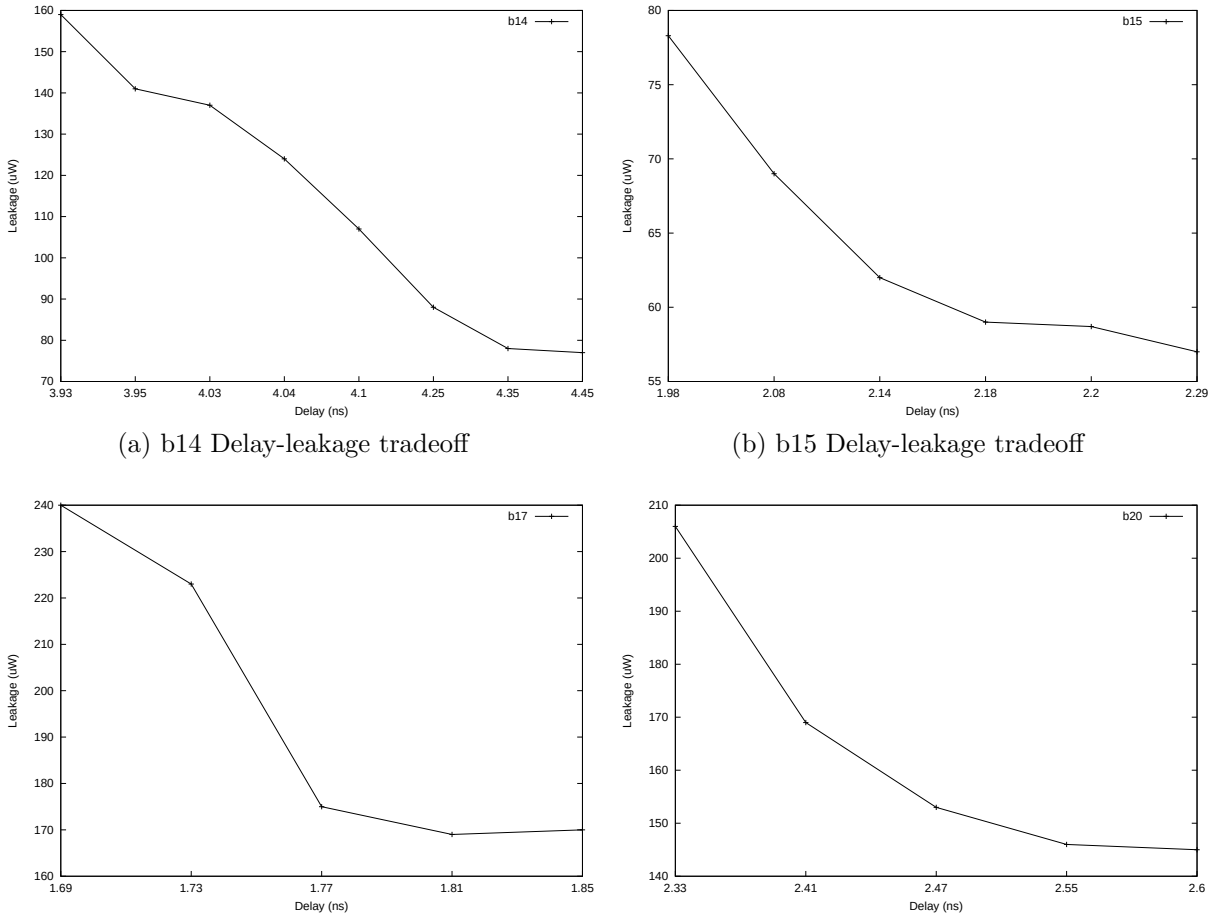
For the second set of results, we demonstrate the effectiveness of our flow on trading off delay for leakage and vice-versa. By adding slack to the initial placement, TSZSA can identify a slack assignment which distributes this additional slack onto the gates, while at the same time, respecting the relative constraints guaranteeing the same sigma value at the outputs. Additional slack, thus, allows for more aggressive downsizing of gates.

For producing Pareto curves, we set the mean constraint, as the initial mean delay plus the additional slack, whereas the sigma constraint is the post-placement sigma. Then, we optimize the circuits with our leakage recovery flow.

Figures 7.7a, 7.7b, 7.7c and 7.7d show the tradeoff for delay-leakage for benchmarks `b14`, `b15`, `b17` and `b20` respectively, where the horizontal axis shows the expected value of delay and the vertical axis shows the expected value of leakage. As shown in Figures 7.7a, 7.7b, 7.7c and 7.7d, in all cases, our flow could effectively utilize the extra slack for more aggressive leakage reduction, until the extra slack reaches a point where it can no longer be traded for less leakage. This point is expected to be reached, as given enough slack, almost all gates will have reached their smallest size, making the addition of more slack redundant.

7.4.3 Leakage Recovery Runtime

In this section we evaluate the runtime results for our leakage recovery flow. We compare our runtime against the industrial flow and breakdown the runtime to its contributing functions. Table 7.11 shows the runtime results obtained from the GNU GPROF profiling tool.



(a) b14 Delay-leakage tradeoff

(b) b15 Delay-leakage tradeoff

(c) b17 Delay-leakage tradeoff

(d) b20 Delay-leakage tradeoff

Figure 7.7: Delay-leakage tradeoff

As shown in Table 7.11, the industrial flow performs optimization in much shorter time than our flow. However, this is expected as the industrial flow does not support statistical optimization and thus, the calculations required at each step are fewer and simpler. As a result, the industrial flow outperforms our flow with respect to runtime, but out flow consistently outperforms the industrial flow with respect to statistical optimization.

The majority of runtime is dominated by our statistical slack assignment algorithms and most notably TSZSA. This is expected, as this function performs all the hard work of identifying which cells can afford extra delay and how much this delay is. A significant amount of time is also devoted to incremental SSTA, which updates the timing graph allowing for more accurate on-the-fly optimization. The process of substituting each gate with its best alternative is

	Runtime(seconds)					
		Statistical Flow				
Circuit	Industrial	Total	MSSA	TSZSA	Inc SSTA	Replace Gate
b01	7	26	3	17	5	1
b02	8	24	3	15	5	1
b03	7	28	5	18	4	1
b04	7	26	4	16	5	1
b05	7	26	5	15	5	1
b06	5	24	5	15	3	1
b14	22	111	12	84	12	3
b15	24	110	10	79	18	3
b17	19	123	13	93	13	4
b18	604	1744	18	1305	382	39
b19	843	2068	20	1592	409	47
b20	32	160	14	117	21	8
b21	31	150	13	109	19	9
b22	49	147	14	98	27	8
aes	41	149	12	102	25	10
des3	44	143	13	96	23	11
leon2	2983	13254	51	12035	917	251
vga_lcd	3014	12945	55	11593	1042	255
b19_10	5941	21036	170	18301	1958	607
b19_20	12093	43798	350	38596	3381	1471

Table 7.11: Leakage optimization runtime breakdown

relatively fast due to the preprocessing step which maintains a list of candidate replacement gates for all library gates. The total runtime of our flow ranges from a few seconds for the smaller circuits to a few hours for the circuits with over one million transistors. Since our flow achieves closure for all benchmarks, runtimes of a few hours are considered acceptable in industrial tools performing large-scale optimization such as full-chip leakage reduction.

In the next section we turn our focus to the evaluation of our placement tool which can handle asynchronous circuits.

7.5 CPlace Results

In this section we discuss the results of CPlace. The aim of CPlace is to place the cells of asynchronous circuits in a way that guarantees the performance of the placed circuit and the satisfaction of timing assumptions required for the correct operation of the circuit. We validate the superiority of CPlace against a state-of-the-art industrial placer and show that CPlace can succesfully place all asynchronous circuits without violating any critical assumptions required for asynchronous operation, which is not the case for the industrial tool. CPlace also creates placements which, in terms of performance, compare reasonably to the placements of

the industrial tool.

7.5.1 CPlace vs Industrial and Capo

IWLS Circ.	ZWD	Constraint	Capo [10]			Industrial Placer			CPlace		
			Delay(ns)	Slack(ns)	HPWL	Delay(ns)	Slack(ns)	HPWL	Delay(ns)	Slack(ns)	HPWL
b01	0.21	0.32	0.23	0.09	6.96×10^{05}	0.21	0.11	6.46×10^{05}	0.22	0.10	6.98×10^{05}
b02	0.20	0.30	0.37	-0.07	4.46×10^{05}	0.33	-0.03	5.41×10^{05}	0.32	-0.02	4.49×10^{05}
b03	0.35	0.53	0.61	-0.08	1.87×10^{06}	0.56	-0.03	2.23×10^{06}	0.56	-0.03	1.87×10^{06}
b04	0.40	0.60	0.82	-0.22	7.07×10^{06}	0.75	0.15	6.88×10^{06}	0.77	-0.17	7.11×10^{06}
b05	0.66	0.99	1.23	-0.24	8.46×10^{06}	1.14	-0.15	8.88×10^{06}	1.20	-0.21	8.54×10^{06}
b06	0.24	0.36	0.49	-0.13	2.39×10^{07}	0.44	-0.08	8.72×10^{06}	0.44	-0.08	2.41×10^{07}
b14	1.63	2.45	3.58	-1.13	2.83×10^{08}	3.45	-1.00	2.55×10^{08}	3.47	-1.02	2.98×10^{08}
b15	1.18	1.77	1.91	-0.14	1.90×10^{08}	1.78	-0.01	1.59×10^{08}	1.75	0.02	1.97×10^{08}
b17	1.23	1.85	1.78	0.07	6.17×10^{08}	1.65	0.20	5.11×10^{08}	1.68	0.17	6.19×10^{08}
b18	3.58	5.37	6.03	-0.66	1.61×10^{09}	5.87	-0.50	1.37×10^{09}	5.93	-0.56	1.78×10^{09}
b19	5.22	7.83	7.65	0.18	1.82×10^{09}	7.13	0.70	1.51×10^{09}	7.21	0.62	2.05×10^{09}
b20	1.67	2.51	2.03	0.48	6.36×10^{08}	1.84	0.67	5.66×10^{08}	1.85	0.66	6.54×10^{08}
b21	1.78	2.67	2.44	0.23	5.71×10^{08}	2.43	0.24	5.02×10^{08}	2.45	0.22	6.12×10^{08}
b22	1.75	2.63	2.83	-0.20	9.38×10^{08}	2.58	0.05	7.83×10^{08}	2.67	-0.04	9.85×10^{08}
aes	0.95	1.43	1.41	0.02	7.65×10^{08}	1.34	0.09	7.00×10^{08}	1.34	0.09	8.04×10^{08}
des3	1.08	1.62	2.09	-0.47	6.18×10^{08}	2.04	-0.52	5.82×10^{08}	2.05	-0.53	6.21×10^{08}
leon2	4.9	7.35	6.2	1.3	4.02×10^{09}	6.04	1.31	3.77×10^{09}	6.11	1.24	4.35×10^{09}
vga_lcd	3.5	5.25	4.9	0.35	2.44×10^{09}	4.81	0.44	2.08×10^{09}	4.9	0.35	2.52×10^{09}
b19_10	6.8	10.2	9.2	1.0	1.85×10^{10}	9.01	1.19	1.61×10^{10}	9.03	1.17	2.1×10^{10}
b19_20	7.2	10.8	10.2	0.6	3.8×10^{10}	9.9	0.9	3.3×10^{10}	10.1	0.7	4.3×10^{10}
Avg.	2.22	3.33	2.94	0.04	3.55×10^{09}	3.17	0.18	3.1×10^{09}	3.2	0.13	3.97×10^{09}

Table 7.12: Synchronous placement results comparison

Table 7.12 illustrates post-placement results obtained by CPlace. We compare and contrast CPlace with (i) Capo [66, 10], a well-known academic placer and (ii) a state-of-the-art, mature, industrial placer, the name of which we cannot disclose. Capo was selected as a reference academic placer, as it supports technology-mapped circuits and *LEF/DEF* formats, similar to CPlace. However, it should be noted that Capo does not support timing-constraints. We illustrate in Table 7.12 the obtained mean and the HPWL (Half-Perimeter Wire Length) for Capo, the industrial placer and CPlace. HPWL is measured in *LEF/DEF* database units (2000 units/μm) by Capo's *WireLengthCalculator*. The results in Table 7.12 have been verified against a very commonly used, STA golden-sign off engine, which considers routing delay estimations for a circuit placement. The slack assigned to each placer was 50% of the zero-wire delay result for each circuit.

We have verified that the placements produced by CPlace firstly satisfy the majority of the wire bound constraints and secondly that are routable. The latter was verified by running trial routing and congestion analysis in a commercial back-end tool.

The results of Table 7.12 show that CPlace can effectively handle the synchronous circuits used in this set of experiments, yielding about the same delay as the industrial placer. CPlace's HPWL results indicate a total wirelength comparable to Capo, even though CPlace focuses secondarily on wirelength.

7.5.2 QDI Satisfaction

In this section, we compare CPlace against a state-of-the-art industrial placer, in terms of their ability to satisfy both timing and Quasi-Delay-Insensitive (QDI) constraints. This type of constraints enforce delay insensitivity for the asynchronous design, *i.e.* the circuit should work correctly with arbitrary delays on the wires except for isochronic forks. Isochronic forks refer to the case where a gate has more than one fanouts. In this case, the wires connecting the gate to all its fanouts must have similar delay, forming an isochronic fork. The satisfaction of isochronic forks is key to CPlace. CPlace creates specific wire bounds which enforce that the lengths of each leg in each isochronic fork will be similar after placement. This property enables CPlace to offer superior results compared to its industrial counterpart, which cannot effectively bound the absolute difference in delay for each leg of each isochronic fork. The framework used for determining the degree of QDI satisfaction for both placers is shown in Figure 7.8.

Absolute and relative constraints are derived by TSE asynchronous timing analysis. The industrial placer can handle absolute constraints using maximum delay bounds (`set_max_delay`, `set_min_delay` or `create_clock` Synopsys Design Constraints (SDC) constraints appropriately). As it does not support relative constraints, each relative constraint is converted into a minimum and maximum delay bound (`set_max_delay`, `set_min_delay` SDC constraints), in order to force the industrial tool to match the delay of each leg in an isochronic fork, within a given margin. We define this margin as the delay of a drive 1 (`D1`) inverter for the target library. Thus, if the difference in the delays of the legs in each isochronic fork is not greater than that margin, we assume that QDI is satisfied.

CPlace, handles both the relative and the absolute constraints as wire bounds, which stem from the solution of the LP problem. After constraints have been setup for each placer, we run the tools and produce the placements. Then, a point-to-point STA using a golden sign-off STA engine is performed validating both absolute and relative constraints. Finally, we report the number of constraints that are violated by each placement.

Table 7.13 summarizes placement results for the asynchronous benchmarks.

Circuit	Area $(um)^2$	Cplace viol	Indust. viol	Cplace viol	Indust. viol	Relative bounds	Pre-pl TSE period (ns)	Period bound (ns)	Slack Indust. (ns)	Slack CPlace (ns)	HPWL Indust. (um)	HPWL CPlace (um)
		25ps const		**0.025ps const**								
c3dec2	42	0	0	3	8	11	0.27	0.41	0.05	0.04	1.23×10^{05}	1.31×10^{05}
ccc	16	0	0	0	1	7	0.16	0.24	0.05	0.04	5.78×10^{04}	6.02×10^{04}
chu133	16	0	0	0	0	6	0.16	0.24	0.05	0.06	5.78×10^{04}	5.69×10^{05}
chu150	25	0	0	0	0	2	0.20	0.30	0.05	0.05	7.44×10^{04}	7.72×10^{04}
converta	47	0	0	3	4	10	0.20	0.30	0.04	0.03	2.17×10^{05}	2.29×10^{05}
half	38	0	0	1	2	5	0.24	0.36	0.04	0.05	1.16×10^{05}	1.13×10^{05}
mmu	85	0	0	5	7	24	0.73	1.1	0.12	0.07	3.02×10^{05}	2.99×10^{05}
mp forward pkt	42	0	0	0	0	5	0.10	0.15	0.03	0.03	1.25×10^{05}	1.32×10^{05}
nak pa	52	0	0	1	3	7	0.17	0.26	0.01	0.02	2.05×10^{05}	1.99×10^{05}
nowick	20	0	0	1	2	4	0.03	0.04	0.00	0.00	7.74×10^{04}	7.86×10^{04}
rcv setup	20	0	0	0	0	2	0.08	0.12	0.02	0.02	6.62×10^{04}	6.71×10^{05}
rpdft	41	0	0	1	3	4	0.06	0.10	0.03	0.03	1.47×10^{05}	1.43×10^{05}
sbuf read_ctl	30	0	0	0	1	6	0.08	0.12	0.03	0.03	9.44×10^{04}	9.38×10^{04}
sbuf send_ctl	61	0	0	1	3	8	0.16	0.24	0.04	0.05	2.05×10^{05}	2.19×10^{05}
seq mix	73	0	0	0	3	10	0.32	0.47	0.04	0.02	2.9×10^{05}	2.87×10^{05}
trimos send	132	0	0	4	11	27	0.32	0.47	0.03	0.02	4.37×10^{05}	4.62×10^{05}
var1	36	0	0	3	4	12	0.17	0.26	0.03	0.03	1.11×10^{05}	1.19×10^{05}
vbe10b	136	0	0	5	18	40	0.41	0.62	0.03	0.01	4.88×10^{05}	4.91×10^{05}
vbe5b	36	0	0	0	2	7	0.14	0.21	0.04	0.04	1.29×10^{05}	1.37×10^{05}
vbe5c	28	0	0	1	1	4	0.13	0.20	0.03	0.04	1.02×10^{05}	1.11×10^{05}
vbe6a	148	0	0	4	16	29	0.31	0.46	0.03	0.01	4.53×10^{05}	4.56×10^{05}
wrdatab	100	0	0	5	9	34	0.62	0.93	0.15	0.09	4.13×10^{05}	4.12×10^{05}
xyz	32	0	0	0	2	7	0.17	0.26	0.03	0.03	1.03×10^{05}	1.05×10^{05}
seq mix 10000	34944	0	0	23	151	843	7.18	10.77	1.43	0.74	3.9×10^{06}	4.17×10^{06}
half 10000	39321	239	1038	382	4741	18549	217	326	41	7	4.2×10^{06}	4.61×10^{06}
mmu 10000	43417	102	4715	2302	12237	25084	402	603	54	19	4.3×10^{06}	4.58×10^{06}
ccc 15000	33587	129	4272	4418	17835	32251	458	687	32	4	1.90×10^{08}	2.06×10^{08}
converta 64000	193331	1089	18549	39043	65062	126967	1017	1523	234	132	7.55×10^{08}	7.94×10^{08}
dlx desync	52276	957	8453	2488	16768	33981	4.39	6.59	0.34	0.12	1.07×10^{07}	1.19×10^{07}

Table 7.13: Comparison of relative constraints violations for CPlace and the industrial placer

We used a set of small asynchronous benchmarks, derived from their respective STGs and mapped with Petrify[18] into a $65nm$ library. The results for the initial set of benchmarks are shown in the upper part of the table. Due to the very small area size of all benchmarks,

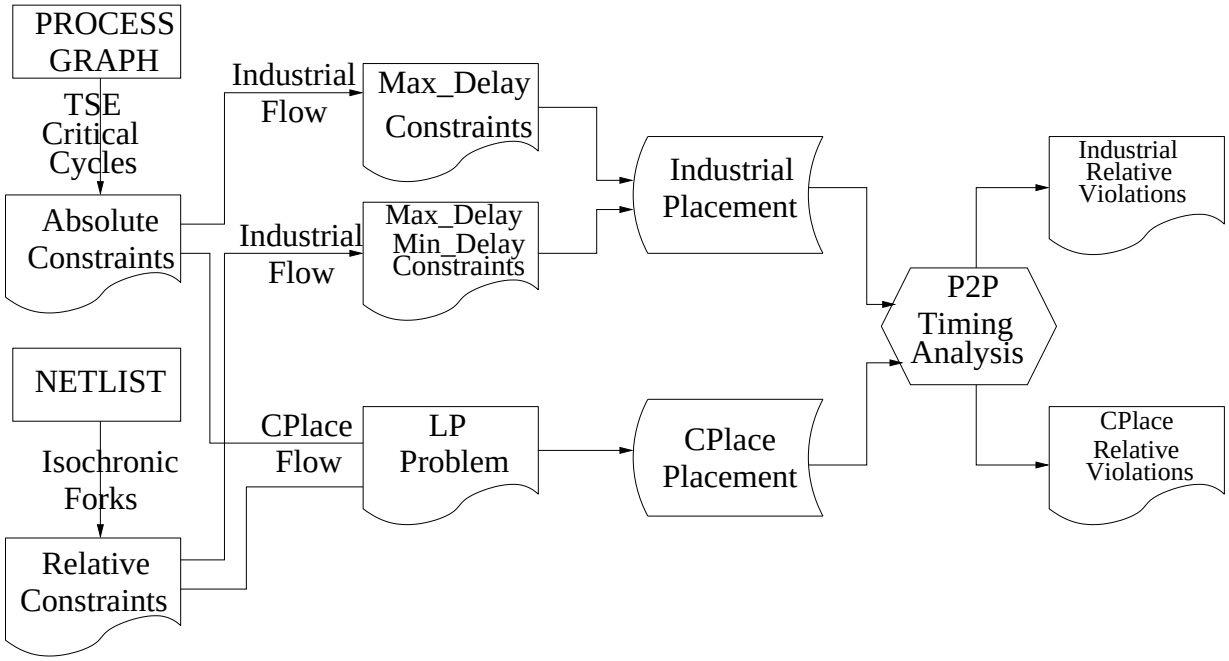


Figure 7.8: Experimental flow

the bound for isochronic forks is easily met by both the industrial placer and CPlace. This is expected, as the delay of a fast inverter, is much greater than the delay of any wire in a reasonably optimized placement of such size. However, if we set a much tighter bound on isochronic forks, we observe that for almost all benchmarks, CPlace results in fewer relative constraints violations than the industrial placer. Although the constraints are unrealistic at this point, this result shows the potential of CPlace. The absolute constraints are easily met for all small circuits, which is expected, as the placer can use short wires for all connections. Although the industrial placer results in higher remaining slack for almost all circuits, both the industrial placer and CPlace can meet the absolute constraints without much difficulty. CPlace uses a part of remaining slack to fix the relative constraints, which is the reason for CPlace's remaining slack to be smaller than that of the industrial placer, while the number of relative constraints violations of CPlace's is smaller for almost all circuits compared to the industrial placer's violations.

In order to have a more realistic set of benchmarks, we selected a number of the small circuits and created larger ones, by instantiating multiple times the smaller circuit. Thus, we created circuits with about 10,000 cells, like `half_10000` which is created from the small circuit `half` (a half handshake) by connecting it in a pipeline multiple times. We also created

a larger circuit, of about 64,000 cells `converta_64000`, which is multiple instances of a two-to-four handshake converter. The replicas are created by placing multiple times the originating instance side-by-side, *i.e.* by connecting the outputs of the previous stage to the inputs of the next stage. The results for the larger circuits are shown in the middle section of Table 7.13. This set of results clearly shows that even for the realistic constraint of an inverter's delay, CPlace yields much fewer relative constraint violations than the industrial placer. This proves the ability of CPlace to handle both absolute and relative constraints, thus maintaining the delay insensitive properties of our circuits, while at the same time, meeting the performance constraints.

The final benchmark, shown in the bottom section of Table 7.13 shows a circuit which has been created using the "desynchronization" approach [19]. This circuit is a real design with reasonably large size. The results show that even for this circuit, CPlace does better than the industrial placer in conserving the relative constraints.

The results of Table 7.12 and 7.13 show that (i) CPlace can effectively handle the synchronous circuits used in this set of experiments, yielding about the same delay as the industrial placer, (ii) CPlace's HPWL results indicate a total wirelength comparable to Capo, even though CPlace focuses secondarily on wirelength and (iii) the synchronous placers cannot effectively handle relative constraints necessary for QDI implementations, which highlights the advantage of CPlace in that respect.

7.5.3 CPlace Runtime

Table 7.14 shows the runtime breakdown for CPlace and compares against Capo and the industrial placer. The fastest placer is the industrial, which however, does not support relative constraints for satisfying QDI constraints. Capo only optimizes for HPWL, so it performs poorly compared to CPlace and the industrial placer in terms of handling asynchronous circuits. The industrial placer is faster than CPlace, but as shown earlier, this comes at the cost of failing to satisfy all QDI constraints. CPlace's runtime profile is dominated by the execution time spent for the LP slack assignment step, *i.e.* the LP solver, and the execution time of the timing analysis engine.

Runtime(seconds)							
	Industrial	Capo	CPlace				
Circuit			Total	TSE	Constraints	LP	Constructive
c3dec2	1	1	4	1	1	1	1
ccc	1	1	4	1	1	1	1
chu133	1	1	4	1	1	1	1
chu150	1	1	4	1	1	1	1
converta	1	1	4	1	1	1	1
half	1	1	4	1	1	1	1
mmu	1	1	4	1	1	1	1
mp_forward_pkt	1	1	4	1	1	1	1
nak_pa	1	1	4	1	1	1	1
nowick	1	1	4	1	1	1	1
rcv_setup	1	1	4	1	1	1	1
rpdft	1	1	4	1	1	1	1
sbuf_read_ctl	1	1	4	1	1	1	1
sbuf_send_ctl	1	1	4	1	1	1	1
seq_mix	1	1	4	1	1	1	1
trimos_send	1	1	4	1	1	1	1
var1	1	1	4	1	1	1	1
vbe10b	1	1	4	1	1	1	1
vbe5b	1	1	4	1	1	1	1
vbe6a	1	1	4	1	1	1	1
wrdatab	1	1	4	1	1	1	1
xyz	1	1	4	1	1	1	1
seq_mix_multi	1	1	4	1	1	1	1
sm_10000	47	731	1649	107	17	1513	12
half_10000	42	649	1409	115	19	1263	12
mmu_10000	46	705	1557	116	17	1410	14
ccc_10000	45	920	2041	108	17	1902	14
converta_64000	110	7411	19461	839	76	18437	109
dlx_desync	18	1043	2449	49	9	2383	8

Table 7.14: CPlace runtime breakdown

Chapter 8

Conclusions

As technology continues to shrink, new challenges emerge forcing contemporary EDA tools to their limits. It is becoming apparent that EDA flows need to adapt to new characteristics of technologies in deep-submicron or to develop entirely new approaches.

One of the main challenges for contemporary EDA tools is the increasing importance of process and operating variations, affecting the speed and power characteristics of individual circuit elements, leading to uncertainty in timing and power performance of the whole circuit. According to the ITRS, designers are likely to adopt two parallel but different approaches. The first is statistical analysis of timing and power and the second is design solutions like the adaptation of speed-independent circuits, most commonly known as asynchronous circuits. This, coupled with the fact that integration will continue to scale will highlight the need for EDA tools fitted for the new challenges. However, still to date, there is lack of EDA tools that can optimize statistically or are ready to be used for speed independent circuits.

This thesis has identified the aforementioned challenges and has proposed efficient solutions through the research and development of appropriate EDA tools and flows.

We have addressed the challenge of the emergence of statistical methods by developing our own statistical timing analysis engine and statistical leakage estimation tool. Both are ready to incorporate statistical models, but can also derive statistical distributions by fitting the values from contemporary corner libraries. We have developed a placement tool, called SCPlace, described in Chapter 4 which can perform placement using SSTA in its inner loop for optimization. SCPlace utilizes novel wire bounds, which aim at propagating delay distributions with bounded mean and standard deviation. For this purpose, we have developed two novel

statistical slack assignment algorithms. The first, called MSSA (c.f. Section 3.3) can find a slack allocation that propagates the delay distribution with the minimum sigma to the circuit's outputs. The second, called TSZSA (c.f. Section 3.4) can propagate appropriate delay distributions that meet targets on both mean and standard deviation at all internal points of the circuits, including the endpoints. We show that SCPlace achieves superior results compared to a state-of-the-art academic placer and a state-of-the-art industrial placer, with respect to statistical timing optimization. We have tackled the problem of the ever increasing importance of leakage power by developing a novel optimization flow which can statistically optimize for leakage. By performing statistical timing and leakage optimization simultaneously, we show that our flow compares favourably to a state-of-the-art industrial leakage optimization flow. Additionally, our flow can trade-off statistical timing for leakage at the designer's discretion. For the design-oriented solution of speed-independent circuits, we have developed the first placer which can efficiently handle asynchronous circuits. CPlace, described in Chapter 6, is the first placer which can place asynchronous circuits in a manner which is performance-efficient and also guarantees the critical assumptions necessary for the circuit's speed independence.

8.1 Future Work

Research efforts for this thesis have uncovered a number of issues that are likely to become important in future EDA technologies and can be addressed as extensions of this thesis.

The first issue is uncertainty in delay of wires. Although wires are less prone to variability due to the different and less complex fabrication procedure that is employed in industry, their delay characteristics are starting to exhibit fluctuations. Thus, statistical methods are likely to be used for the characterization of wire delay in a similar way as is the case for gates. Our statistical slack assignment algorithms can be extended in order to encapsulate statistical analysis of wire delay in order for more robust calculation of the optimal slack allocation.

The second issue is that, as leakage current is becoming increasingly important, the variance of its statistical estimation will become more interesting, as currently is the case for statistical delay. Our leakage reduction flow currently focuses on optimizing for the expected value of leakage, rather than its variance. As an extension, a slack allocation which targets variance optimization of leakage can improve the statistical gains on leakage.

The third issue is due to the emergence of asynchronous circuits as an important fraction of contemporary digital designs. Currently, asynchronous circuits account mainly for control

circuits, which are significantly smaller than datapath circuits. With this in mind, our CPlace tool has not been coupled with a hierarchical approach which could enhance its scalability. We have shown that CPlace can efficiently handle circuits of several tenths of thousands of cells, but with the incorporation of a hierarchical approach, this number could scale to a few millions of standard cells.

Appendix A

Synchronous and Asynchronous Timing Models

In this section we describe the models employed and the assumptions commonly used by timing analysis engines regarding the timing behaviour of a circuit and the elements it consists of. We distinguish among static timing analysis, asynchronous timing analysis and statistical timing analysis models.

A.1 Static Timing Analysis Models

Any static timing analysis engine requires at least one timing characterization of all elements that appear in the circuit. Additional characterizations may be present if the elements exhibit different timing behaviour with respect to fluctuations in their fabrication or operating conditions. One characterization must be present for at least the gates consisting of the circuit. Additionally, characterization of wire delays can enhance the accuracy of timing analysis. In the absence of detailed characterization of wire delays, an approximation model, called wire-delay model is required. This approach is widely adopted both in industry and academia, especially before routing has been completed and thus, the actual wire characteristics are unknown.

Characterization of gates typically consists of lookup-tables, which outline the basic notions related to timing. These are rise and fall transition delays, which correspond to the time required for any output of a gate to rise or fall caused by a rise or fall from any input of the same gate. As shown in Figure A.1 slower rise (or fall) times for the input, or larger

driving capacitance for the output slows the rise (or fall) transition of the output. Different values are given for different scenaria describing possible values for the input's transition time and the output's driving capacitance. These values are given in table format and must be looked up while timing analysis is performed, as the transition delay of an output is depends proportionally to the transition delay of an input and its driving capacitance.

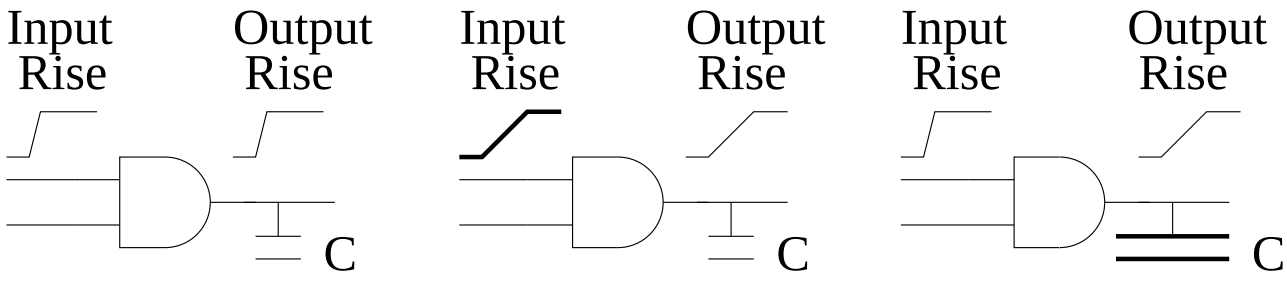


Figure A.1: Rise (fall) transition depending on input rise (fall) transition and output driving capacitance

In contemporary designs, fluctuations are expected both during the manufacturing process and the operating conditions of the chip. These alter the nominal, "typical" characteristics of any element in the circuit. Thus, different characterizations are provided for a number of process and operating scenaria. These are called "corners". Each corner describes the timing behaviour of all circuit elements using the aforementioned lookup tables. Timing analysis performed on more than one corners is called multi-corner analysis.

Timing analysis of a single element consists of calculating three key metrics, which are expressed as real numbers. These are the arrival time, the required time and the slack. Arrival time is the time at which the value at a single point is stabilized. In case of a gate input, it is the time the input has a steady value, as defined by its driver. In the case of a gate output, it is the time at which the gate has evaluated its value based on the values at the gate's inputs. Required time refers to the time the values should be stabilized at any point in the circuit. Required times are typically inferred by timing constraints. Slack is the difference of required time minus arrival time. STA uses these metrics to evaluate the delay and the slack of the circuit.

The circuit is modeled as a directed acyclic graph (DAG), where all the inputs and outputs of the circuit and of all gates are the nodes of the graph and the wires are the edges. As STA is typically applied to synchronous circuits, the transformation of the circuit to its corresponding DAG is straightforward, as a synchronous circuit does not contain any cycles. The delay of the circuit is modeled after the circuit's "critical path". This is essentially the longest path of the

circuit's DAG. The arrival and required times of the last node in the critical path, as well as its slack define the values of these metrics for the whole circuit.

Next, we discuss the extension of static timing analysis models into statistical timing analysis models.

A.2 Statistical Timing Analysis Models

Statistical timing analysis is an extension of static timing analysis, the difference being that the values stored at each node are not real numbers, but random distributions. This eliminates the need for multi-corner analysis, as information of all corners is encapsulated into the random distribution of each circuit element's delay. However, all the other notions of STA are applicable to SSTA.

Rise and fall transitions are similarly calculated in SSTA with the use of lookup-tables. Each entry of the lookup-table is a random distribution referring to the expected transition time for the output of a gate, given the delay distribution corresponding to the transition time of one of its inputs and the output's driving capacitance. The difference from STA is that the inferred transition time is a random distribution. There are two ways to infer the statistical delay of a gate. The first, is to assume a specific statistical distribution for the delay and then to fit the values from a number of given technology library corners into the assumed distribution. The second, is to use a technology library which already provides delays statistically. There has been a recent advance towards this direction by the industry in the form of Effective Current Source Model (ECSM) [20]. This is an open model, aspiring to be the industry standard for statistical characterization of technology gates. However, this model has not yet been developed fully, so this option is yet to be available for statistical timing engines.

Arrival, required times and slack can also be calculated for each gate. They too, are random distributions. Slack is the difference of required time minus arrival time. Since SSTA uses random distributions, this calculation may not be straightforward and depends on the exact distributions used. In the next sections we will show the typical distributions used and how this calculation is done.

Critical path is also defined in SSTA as the longest path in the circuit's timing DAG. The delay of the circuit refers to the delay of the path, which is now expressed as a random distribution. This allows for the introduction of a new metric for the timing performance of the circuit, which is the statistical timing yield. Given a delay constraint and a delay

distribution, the timing yield is the probability that the statistical delay meets the constraint. The calculations required for deriving timing yield depend again on the exact distributions used.

Additionally, SSTA employs variation models from which the exact expressions for the delay distributions are derived. Also, since statistical functions require a correlation value between their statistical operands, correlation models are also used in SSTA. In the next section we review the state-of-the-art variation and correlation models.

A.3 Variation and Correlation Models

Early works have proposed ways to handle variation, be it either inter-die or intra-die [13, 80, 61]. However, these approaches require a-priori knowledge of a variation model [3, 84], which has been extracted from real data from fabricated chips. This approach is not always practical in EDA, especially for companies or research groups which do not have access to a fabrication facility. Experimental data measurement on the other hand, can have accuracy limitations which may directly impact the accuracy of variation model. For this reason, there have been works which try to approximate the characteristics of variation without relying on physical data. Next, we describe some of these methods, each of them focusing on different characteristics of variation.

Xiong et.al [82] describe a variation model using probability density functions (pdf). Variation is described as a random variable F, which has three components; a chip component Y, a wafer component X and a random component Z. It holds:

$$F = f_0 + X + Y + Z, \tag{A.1}$$

where f_0 is variation's mean. Variation components X and Y are independent. Each chip in the same wafer is assumed to experience the same amount of X variation. Since variation is a random variable, it has variance σ_F^2, for which it holds:

$$\sigma_F^2 = \sigma_X^2 + \sigma_Y^2 + \sigma_Z^2, \tag{A.2}$$

where σ_X^2 , σ_Y^2 and σ_Z^2 are the variance coefficients of variation of wafer, chip and random components respectively. The covariance of two physical locations is also defined. If i and j are two locations with variation F_i and F_j respectively, then the covariance is equal to:

$$\begin{aligned} cov(F_i, F_j) &= cov(X + Y_i + Z, X + Y_j + Z) \\ &= cov(X, X) + cov(Y_i, Y_j) \\ &= \sigma_X^2 + \rho(u)\sigma_Y^2, \end{aligned}$$

where $\rho(u)$ is the correlation between locations i and j. Correlation between two locations is given by:

$$\rho_u = \frac{cov(F_i, F_j)}{\sigma_{F_i}\sigma_{F_j}} = \frac{\sigma_X^2 + \rho(u)\sigma_Y^2}{\sigma_X^2 + \sigma_Y^2 + \sigma_Z^2} \tag{A.3}$$

Using the equations described above, authors in [82] propose functions which describe the spatial correlation of neighbouring devices and wires.

Orshansky et.al in [63], propose that the gate length variation L^{intra} within the same chip consists of three components, a proximity component L^{prox}, a spatial component L^{spat} and a random component L^{rand}. The proximity component is dependent on the circuit and can be approximated experimentally. The spatial component encapsulates the spatial correlation between gates and can be described with a random variable. The random component can also be described as a random variable. That means that $L^{spat} \sim N(0, \sigma_{spat}^2)$ and $L^{rand} \sim N(0, \sigma^2)$. Thus, both spatial correlation and random variation can be expressed by a pdf with *zero* mean and σ variance. The authors then express analytically the gate length at each location on the layout using the aforementioned pdfs.

Friedberg et.al in [24], use experimental data to extract a mathematical model for the correlation of variation between two locations on the layout. Given the distance x between the two locations, a base correlation ρ_B and a base correlation length X_L, the correlation between the two locations is given by:

$$\rho = \begin{cases} 1 - \frac{x}{X_L}(1 - \rho_B), x \leq X_L \\ \rho_B, x \geq X_L \end{cases} \tag{A.4}$$

This means that if the distance between the two locations is more than X_L, then there is no significant correlation between them. In this case, the correlation is given by the base correlation ρ_B. Variable X_L denotes the within-die variation. Authors in [24] claim that a normal value for this variable is half the chip length. If the distance between the two locations is less than X_L, then the correlation is almost 1. Correlation then is decreasing linearly with the distance. Variable ρ_B is proportional to the factor of within-wafer variation by within-

die variation. Overall, this work assumes a linear model of variation which manifests itself geometrically in the chip. Compared to the previous works, Friedberg et.al also assume that gates in close proximity are affected by variation in similar ways.

Xiong et. al in [81] adopt a probability-based approach to compute the arrival time and the input capacitance for gates. According to the authors, the resistance of each wire equals $r \times l_i$, where r is the unit resistance and l_i is the wire length. Similarly, capacitance is given by $c \times l_i$, where c is the unit capacitance. The basic idea in [81] is to add buffers in the clock tree. Thus, the variables under consideration are L_t (buffer input capacitance) and T_t (buffer required arrival time). Through transformation of random variables, described in [31], the joint pdf of the two random variables under consideration is given by:

$$f_{L_t,T_t} = \int_{\Omega_{X,Y}(L_t,T_t)} f_{X,Y,L_t,T_t} dX dY, \tag{A.5}$$

where X, Y are the random variables describing the values that c and r can take. $\Omega_{X,Y}(L_t, T_t)$ is the definition field of X, Y with respect to L_t and T_t.

An analysis of the covariance among the paths' delays is described by Orshansky et. al in [62]. The delay of each path is described by a random variable which follows the normal distribution. Since the delays of paths are not independent to each other, mainly because they may share common gates, the authors define joint probability density functions for the delays among paths. The correlation among the random variables is also defined. Finally, the authors propose a formula for the covariance of two random paths' delays, which is given by:

$$cov\{D_i, D_j\} = \sum_{k_i=1}^{m_i} \sum_{k_j=1}^{m_j} cov\{d_g(i,k_i), d_g(j,k_j)\}, \tag{A.6}$$

if path i has m_i gates and path j has m_j gates. In order to compute $cov\{d_g(i,k_i), d_g(j,k_j)\}$ a Taylor expansion is required as described in [62]. In short, the covariance of delay between two gates depends on their relative location and the variation sources within-die.

A method which is applicable to the order of paths' delays under the presence of variations is described by Jess et. al in [36]. The authors describe a method which calculates the probability that the delay of a path is greater than the delay of another path. The assumption for the path delay is that it follows a normal distribution. Next, for a given pair of paths, whose delays have variance of σ_1 and σ_2 respectively and correlation ρ, their correlation matrix is given by:

$$\Phi = \begin{bmatrix} \sigma_1^2 & \sigma_1\sigma_2\rho \\ \sigma_1\sigma_2\rho & \sigma_2^2 \end{bmatrix}.$$

If s_1 and s_2 is the slack for the two paths respectively, A_i^T is the i-th line of A, z are the sources of variation and V is the correlation matrix $\eta \times \eta$ between the sources of variation, we can find σ_1, σ_2 and ρ from the equations:

$$\Phi = cov(A^T V) = \begin{bmatrix} (\frac{\partial s_1}{\partial z})^T \\ (\frac{\partial s_2}{\partial z})^T \end{bmatrix} [V] \begin{bmatrix} \frac{\partial s_1}{\partial z} & \frac{\partial s_2}{\partial z} \end{bmatrix} = \begin{bmatrix} A_1^T \\ A_2^T \end{bmatrix} [V][A_1 \ A_2]. \tag{A.7}$$

The probability that path 1 has delay greater than path 2 is: $b_1 = \int_{\eta=-\infty}^{\infty} p_1 p_4$, while the probability that path 2 has delay greater than path 1 is $b_2 = 1 - b_1 = \int_{\eta=-\infty}^{\infty} p_2 p_3$, where η is the slack between the two paths and p_1, p_2, p_3, p_4 are given by:

$$p_1 = p(s_1 = \eta) = \frac{1}{\sqrt{2\pi}} \frac{1}{\sigma_1} e^{-\frac{1}{2}\left(\frac{\eta-\mu_1}{\sigma_1}\right)^2} \tag{A.8}$$

$$p_2 = p(s_2 = \eta) = \frac{1}{\sqrt{2\pi}} \frac{1}{\sigma_2} e^{-\frac{1}{2}\left(\frac{\eta-\mu_2}{\sigma_2}\right)^2} \tag{A.9}$$

$$p_3 = p(s_1 \leq \eta | p_2) = 0.5 + \frac{1}{2} erf \left(\frac{\left(\frac{\eta-\mu_1}{\sqrt{2}\sigma_1}\right) - \left(\frac{\eta-\mu_2}{\sqrt{2}\sigma_2}\right)\rho}{\sqrt{1-\rho^2}} \right) \tag{A.10}$$

$$p_4 = p(s_2 \leq \eta | p_1) = 0.5 + \frac{1}{2} erf \left(\frac{\left(\frac{\eta-\mu_2}{\sqrt{2}\sigma_2}\right) - \left(\frac{\eta-\mu_1}{\sqrt{2}\sigma_1}\right)\rho}{\sqrt{1-\rho^2}} \right), \tag{A.11}$$

where p_1 is the probability that path 1 has slack of exactly η, p_2 is the probability that path 2 has slack of exactly η, p_3 is the conditional probability that path 1 has slack smaller than η provided that path 2 has slack of η and p_4 is the conditional probability that path 2 has slack smaller than η provided that path 1 has slack of η. Calculating the probabilities for all pairs of consecutive paths, as defined by static timing analysis without variation, one can calculate the probability that the relative order of paths will change after the application of variation.

Amin et. al in [5] analyze the way the delay of gate depends on the variation sources. They claim that the main sources of variation affecting gate delay are variations on gate length and threshold voltage. The analysis begins with the relation between the amount of variation with channel length:

$$le = le_{nom} + le_s + le_r, \tag{A.12}$$

where le is the actual channel length, le_{nom} is the nominal channel length, le_s is the fluctuation due to systematic variation and le_r is the fluctuation due to random variation. For threshold voltage it holds:

$$vt = vt_{nom} + vt_r, \tag{A.13}$$

where vt_r is the fluctuation due to random variation. The authors introduce the notion of correlation due to proximity, which for two gates i and j is given by:

$$\rho_{ij} = \rho(d_{ij}), \tag{A.14}$$

where d_{ij} is the distance of two gates i and j. The correlation ρ can take any value between 0 and 1 and is inversely proportional to the distance. All variables le_s, le_r and vt_r are random variables following the normal distribution. Thus, their variances are given by $\sigma^2_{le_s}$, $\sigma^2_{le_r}$ and $\sigma^2_{vt_r}$ respectively. The variance of a gate's delay is then given by:

$$\sigma^2_{delay} = \sigma^2_{delay,le_s} + \sigma^2_{delay,le_r} + \sigma^2_{delay,vt_r}, \tag{A.15}$$

where $\sigma^2_{delay,le_s}, \sigma^2_{delay,le_r}$ and σ^2_{delay,vt_r} is the correspondence of variations le_s, le_r and vt_r to delay variations. Based on [69], this correspondence is given by:

$$\begin{aligned} \sigma^2_{delay,le_s} &= f_1(\sigma_{le_s}, tt, C_L) \\ \sigma^2_{delay,le_r} &= f_2(\sigma_{le_r}, tt, C_L) \\ \sigma^2_{delay,vt_r} &= f_3(\sigma_{vt_r}, tt, C_L), \end{aligned} \tag{A.16}$$

where tt is the input slope of delay and C_L is the output capacitance. Based on the expression for a gate's delay and having defined the spatial correlation, the authors then proceed with a statistical static timing analysis framework.

The authors in [3] divide the physical layout into segments in order to approximate the correlation between gates. The layout is divided in the following way. First, the whole layout consists of a single segment. This segment is level-0 segment. Next, the layout is quadrisected into four equal segments. The new segments are level-1 segments. Next, each level-1 segment is further quadrisected to form four new level-2 segments. In total, there will be sixteen level-

2 segments. The quadrisection process continues until each segment is small enough. The threshold at which quadrisection stops depends on the mapping technology used for the gates. Figure A.2 shows an example with level-0, level-1 and level-2 segments.

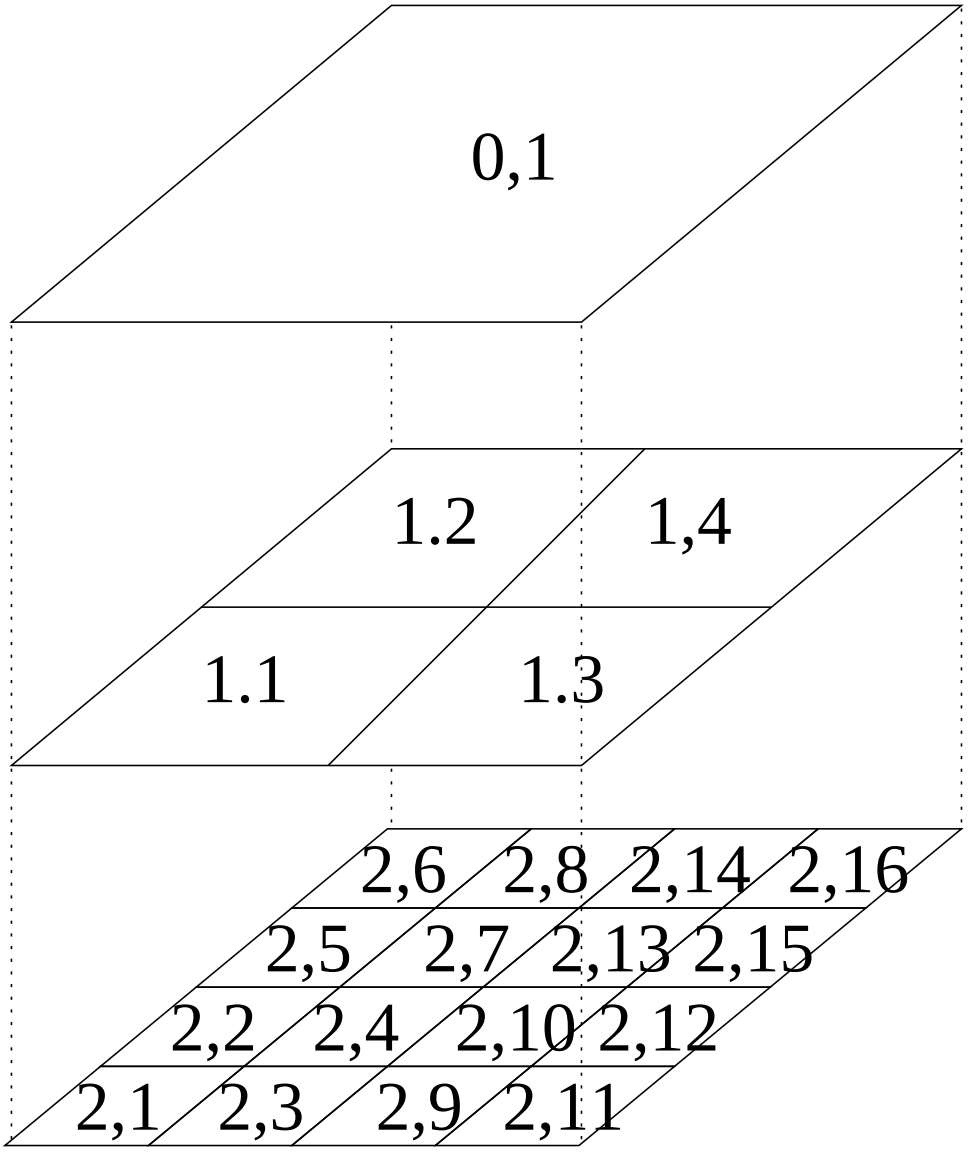


Figure A.2: Physical layout quadrisection

Each segment is labeled (l, r), where l denotes the level of the segment and r is a unique identifier for each segment belonging to the same level. For r, it holds that $r \in [1, 2^{l+1} - 1]$. Each gate in segment (l, r) experiences variation which is given by the sum $\Delta L(k, r)$, where

$0 \leq k \leq l$ and r is each segment from level-0 to level-l that the gate belongs to. For example, a gate belonging to segment (2,9), experiences variation which is given by:

$$\Delta_1 = \Delta(2,9) + \Delta(1,3) + \Delta(0.1). \tag{A.17}$$

This computation is used for the calculation of correlation between a gate pair. If, for example, a gate belongs to segment (2,10) and another gate belongs to gate (2,5), then it holds:

$$\Delta_2 = \Delta(2,10) + \Delta(1,3) + \Delta(0,1) \tag{A.18}$$

$$\Delta_3 = \Delta(2,5) + \Delta(1,2) + \Delta(0,1) \tag{A.19}$$

Thus, gates belonging to segments (2,10) and (2,9) are strongly correlated, as they experience similar variation. This is shown by the right-hand part of equations ($\Delta(1,3)$ and $\Delta(0,1)$). On the other hand, gates belonging to segments (2,9) and (2,5) are not so strongly correlated, since their correlation functions share only one common term ($\Delta(0,1)$).

Appendix B

Placement Approaches

B.1 Simulated Annealing

Simulated annealing-based optimization stems from the idea of metal annealing [45]. During annealing, a metal is melted and then it is cooled progressively until it reaches a structure which is close to a perfect crystal. The quality of the end result depends on the cooling progress which affects the way the metal's molecules move towards their final locations. Algorithms based on this idea are called simulated annealing algorithms. One of the key characteristic of such algorithms is that they allow "bad" moves which help them escape local minima. Typical simulated annealing algorithms use a variable called "temperature" which controls the cooling process.

In simulated annealing placement algorithms, metal's molecules correspond to standard cells of the circuit. These algorithms start from an initial placement which is thought to have been "melted". Then it is progressively cooled towards the final placement. During cooling, standard cells are allowed to move on the layout area. The freedom cells have with respect to their movement depends on the temperature. The lower the temperature, the harder it is for a cell to move, especially if this move does not improve the overall quality of placement.

The inner loop of a placement algorithm using simulated annealing can be divided into two phases. In the first phase, some cells are selected to move. The selection can be made according to a heuristic which depends on the actual implementation of the algorithm. Typically, the "worst" cells will be selected, which in the case of simple wirelenegth minimization, will be the cells causing the longest wires. The selected cells are moved to new locations which can

be random or stochastically better than the current location of cells. The second phase of the algorithm's loop consists of the evaluation of the new placement. The cells that have moved can have either optimized the overall quality of placement or have deteriorated it. In the first case the move is considered good and it is accepted. In the second case, the move may be rejected and the cells returned to their previous location. The rejection decision depends on the temperature and the amount of deterioration. If the temperature is high enough, then even very bad moves may be accepted to help the algorithm escape local minima. After the evaluation step the new placement forms the current solution which the algorithm will try to optimize in the same way in the next iteration. The most successful implementation of simulated annealing algorithm is the Timberwolf placer [74].

The challenges for a simulated annealing algorithm is to find the best initial temperature and the best cooling process. These depend on the structure of the circuit in total analogy with physical annealing, where temperature and cooling depends on the metal itself. Moreover, the granularity at which cells will be selected depends on the circuit size. Finally, simulated annealing algorithms are known to be able to find global minima but at a high runtime cost.

B.2 Genetic

Genetic algorithms form another category of stochastic algorithms. Genetic algorithms like the ones proposed in [17, 71] are based on the idea of simulating the evolution process which can be observed on living species. In environments like this, the "law of the fittest" is applied which tends to favour the characteristics of the fittest organisms over the centuries. This leads to the creation of new, even fitter organisms. In analogy to placement, the evolution process is simulated in order for the characteristics of the best placements to be preserved and slowly evolve to the best placement.

In a typical genetic algorithm for placement, the "individual" is defined as one full placement. For each individual, there is a metric called "health" which defines the quality of placement. Health can be measured in a way similar to any other placement algorithm and can be *e.g* the total wirelength required. At all times, a number of full placements (or individuals) is maintained which form the total "population". A "mating" is the process of combining two individuals in order to create a new individual. A "mutation" is a change randomly made to the characteristics of one individual.

A genetic algorithm for placement starts from an initial population which corresponds to a

number of different initial placements. New placements are created by combining the existing placements and are added to the population. The less fit placements, or in other words the placements which rank lowest with respect to the objective function, are discarded to keep the size of population constant. The new population forms the new generation which will be optimized in the next iteration. In order to escape local minima, a number of mutated individuals is created and inserted into the population. The optimization process ends after the objective function has been met, or after a predefined number of iterations.

Challenges for genetic placement algorithms are the size of population, the number of mutations and the way individuals are combined to form new placements. Especially for the mating process, a sophisticated stochastic heuristic is required to identify the pairs that are most likely to produce better placements. The iterative nature of genetic algorithms also incurs long runtime costs.

B.3 Min-Cut

The first two min-cut algorithms, Kernighan-Lin [43] and Fiduccia-Mattheyses [23] were based on recursive bisection. The cut cost was defined as the number of nets spanning between the two segments which were created after bisection. The first, tries to minimize the cut cost by swapping cells between the two segments and the second, uses simple moves from one segment to the other. Typically, bisection algorithms work recursively until the segments are small enough so that the problem can be solved through exhaustive search of the solution space.

Today, most placement algorithms employ a bisection phase to enable them to handle circuits with a very large number of instances. Most notably, placers like Dragon [75, 76], Capo [66] and Feng-Sui [44] base some stage of their optimization process on recursive bisection. Other placers like PROUD [77], Gordian [46], BonnPlace [8] and hATP [57] employ quadratic partitioning in their main optimization loop. The success of the aforementioned placers stems from the fact the bisection processes, enhanced with the idea of multilevel bisection [40], boost the scalability of these algorithms.

Algorithms which use a bisection phase must tackle a number of problems stemming from bisection itself. The quality of a bisection algorithm depends on the number of bisections allowed and the relative size of segments. The direction of cuts is also very important and depends on the structure of the circuit [4]. The cut direction affects the aspect ratio of segments which may play a critical role in the placement process. Another known problem is the efficient application

of terminal propagation [37]. The idea of terminal propagation is to control the distance between gates belonging to different segments. Due to recursive bisection, these gates may become too far apart, negatively impacting the cost function. Another problem with bisection approaches is that they are not correct-by-construction, *i.e.* they do not place the cells on legal locations. This forces these algorithms to employ a detailed legalization placement procedure which, in the general case, is not a straightforward problem to solve under the presence of tight constraints. Finally, bisection algorithms ignore any white (or free) space available on the layout, so a step which optimizes the placement by efficiently utilizing the white space may also be necessary.

B.4 Analytical

Non-linear placers employ a non-linear cost function like one of log-sum-exp type [58]. Due to the high computational load required by non-linear optimization, approaches of this type also employ a partitioning approach. Successful representatives of this category are APlace [38], mPL [22] and NTUPlace [21].

Quadratic placers on the other hand, employ a quadratic cost function. As mentioned earlier, successful quadratic placers which use partitioning are PROUD [77], Gordian [46], BonnPlace [8] and hATP [57]. A type of quadratic placers are force-directed placers. Force-directed placers model the interconnect as forces which try to place the cells as close as possible. They also employ spreading forces which prevent the placement from being over-congested. Force-directed placers start from an initial placement and calculate the forces exerted on each standard cell. These forces stem from the cells each standard cell is connected to and try to force the standard cell to move towards its connections. At each step, each cell will try to move towards a location which is represented by the vector sum of all the forces placed on the cell. Optimization continues as long as there are cells which have not reached their equilibrium location, or when any move results in a worse placement with respect to the cost function. Force-directed placers are known to produce excellent results, but care must be taken for runtime, as the computational load on the calculation of the quadratic cost function may be high. Successful placers of this kind are FastPlace [78], mFAR [32] and KraftWerk [73].

Appendix C

Statistical Distributions

C.1 Normal Distribution

Normal distribution is a probability distribution which describes phenomena that exhibit random behaviour around a *mean* value. Its probability density function is:

$$f_X = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{C.1}$$

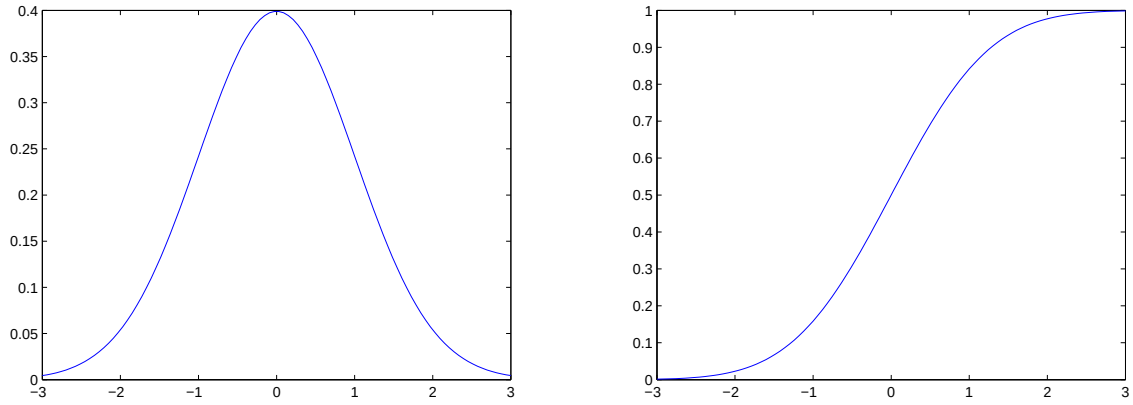
where μ is the mean and σ is the standard deviation. These are the two values necessary for describing any normal distribution.

It is often referred to as the "bell" distribution due to the characteristic shape in a probability-value plot. The normal distribution is also known as the "Gaussian" distribution. A typical plot of the probability density function (pdf) for a normal distribution is shown in Figure C.1a.

Its cumulative density function (cdf) is given by:

$$\frac{1}{2}\left[1 + erf\left(\frac{x-\mu}{\sqrt{2\sigma^2}}\right)\right] \tag{C.2}$$

A typical graph of a normal distribution's cdf is shown in Figure C.1b.



(a) Normal Distribution pdf

(b) Normal Distribution cdf

Figure C.1: Normal Distribution pdf and cdf

C.2 Log-Normal Distribution

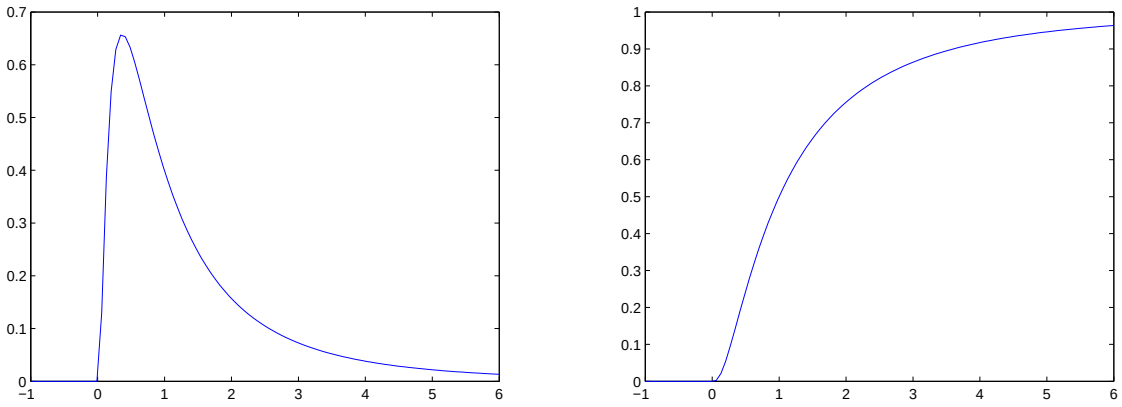
A Log-normal is associated with a random variable, whose logarithm is normally distributed. Like the normal distribution, a log-normal distribution can be described with its mean μ and its standard deviation σ. Its probability density function is given by:

$$f_X = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{(lnx-\mu)^2}{2\sigma^2}} \tag{C.3}$$

and its cumulative density function is given by:

$$F_X = \frac{1}{2} erf\left[-\frac{lnx-\mu}{\sigma\sqrt{2}}\right] \tag{C.4}$$

Typical plots of a log-normal distribution's pdf and cdf are shown in Figures C.2a and C.2b.



(a) Log-Normal Distribution pdf

(b) Log-Normal Distribution cdf

Figure C.2: Log-Normal Distribution pdf and cdf

References

[1] S.N. Adya and I.L. Markov. Consistent placement of macro-blocks using floorplanning and standard-cell placement, 2002.

[2] S.N. Adya and I.L. Markov. Combinatorial techniques for mixed-size placemen, 2004.

[3] A. Agarwal, D. Blaauw, and V. Zolotov. Statistical Timing Analysis for Intra-die Process Variations with Spatial Correlations. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 900–907, San Jose, CA., November 2003.

[4] A. R. Agnihotri, S. Ono, and P. H. Madden. Feng shui 5.0 implementation details. In *ISPD*, 2005.

[5] C. S. Amin, N. Menezes, K. Killpack, F. Dartu, U. Choudhury, N. Hakim, and Y. I. Ismail. Statistical static timing analysis: How simple can we get? In *DAC*, 2005.

[6] K. Van Berkel, R. Burgess, J. Kessels, M. Roncken, F. Schalij, and A. Peeters. Asynchronous circuits for low power: a dcc error corrector. *Design and Test of Computers, IEEE*, 11, 1994.

[7] D.A. Bertke. A simulation for dynamic timing analysis. In *Systems Engineering, 1990., IEEE International Conference on*, 1990.

[8] U. Brenner and M. Struzyna. Faster and better global placement by a new transportation algorithm. In *Proceedings ACM/IEEE Design Automation Conference*, pages 591–596, 2005.

[9] Y. Cao and L. T. Clark. Mapping statistical process variations toward circuit performance variability: An analytical modeling approach. In *DAC*, 2005.

[10] Capo Placer. http://vlsicad.eecs.umich.edu/BK/PDtools/Capo.

[11] P. Cardieri and T.S. Rappaport. Statistics of the sum of lognormal variables in wireless communications. In *Vehicular Technology Conference Proceedings*, pages 1823–1827, 2000.

[12] S. Chakraborty, K.Y. Yun, and D.L. Dill. Timing analysis of asynchronous systems using time separation of events. *IEEE Trans. Comput. Aided Design*, 18:1061–1076, 1999.

[13] H. Chang and S. Sapatnekar. Statistical timing analysis considering spatial correlations using a simple pert-like traversal. In *Proceedings of Int. Conf. on Computer Aided Design*, pages 621–625, 2003.

[14] Yi.-C. Chang, Y.-W. Chang, G.-M. Wu, and S.-W. Wu. B*-trees: a new representation for non-slicing floorplans. In *Proceedings of Design Automation Conference*, pages 458–463, 2000.

[15] C. E. Clark. The greatest of a finite set of random variables. *Operations Research*, pages 145–162, 1961.

[16] C.E. Clark. The greatest of a finite set of random variables. *Operations Research*, pages 145–162, 1961.

[17] J. P. Cohoon and W. D. Paris. Genetic placement. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 422–425, 1986.

[18] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and System*, E80-D:315–325, 1997.

[19] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou. A concurrent model for desynchronization. In *iwls*, pages 294–301, 2003.

[20] ECSM Format. http://www.cadence.com/Alliances/languages/Pages/ecsm.aspx.

[21] T.-C. Chen et al. Ntuplace3: An analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints. *IEEE Transactions on Computer-Aided Design*, 27(7):1228–1240, 2008.

[22] T. F. Chan et al. mpl6: Enhanced multilevel mixed-size placement. In *ISPD*, pages 212–214, 2006.

[23] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, 1982.

[24] P. Friedberg, Y. Cao, J. Cain, J. Rabaey, and C. Spanos. Modeling within-die spatial correlation effects for process-design co-optimization. In *ISQED*, 2005.

[25] GLPK solver. http://www.gnu.org/software/glpk/glpk.html.

[26] GNU Octave. http://www.gnu.org/software/octave/.

[27] GNU GPROF.
http://gcc.gnu.org/.

[28] R. Gonzalez, B.M. Gordon, and M. A. Horowitz. Supply and threshold voltage scaling for low power cmos. *IEEE JOURNAL OF SOLID-STATE CIRCUITS*, 32(8), 1997.

[29] K.R. Heloue and F.N. Najm. Early analysis and budgeting of margins and corners using two-sided analytical yield models. *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS*, 27(10), 2008.

[30] A. Hemani. Charting the eda roadmap. *IEEE Circuits and Devices Magazine*, 2004.

[31] R. Hogg and A. Craig. *Introduction to mathematical statistics*. Macmillan, 1995.

[32] B. Hu and M.M. Sadowska. Multilevel fixed-point addition-based vlsi placement. *IEEE Transactions on Computer-Aided Design*, 24(8):1188–1203, 2005.

[33] H. Hulgaard, S.M. Burns, T. Amon, and G. Borriello. n algorithm for exact bounds on time separation of events in concurrent systems. *IEEE Transactions on Computer*, 44:1306–1317, 1995.

[34] International technology roadmap for semiconductors. http://www.itrs.net/.

[35] IWLS 2005 Benchmarks. http://www.iwls.org/iwls2005/benchmarks.html.

[36] J. A. G. Jess, K. Kalafala, S. R. Naidu, R. H. J. M. Otten, and C. Visweswariah. Statistical timing for parametric yield prediction of digital integrated circuits. In *DAC*, 2003.

[37] A. B. Kahng and S. Reda. Placement feedback: A concept and method for better mincut placements. In *DAC*, 2004.

[38] A.B. Kahng and Q. Wang. Implementation and extensibility of an analytic placer. *IEEE Transactions on Computer-Aided Design*, 24(5):734–747, 2005.

[39] K. Kang, B.C. Paul, and K. Roy. Statistical Timing Analysis Using Levelized Covariance Propagation Considering Systematic and Random Variations of Process Parameters. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 11(4):848–879, 2006.

[40] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. In *Proceedings of 36th ACM/IEEE conference on Design automation*, 1999.

[41] An eda tool for the timing analysis, optimization and timing validation of asynchronous circuits. http://elocus.lib.uoc.gr/dlib/0/5/7/metadata-dlib-7d94bc32d455a2550cbc35aa61fc8acc_1276761364.tkl.

[42] J.F. Kenney and E.S. Keeping. *Mathematics of Statistics*. N.Y. : Van Nostrand, 1951.

[43] B Kernighana and S Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 29, 1970.

[44] A. Khatkhateand, C. Liand, A. R. Agnihotriand, M. C. Yildizand, and S. Ono. Recursive bisection based mixed block placement. In *ISPD*, 2004.

[45] S. Kirkpatrick, C. Gelatt Jr, and M. Vecchi. Optimization by simulated annealing. In *Science*, volume 220, pages 671–680, 1983.

[46] J.M. Kleinhans, G. Sigl, F.M. Johannes, and K.J. Antreich. Gordian: Vlsi placement by quadratic programming and slicing optimization. *IEEE Transactions on Computer-Aided Design*, 10(3):356–365, 1991.

[47] E. Kounalakis and Ch.P. Sotiriou. Cplace: A constructive placer for synchronous and asynchronous circuits. In *Asynchronous Circuits and Systems (ASYNC), 2011 17th IEEE International Symposium on*, pages 22–32, 2011.

[48] E. Kounalakis and Ch.P. Sotiriou. Scplace: A statistical slack-assignment based constructive placer. In *Quality Electronic Design (ISQED), 2011 12th International Symposium on*, pages 136–144, 2011.

[49] E. Kounalakis, Ch.P. Sotiriou, and Vassilis Zebilis. Statistical timing-based post-placement leakage recovery. In *Proceedings of Annual Symposium on VLSI (ISVLSI)*, 2011.

[50] LEF/DEF Formats. http://www.si2.org/openeda.si2.org/project/showfiles.php?group_id=6#p44.

[51] E. T. Leighton. Complexity issues in vlsi. MIT Press, 1983.

[52] Liberty Format. http://www.opensourceliberty.org.

[53] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani. Rectangle-packing-based module placement. In *Proceedings of ICCAD*, pages 472–479, 1995.

[54] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[55] K.G. Murty. *Linear Programmin*. John Wiley & Sons, New York, 1983.

[56] R. Nair, C.L. Berman, P.S. Hauge, and E.J. Yoffa. Generation of Performance Constraints for Layout. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(8):860–874, 1989.

[57] G.-J. Nam, S. Reda, C.J. Alpert, P.G. Villarrubia, and A.B. Kahng. A fast hierarchical quadratic placement algorithm. *IEEE Transactions on Computer-Aided Design*, 25(4):678–691, 2006.

[58] W. Naylor, R. Donelly, and L. Sha. Non-linear optimization system and method for wire length and delay optimization for an automatic electric circuit placer, 2001.

[59] L.S. Nielsen and J. Sparso. Designing asynchronous circuits for low power: an ifir filter bank for a digital hearing aid. *Proceedings of the IEEE*, 87:268–281, 2002.

[60] OpenCores.org.
http://www.opencores.org.

[61] M. Orshansky and A. Bandyopadhay. Fast statistical timing analysis handling arbitrary delay correlations. In *DAC*, 2004.

[62] M. Orshansky and K. Keutzer. A general probabilistic framework for worst case timing analysis. In *DAC*, 2002.

[63] M. Orshansky, L. Milor, P. Chen, K. Keutzer, and C. Hu. Impact of systematic spatial intra-chip gate length variability on performance of high-speed digital circuits. In *DAC*, 2000.

[64] A. Papoulis. *Probability, Random Variables and Stochastic Processes*. McGraw-Hill, 1991.

[65] R. Rao, A. Srivastava, D. Blaauw, and D. Sylvester. Statistical analysis of subthreshold leakage current for vlsi circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(2), 2004.

[66] J. A. Roy, D. A. Papa, S. N. Adya, H. H. Chan, A. N. Ng, J. F. Lu, and I. L. Markov. Capo: Robust and scalable open-source min-cut floorplacer. In *Proceedings ACM/SIGDA ISPD*, pages 224–226, 2005.

[67] J.A. Roy, S.N. Adya, D.A. Papa, and I.L. Markov. Min-cut floorplacement. *IEEE Transactions on Computer-Aided Design*, 25(7):1313–1326, 2006.

[68] J. Rubenstein, P. Peneld, and M. A. Horowitz. Signal delay in rc tree networks. *IEEE Transactions on Computer-Aided Design*, CAD-2:202–211, 1983.

[69] T. Sakurai and A. Newton. Alpha-power law mosfet model and its application to cmos inverter delay and other formulas. *IEEE JSSC*, 25(2):584–594, 1990.

[70] M. Sarrafzadeh, M. Wang, and X. Yang. *Modern Placement Techniques*. Kluwer Academic Puvlishers, 2002.

[71] K. Shahookara and P. Mazumder. Gasp - a genetic algorithm for standard cell placement. In *Proceedings of the European Design Automation Conference*, pages 660–664, 1990.

[72] Debjit Sinha, Hai Zhou, and Narendra V. Shenoy. Advances in computation of the maximum of a set of random variables. In *ISQED '06: Proceedings of the 7th International Symposium on Quality Electronic Design*, pages 306–311, 2006.

[73] P. Spindler, U. Schlichtmann, and F.M. Johannes. Kraftwerk2 - a fast force-directed quadratic placement approach using an accurate net model. *IEEE Transactions on Computer-Aided Design*, 27(8):1398–1411, 2008.

[74] W.-J. Sun and C. Sechen. Efficient and effective placement for very large circuits. In *In Proceedings IEEE/ACM ICCAD*, pages 170–177, 1993.

[75] T. Taghavi, X. Yang, and B.-K. Choi. Dragon 2005: Large-scale mixed-size placement tool. In *Proceedings ACM/SIGDA ISPD*, pages 245–247, 2005.

[76] T. Taghaviand, X. Yang, and B-K. Choi. Blockage-aware congestion-controlling mixed-size placer. In *ISPD*, 2006.

[77] R.-S. Tsay, E.S. Kuh, and C.-P. Hsu. Proud: A sea-of-gates placement algorithm. *IEEE Design and Test of Computers*, 5(6):44–56, 1988.

[78] N. Viswanathan, M. Pan, and C. Chu. Fastplace 3.0: A fast multilevel quadratic placement algorithm with placement congestion control. In *ASPDAC*, pages 135–140, 2007.

[79] C. Visweswariah, K. Ravindran, K. Kalafala, S. G. Walker, and S. Narayan. First-order incremental block-based statistical timing analysis. In *DAC*, 2004.

[80] C. Visweswariah, K. Ravindran, K. Kalafala, S. G. Walker, and S. Narayan. First-order incremental block-based statistical timing analysis. In *Proceedings of the IEEE Design Automation Conference*, San Diego, CA., June 2004.

[81] J. Xiong, K. Tam, and L. He. Buffer insertion considering process variation. In *DATE*, 2005.

[82] J. Xiong, V. Zolotov, and L. He. Robust extraction of spatial correlation. In *ISPD*, 2006.

[83] T.-Y. Yen, A. Ishii, A. Casavant, and W. Wolf. Efficient algorithms for interface timing verificat. *Formal Methods Syst. Design*, 12:241–265, 1998.

[84] L. Zhang, W. Chen, Y. Hu, J. A. Hubner, and C. C.-P Chen. Statistical timing analysis with extended pseudo-canonical timing model. In *Proceedings of Design Automation and Test in Europe*, 2005.