Lecture Notes in Electrical Engineering

Volume 166

For further volumes:
http://www.springer.com/series/7818

David A. Papa · Igor L. Markov

Multi-Objective Optimization in Physical Synthesis of Integrated Circuits

Springer

David A. Papa
Broadway Technology
5000 Plaza on the Lake
Austin, TX 78746
USA

Igor L. Markov
University of Michigan
Ann Arbor, MI 48109
USA

ISSN 1876-1100 ISSN 1876-1119 (electronic)
ISBN 978-1-4614-1355-4 ISBN 978-1-4614-1356-1 (eBook)
DOI 10.1007/978-1-4614-1356-1
Springer New York Heidelberg Dordrecht London

Library of Congress Control Number: 2012937727

© Springer Science+Business Media New York 2013
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Acknowledgments

This book would not have been possible without the immeasurable self-sacrifice of my perfect wife, Amy. She has worked day and night by my side for years to make our home and family prosperous. I love you very much. Our two beautiful sons George and Victor have brought me indescribable joy and gave me hope for the future when it seemed all was lost. I love you two in ways I never thought possible. I am eternally grateful to her for the faith she has placed in me. I will do everything I can to reward her investment.

I am also deeply indebted to her parents Ren Fang Zhang and Yue Xia Gong who have come from their home in China to live with us and help raise our babies. Without them, I don't know how it would be possible for me to balance graduate school, a full-time job, and a new family. I will be sorry when they return home.

My advisor, Professor Igor Markov, has also poured an incredible amount of work into training me to be capable of writing this. He has defended me when it was not convenient, supported me when it seemed hopeless, and never gave up on me until the task was complete. I am grateful for all his efforts as well as all the opportunities and second chances he has given me. I truly hope it has been as worth it for him as it has been for me.

I also want to thank all the people at IBM Austin Research Lab, especially my manager Chuck Alpert and my mentor Gi-Joon Nam. Chuck's approach to industrial research is truly unique and I feel very fortunate to have worked with him during graduate school. Gi-Joon has seen my value from the very start and stuck his neck out for me when it mattered most. I hope he feels proud of his judgment. I have also made many friendships here, and this research group is an amazing place to work. Zhuo Li, Jarrod Roy, Cliff Sze, Natarajan Viswanathan, Mehmet Yildiz and Nancy Zhou are brilliant people who have all had their impact on this book. Working with these people on a topic I love has truly made it a pleasure to come to work each day. I also want to thank Anne Gattiker for accompanying me on so many occasions while I burned the midnight oil to finish this book. John Keane and Vipin Sachdeva have also been close friends who enriched my life during our time together at IBM.

I also have to thank all my friends from igroup who have stayed close despite long distances and made life bearable. Jarrod Roy has been an especially close friend, and he is not so far away today. Smita Krishnaswamy has been my friend for even longer, and still brainstorms with me daily. Aaron may have left igroup a long time ago, but he is still making me laugh every day. George Viamontes has moved on to greener pastures, but he is always willing to lend good advice when its needed. Esha Krishnaswamy was never a member of igroup, but she has been there for moral support when I wanted it. Jin Hu has been a good friend and helped me with this book and been my boots on the ground in Michigan. And while I haven't been in Michigan for the current group of graduate students, Myung-Chul Kim, and Dong-Jin Lee, and Hector Garcia have all been good friends during my visits.

I also want to thank the other friends I have managed to keep over the years despite everyone being spread all over the country. Brandon Hanson has been by my side through a lot of interesting adventures, and would be there again if I needed him. Jason Feyers has been a really close friend and showed me a lot about life. Sid Bottoms and his family have always been good to me, and he is a really great guy. Max Mass taught me a lot of things and I have to thank him for that. Ryan Park has always been fun and I wish I stayed in better touch with him now.

Last but not least, I want to thank my parents George and Maureen who gave everything they had to support me, as well as my sister and brothers Crystal, Mitchell and Evan, who look up to me and give me motivation to carry on. I love you all very much. I could not have been successful without the foundation they built. I expect my family's future to be bright thanks to their support and sacrifices.

Contents

Part I Introduction and Prior Art

1 Timing Closure for Multi-Million-Gate Integrated Circuits 3
1.1 Challenges in Physical Synthesis 3
1.2 Our Contributions 5
1.3 Organization of the Book 8
References 8

2 State of the Art in Physical Synthesis 11
2.1 Progression of a Modern Physical-Synthesis Flow 11
2.2 The Controller/Transformation Approach 13
2.3 Circuit Delay Estimation 14
2.4 Current Trends in Physical Synthesis 16
References 17

Part II Local Physical Synthesis and Necessary Analysis Techniques

3 Buffer Insertion During Timing-Driven Placement 21
3.1 Introduction 21
3.2 Background 24
3.3 The RUMBLE Timing Model 25
3.4 Timing-Driven Placement with Buffering 29
3.5 The RUMBLE Algorithm 34
3.6 Empirical Validation 38
3.7 Conclusions 45
References 45

4 Bounded Transactional Timing Analysis 47
4.1 Introduction 47
4.2 Background 49
4.3 Transactional Timing Analysis 55
4.4 Empirical Validation 60
4.5 Conclusions 61
References 62

5 Gate Sizing During Timing-Driven Placement 65
5.1 Introduction 65
5.2 Background 68
5.3 Problem Formulation 68
5.4 Our Simultaneous Placement and Gate-Sizing Algorithm 72
5.5 Empirical Validation 77
5.6 Conclusions 79
References 79

Part III Broadening the Scope of Circuit Transformations

6 Physically-Driven Logic Restructuring 83
6.1 Introduction 83
6.2 Background and Preliminaries 86
6.3 Fast Timing-Driven Gate Cloning 88
6.4 Empirical Validation 97
6.5 Extensions 100
6.6 Conclusions 102
References 103

7 Logic Restructuring as an Aid to Physical Retiming 105
7.1 Introduction 105
7.2 Background, Notation and Objectives 107
7.3 Joint Optimization for Physical Synthesis 110
7.4 Empirical Validation 118
7.5 Extensions 120
7.6 Conclusions 121
References 122

8 Broadening the Scope of Optimization Using Partitioning 123
8.1 Introduction 123
8.2 Background 124
8.3 Forming Subcircuits Using Top-Down Netlist Partitioning 125
8.4 Trade-Offs in Window Selection 127
8.5 Empirical Validation 129

8.6 Conclusions 131
References 132

9 Co-Optimization of Latches and Clock Networks 133
9.1 Introduction 133
9.2 Background 135
9.3 Disruptive Changes in Physical Synthesis 137
9.4 A Graceful Physical-Synthesis Flow 139
9.5 Empirical Validation 143
9.6 Conclusions 147
References 148

10 Conclusions 149
10.1 Summary of Results 149
10.2 Opportunities for Further Optimizations 153
References 155

Part I
Introduction and Prior Art

Chapter 1
Timing Closure for Multi-Million-Gate Integrated Circuits

Sophisticated integrated circuits (ICs) can be classified as processors (CPUs), application-specific integrated circuits (ASICs) or systems-on-a-chip (SoCs), which embed CPUs and intellectual property blocks into ASICs. Mass-produced on silicon chips, these circuits fuel consumer and industrial electronics, maintain national and international computer networks, coordinate transportation and power grids, and ensure the competitiveness of military systems. The design of new integrated circuits requires sophisticated optimization algorithms, software and methodologies—collectively called Electronic Design Automation (EDA)—which leverage synergies between Computer Science, Computer Engineering and Electrical Engineering. From the algorithmic perspective, a number of NP-hard problems need to be solved quickly in practice, while their instances grow year after year with Moore's law. From the software perspective, multiple optimizations must operate on sophisticated design databases and coordinate to ensure consistent results over a large variety of chip designs. Electrical-engineering aspects of EDA emphasize physical characteristics of integrated circuits, such as speed-of-light limitations observed in large, high-speed chips manufactured at sub-65 nm technology nodes.

1.1 Challenges in Physical Synthesis

State-of-the-art automated IC design flows begin at a planning stage with rough estimates of chip performance and cost. During this stage, a block-level layout or *floorplan* of the chip is determined. Next, designers describe the function of the chip using a hardware description language (HDL), such as Verilog or VHDL. A logic synthesis tool is run on the HDL code to create a mapped netlist that implements the design in the target standard-cell library. Timing analysis can then calculate crude, optimistic estimates of chip performance, and the HDL code can be improved until it passes this sanity check.

D. A. Papa and I. L. Markov, *Multi-Objective Optimization in Physical Synthesis of Integrated Circuits*, Lecture Notes in Electrical Engineering 166,
DOI: 10.1007/978-1-4614-1356-1_1, © Springer Science+Business Media New York 2013

The physical synthesis stage begins after logic synthesis produces a gate-level netlist that meets agreed performance targets under optimistic timing conditions. A physical synthesis tool reads the netlist, creates an overlap-free placement of gates, and proceeds to optimize circuit performance. During physical synthesis, the availability of gate locations enables more accurate interconnect-delay modeling in timing analysis. Common physical synthesis operations include inserting or removing buffers and inverters, resynthesizing small windows, increasing and decreasing gate sizes, as well as relocating gates. When a design meets its performance constraints, it is said to *have closed on timing*. When physical synthesis is unable to achieve timing closure, designers must study the tool's logs and the optimized circuit then manually generate additional constraints to guide the optimization process. More substantial timing-closure difficulties can cause an expensive return to the logic synthesis stage, necessitate floorplanning changes or even require changes in the HDL code.

Designs that have passed timing checks during physical synthesis, transition into the routing stage, where more accurate timing models are available and new timing-closure problems may arise. Failure to route or meet timing constraints at this stage can again cause a return to earlier stages and further iterations. Finally, post-routing optimizations address any timing-closure issues that remain after routing, such as changing wire layers to reduce variability, moving detailed routes to reduce crosstalk, or adding redundant vias to improve manufacturing yield.

Challenges Aggressive scaling of transistor dimensions according to Moore's Law has historically driven performance improvements of CMOS-based integrated circuits (ICs). This trend has been so successful that now the greater part of critical path delay is no longer in the transistors that compose logic gates—delay through signal nets and repeaters now dominates [1]. As a result, logic synthesis can no longer estimate design performance effectively without physical information. A relatively recent solution, physical synthesis optimization algorithms employ a complex, multi-phase process that combines netlist optimization, placement, routing and timing analysis [2–4]. Physical synthesis optimization algorithms are primarily designed to achieve timing closure, but there are other important objectives such as reducing wirelength, area and power consumption while maintaining routability.

Another consequence of technology scaling trends gives IC designers more transistors at their disposal, which leads to increased design size and complexity. Today's ICs have tens of millions of gates and each design has its own performance requirements, which include reducing power consumption, satisfying area bounds and increasing manufacturing yield. A physical synthesis tool must accommodate these requirements as well as ensure that basic physical constraints are met, such as producing a legal, routable placement. As a result, throughout the physical synthesis flow *multiple objectives* are always present and must be optimized simultaneously.

Several prior publications formulate non-linear, multi-objective optimization problems and solve them with some success [5], but these algorithms typically exhibit super-linear runtime complexity and do not scale well enough to optimize an entire modern VLSI design at once. Other approaches focus on a handful of gates, and apply more time-consuming algorithms to relocate several gates at once,

increase drive strength, or insert buffers to improve performance [2, 6, 7]. However, these approaches are limited in scope and only near-linear-time algorithms such as *wirelength-driven placement* can be applied at a truly global scope. For example, the scope of *timing-driven gate relocation* is typically limited to finding new positions for a handful of gates so as to improve the delay of incident paths.[1] Few techniques are available between the global and local scopes, but resynthesis is a notable exception. While logic synthesis techniques are applied to more than a few gates at once, the delay estimations considered at that scale do not typically utilize all of the physical information available and are therefore less accurate [8]. Consequently, in state-of-the-art physical synthesis tools there is a large gap between the scope of accurate, local transformations and coarse, global transformations.

More recently, a trend toward integration of such *point optimizations* as repowering, buffering, and timing-driven detailed placement has gained strength. Increasing the scope of such *compound transformations* to close the aforementioned gap while maintaining acceptable runtime and accuracy remains a challenging research problem. It is unclear *a priori* if established techniques based on static timing analysis and single-objective optimizations remain sufficient in the context of physical synthesis for sub-45 nm ICs. To this end, Chap. 9 reports successful experiments with 32 nm and 22 nm designs.

1.2 Our Contributions

In this book, we make several contributions that advance the capabilities and strength of modern software tools for physical synthesis, with the ultimate goal to improve the quality of leading-edge semiconductor products. In so doing, we broaden the scope of physical synthesis optimization in two distinct ways: (i) we integrate related transformations to break circular dependencies and find optimization synergies and (ii) we increase the number of objects that can be jointly optimized to escape local minima.

Integrated transformations in this book are developed by first considering a successful optimization and identifying obstacles to its further application. We then derive methods to overcome those obstacles that call for integration. Integration is achieved through mapping multiple operations to rigorous mathematical optimization problems and solving them simultaneously. We achieve scalability in our techniques by leveraging analytical delay models and restricting consideration to carefully selected regions of the chip. In particular, we make extensive use of a linear interconnect-delay model that accounts for the impact of subsequent repeated insertion. We also demonstrate that bottom-up clustering and top-down partitioning can

[1] This is in contrast to *timing-driven placement*, which in previous literature usually refers to the application of net weights during placement that are based on timing information. Here we are referring to the detailed placement of a small number of gates while interacting incrementally with a timing analysis engine.

be used to select small regions of large circuits on which our optimizations have a large impact.

Simultaneous placement and buffering

At advanced technology nodes multiple cycles are required for signals to cross the chip, making latch placement critical to timing closure. The problem is intertwined with buffer insertion because the placement of such latches depends on the location of buffers on adjacent interconnect. In Chap. 3 we broaden the scope of timing-driven latch placement by integrating it with buffer insertion. We enhance computational scalability by employing analytical delay models and optimizing delay using state-of-the-art linear programming software.

Bounded transactional timing analysis

As local circuit optimizations become increasingly multi-objective in modern physical synthesis flows, a tighter interaction between optimization algorithms and timing analysis is necessary. Such optimizations must employ heuristics to search for good implementations of subcircuits, but many main stream timing analysis tools offer no support for retracting circuit modifications. In Chap. 4 we describe an extension to traditional static timing analysis that records a history of incremental network delay computations in a stack-based data structure, so that the timing can be returned to a previously-known state upon retraction of a circuit modification. It also explicitly *bounds* the scope of propagation to a local window in anticipation of retraction. These extensions form a necessary infrastructure for modern physical synthesis optimizations and greatly improve the performance of static timing analysis for local circuit modifications in the presence of retraction.

Simultaneous placement and gate sizing in a discrete domain

Gate locations that optimize timing depend on boundary timing conditions in the local subcircuit. Similarly, the optimal drive strength of a gate depends on the input slew rate and output capacitance. But these two problems are related because the output capacitance of a gate depends upon the length of interconnect it drives. In Chap. 5 we describe our pairwise delay model that allows us to analyze the impact of these optimizations simultaneously. Integrating gate sizing as well as threshold voltage assignment with timing-driven detailed placement allows our algorithm to explore a broader range of solutions and ultimately improve the most critical paths in the circuit.

Timing-driven gate cloning for interconnect optimization

In a complete physical synthesis flow, optimization transformations that can improve the timing on critical paths that are already well-optimized by a series of powerful transformations (timing driven placement, buffering and gate sizing) are invaluable. We develop an innovative gate cloning technique that integrates placement and buffer insertion to improve interconnect delay on critical paths during physical synthesis. Our polynomial-time algorithm simultaneously finds locations for the original and copied gates and assigns sinks to one of the copies so as to minimize interconnect delay. Our algorithm leverages analytical delay models developed in Chap. 3 and thereby accounts for the impact of future buffer insertion.

Performance-driven retiming, placement, buffering and logic cloning

One of the most common situations in which the latch placement techniques of Chap. 3 are insufficient is a critical path wherein moving a gate immediately next to its most-critical input is the optimal solution but does not meet timing constraints. For example, when relocating the latch adjacent to its only input still violates a setup time constraint, placement is insufficient to further improve timing. In order to remove this barrier, we develop SPIRE, a new physical synthesis transformation that integrates retiming with gate relocation and buffer insertion. To broaden the scope of retiming, we extend this transformation with gate duplication designed to create new retiming opportunities. We demonstrate the need for this transformation by example, motivating the integration of all considered techniques to meet timing constraints.

Broadening the scope of physical-synthesis optimization using partitioning

The optimizations developed in this book extend physical-synthesis transformations beyond a handful of gates. Unfortunately, the computational complexity of such optimizations makes them too inefficient to apply to entire netlists of large ASIC and SoC designs. Therefore, we develop a technique to identify appropriately-sized subsets of large designs on which our transformations can be applied efficiently. Our method utilizes existing hypergraph partitioning algorithms to divide the circuit in a top-down fashion until the subsets reach the desired size. We show that this technique can work in practice and demonstrate a run-time solution quality trade-off for SPIRE, the transformation developed in this book that can optimize subcircuits with thousands of standard cells.

Co-optimization of latches and clock networks in large-block physical synthesis

Optimizations developed in this book affect nearly every stage of a typical industrial state-of-the-art physical-synthesis flow. In order to obtain synergies between

them, we explore the infrastructure for physical synthesis used by IBM for large commercial microprocessor designs. We focus our attention on a very challenging high-performance design style called large block synthesis (LBS). In such designs the placement of the latches is particularly critical to the performance of the clock network, which in turn affects timing and power. Our research uncovers deficiencies in the state-of-the-art physical synthesis flow vis-à-vis latch placement that result in timing disruptions and hamper design closure. We introduce a next-generation physical synthesis methodology that seeks a more graceful timing-closure process. This is accomplished through careful latch placement and clock-network routing to (i) avoid timing degradation where possible, and (ii) immediately recover from unavoidable timing disruptions.

1.3 Organization of the Book

The rest of the book is organized as follows.

- Part I introduces our work in this chapter, and outlines relevant background on physical synthesis in Chap. 2.
- Part II covers local transformations and necessary timing analysis techniques for physical synthesis. Chapter 3 describes a method for simultaneous placement of sequential gates and buffering of incident interconnect. Chapter 4 describes a timing analysis technique that is necessary for the efficient implementation of compound transformations such as the one described in Chap. 5. Chapter 5 describes an abstract model for circuit timing under movement and repowering that can be solved optimally using branch and bound.
- Using the timing models developed in Part II, Part III develops new transformations that significantly extend the scope of existing physical synthesis optimization. Chapter 6 describes a new physical synthesis optimization for gate cloning that improves worst slack by estimating interconnect delay using the linear delay model described in Chap. 3. Chapter 7 integrates retiming, placement, cloning and static timing analysis into a unified mixed integer-linear program (MILP) that scales to circuits over 10 times larger than those presented in Chaps. 3 and 5. We apply the techniques in these chapters to larger circuits using partitioning in Chap. 8. In Chap. 9, we combine these techniques into a single methodology for application to large, high-performance designs.

References

1. Saxena P, Menezes N, Cocchini P, Kirkpatrick DA (2004) Repeater Scaling and its Impact on CAD. IEEE Trans. CAD 23(4):451–463
2. Trevillyan L et al (2004) An Integrated Environment for Technology Closure of Deep-submicron IC Designs. IEEE Des Test Comput 21(1):14–22

3. Alpert CJ et al (2007) Techniques for Fast Physical Synthesis. Proc IEEE 95:(3) 573–599
4. Alpert CJ, Chu C, Villarrubia PG (2007) The Coming of Age of Physical Synthesis, ICCAD 2007, pp 246–249
5. Srinivasan A, Chaudhary K, Kuh ES (1991) RITUAL: Performance Driven Placement Algorithm for Small Cell ICs, ICCAD 1991, pp 48–51
6. Alpert CJ, Devgan A, Quay ST (1999) Buffer Insertion with Accurate Gate and Interconnect Delay Computation, DAC 1999, pp 479–484
7. van Ginneken LPPP, Buffer Placement in Distributed RC-Tree Networks Forminimal Elmore Delay, ISCAS 1990, pp 865–868
8. Plaza SM, Markov IL, Bertacco VM (2008) Optimizing Non-Monotonic Interconnect using Functional Simulation and Logic Restructuring. IEEE Trans. CAD 27(12):2107–2119

Chapter 2
State of the Art in Physical Synthesis

Physical synthesis is a multi-phase optimization process performed during IC design to achieve *timing closure*, though *area*, *routability*, *power* and *yield* must be optimized as well. Individual steps in physical synthesis, called *transformations* are invoked by dynamic *controller* functions in complex sequences called *design flows* (EDA flows). Transformations rely on abstract delay models to analyze timing requirements and guide optimization, as illustrated in Sect. 2.3. Finally, we describe recent evolution of requirements for physical synthesis and discuss current trends.

2.1 Progression of a Modern Physical-Synthesis Flow

The physical design of a semiconductor chip begins when the design's architect formalizes plans for different components. This plan may include partitioning the functionality into hierarchical blocks, setting performance constraints, or counting the occurrences of particular functional units such as memories. Designers then write hardware description language (HDL) code to describe the behavior of the chip in a manner that can be synthesized in hardware. Logic synthesis is responsible for translating the HDL code into a gate-level netlist for the next stage. With input from the early planning stage and the netlist produced by logic synthesis, floorplanning begins to define the area of the chip and embed the circuit blocks into those physical boundaries. Figure 2.1 illustrates this process in a flow chart. After floorplanning, the design enters the physical synthesis phase, beginning with global placement. Recent publications [1, 2] describe the major phases of physical synthesis which can be briefly summarized as follows.

1. **Global placement** Computes non-overlapping physical locations for gates. Typically optimizes half-perimeter wirelength (HPWL) or weighted HPWL. During this phase, usually some amount of detailed placement is done, and legalization is called to ensure a legal optimization result.

D. A. Papa and I. L. Markov, *Multi-Objective Optimization in Physical Synthesis of Integrated Circuits*, Lecture Notes in Electrical Engineering 166,
DOI: 10.1007/978-1-4614-1356-1_2, © Springer Science+Business Media New York 2013

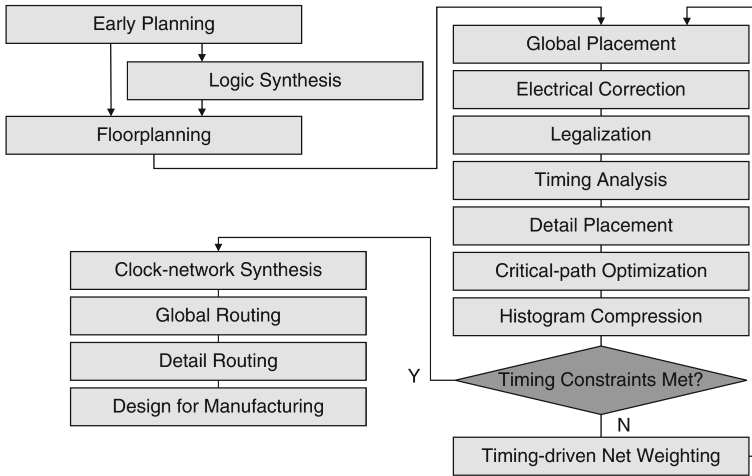


Fig. 2.1 Major stages of physical design include floorplanning and logic synthesis, followed by physical synthesis beginning with global placement, and finishing with routing and design for manufacturing. Physical synthesis can be iterated with modified parameters to improve the result, however, this flow does not always converge to an acceptable solution

2. **Electrical correction** Fixes capacitance and slew rate violations using gate sizing and net buffering.
3. **Legalization** An incremental placement capability that removes overlaps caused by circuit optimization with minimal disturbance to placement and timing.
4. **Timing analysis** Assesses the speed of the design and determines if performance targets are met. Among other metrics, this phase determines the *slack* of every path in the circuit, which is the difference between the clock period and how long it takes a signal to traverse the path.
5. **Detailed placement** Moves gates to further reduce wirelength and improve timing. In this phase it is possible to do *timing-driven* detailed placement wherein timing information is explicitly considered when optimizing gate placements.
6. **Critical-path optimization** At this point one can identify most-critical paths and can invoke a variety of techniques to increase the slack of the worst timing violations. These techniques include buffering, gate sizing, logic restructuring, etc. [3]. Figure 2.2 illustrates critical path optimization with an arrow pushing the worst paths toward increasing slack.
7. **Slack-histogram compression** When improvements on most-critical paths are no longer possible, one can improve the other paths that are less critical, but still violate timing constraints. The goal is to *compress* the slack histogram by reducing the total number of paths that fail to meet timing constraints. Figure 2.2 illustrates slack-histogram compression by a series of arrows pushing the histogram downward.

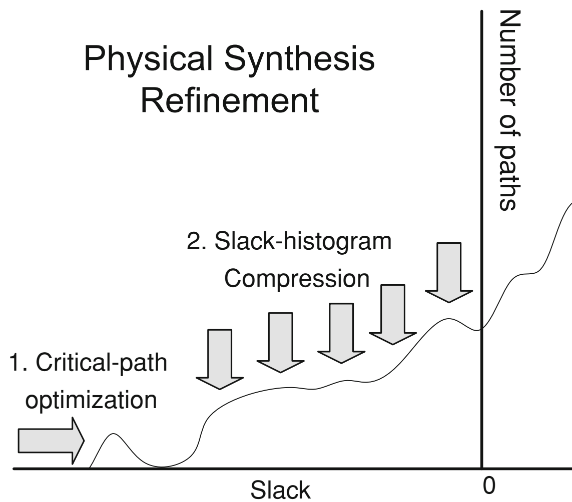


Fig. 2.2 During physical synthesis refinement, optimization is first applied to most-critical paths, then different optimizations are used to reduce the total number of critical paths

The flow can be repeated with net weighting and timing-driven placement for the first stage to potentially improve results.

With any particular flow of optimizations, at a high level, one can think of physical synthesis as progressing with increasing detail and accuracy over time, but with reduced scope and magnitude of change, as shown in Fig. 2.3. For example, during global placement, physical synthesis changes the location of all movable cells in a design but usually optimizes weighted wirelength, which is a crude model of circuit timing. Later in physical synthesis, buffers may be inserted to optimize a long wire using an Elmore interconnect-delay model with Steiner-tree estimates. As the design starts to converge, one can apply fine-grained buffering along actual detailed routes using a statistical timing model.

After physical synthesis, clock networks are formed by inserting clock buffers and routing clock nets. Next, signal nets are routed, first by global routing then by detailed routing. After routing, some optimization is usually necessary to fix any timing degradations. Finally, the design is optimized for the manufacturing process to increase the yield of functional chips.

2.2 The Controller/Transformation Approach

With a trend toward larger fractions of critical path delay in interconnect rather than in gates, it is essential for logic synthesis to be aware of physical information. A recent development, physical synthesis optimization flows address this challenge with an approach that integrates logic synthesis and physical design optimizations into a single tool.

Physical synthesis tools read a circuit that satisfies timing constraints assuming optimistic timing estimates, based on *zero wire load* models. The first step is to

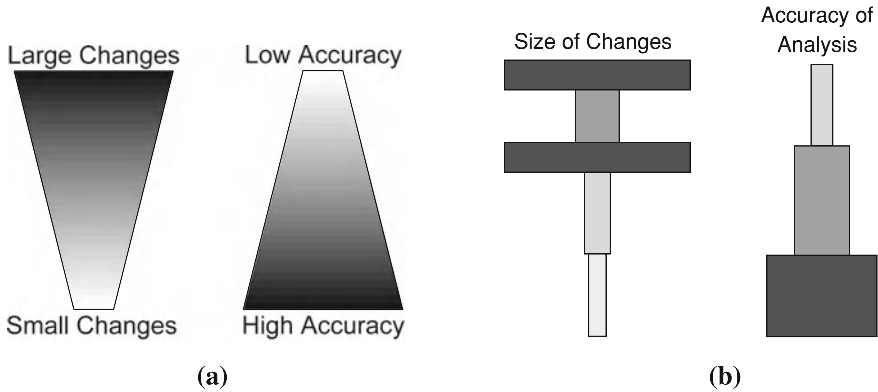


Fig. 2.3 In physical synthesis flows, the amount of change to the design is large in early phases and reduces quickly in later phases. Timing models become more accurate as the flow progresses. This trade-off is necessary because using the highest accuracy of analysis while making large changes to the design is too expensive. **a** An ideal physical synthesis flow that gradually reduces the size of changes as it increases accuracy. **b** A more realistic example flow with two global placement steps that move every gate in the design, and refinement stages that apply local optimizations to one object at a time. Accuracy is increased in discrete steps

run global wire-length driven placement, followed by the other steps introduced in Sect. 2.1. In each of the remaining phases, local *transformations* are applied to the netlist. Transformations such as buffer insertion, gate resizing, and detailed placement are applied to improve performance metrics such as timing, power consumption and yield. The decision as to which part of the netlist will be optimized is left to a *controller*, which has a focus such as the most critical nets, all critical nets, or non-critical gates. As the controller proceeds, it can call a timing analysis tool for incremental updates to provide the transformation with fresh timing data to guide its progress. In this way one can target optimizations to problem areas and produce a flow which converges on a well optimized design.

2.3 Circuit Delay Estimation

Historically, wirelength was used as a coarse metric for optimizing timing and routability during layout synthesis. Efficient algorithms have been developed to compute and optimize many different wirelength calculations, including half-perimeter wirelength (HPWL), quadratic net length, rectilinear Steiner-minimal tree (RSMT) length [4–6]. At technology nodes larger than 250 nm interconnect delay was a negligible fraction of total path delay, and merely minimizing wirelength was suitable for optimizing design performance, but this has changed. It is now necessary to consider the delay of connected wires when choosing the location of a gate. Similarly, when estimating the delay of a gate, one must consider the capacitance of nets it drives in

addition to the gates, as well as the slew rate of the input signals. Various models are used to abstract these delay calculations into an analytical form so that they can be efficiently optimized. We discuss several such abstractions in the following sections.

Elmore Delay

Elmore delay is a simple and efficient way to find the delay through a net. To compute Elmore delay of a net proceed from the sinks of a net toward the root, summing the resistance of the current segment times the downstream capacitance. This approach assumes the net is has a tree topology, which is true for virtually every signal net in digital logic. An RSMT of a net will be computed for the purposes of finding the Elmore delay through the net. This model is known to have some pessimism, but provides suitable accuracy to guide optimization in the sense that reducing Elmore delay usually results in a reduction in actual delay [7]. For example, this model could be used to efficiently estimate the delay impact of moving a gate during detailed placement. More recent works have improved upon the accuracy of the Elmore delay model. The authors of [8] improve the accuracy of Elmore delay by fitting curves to HSpice data with technology-specific parameters while maintaining a closed-form equation for delay. In addition, several technology-independent, closed-form equations for computing RC network delay were shown to have a low error while being relatively easy to implement [9, 10].

Buffered Path Delay

Buffering has become indispensable in timing closure and cannot be ignored during interconnect delay estimation [11–13]. Therefore to calculate new locations of movable gates, one must adopt a buffering-aware interconnect delay model that accounts for future buffers. We found that the linear delay model described in [13, 14] is suited to physical synthesis applications. In this model, the delay along an optimally buffered interconnect is

$$\mathrm{delay}(L) = L(R_b C + R C_b + \sqrt{2 R_b C_b R C}) \tag{2.1}$$

where L is the length of a 2-pin buffered net, R_b and C_b is the intrinsic resistance and input capacitance of buffers and gates while R and C are unit wire resistance and capacitance respectively. This model is described in more detail in Chap. 3.

Empirical results in [13] indicate that Eq. 2.1 is accurate up to 0.5% when at least one buffer is inserted along the net.

Slew Rate Propagation

One of the most costly computations in timing analysis is propagating the slew rate of a signal through the circuit. However, changes in slew rate typically do not propagate

beyond a small number of logic levels. In order to mitigate the runtime expense of accurate slew rate computation, an abstraction called *pin-slew mode* can be used. In *em path-slew mode*, all slew rates are propagated through all wires and logic to compute the slew rate at a given point in the circuit. In pin-slew mode, the slew rate at a given point is computed by looking at the previous logic stage, and asserting a *default slew* rate on its input signals. The slew rate is then propagated through that gate and its output net to find the slew rate at the given point. The default slew rate may be provided as input, or computed as the average slew rate throughout the circuit. Leveraging pin-slew mode, one can create models which are accurate, but also smaller in scope.

2.4 Current Trends in Physical Synthesis

Physical synthesis is transitioning from a novelty into a mature and highly-integrated capability required of industrial EDA flows. During this transition, the challenges in physical synthesis and greatest possibilities for improvement correspond to the following key trends.

Increased Interaction with Timing

With advancing technology nodes, increasingly aggressive and complex transformations are prone to cause inadvertent timing degradations. An increasing number of transformations have been developed that are aware of their impact on timing, congestion, wirelength, and other design performance metrics, and are capable of reversing actions that do unwanted harm.

Transformations already exist in state-of-the-art physical synthesis flows to optimize the performance of a handful of gates and nets under several known timing models, including black-box models and exhaustive search. One consequence of this level of maturity is that it is not likely, for example, that adding a new algorithm to repower one gate at a time while considering its neighbors' timing will improve the results of a modern physical synthesis flow. However, improvement is possible by increasing the scope or accuracy of such optimizations. This includes increasing the number of objects optimized simultaneously, increasing the number of objectives in the optimization, and improving the delay models used. Making these extensions affordable by decreasing their computational complexity is a key challenge addressed in this work.

Early, Accurate Analysis

Nearly every physical design objective entails a chicken-and-egg problem between analysis and optimization. For example, placement must choose non-overlapping

locations for gates such that worst slack is optimized but accurate static timing analysis (STA) requires the locations of gates to compute timing slack. This pattern repeats with such top-tier physical design metrics as timing, power, routability and yield. Traditionally, iteration-based flows have been used to break the chicken-and-egg cycle, leveraging the previous analysis to drive the subsequent optimization. This approach consumes considerable runtime, requires consistency of results from algorithms and is not guaranteed to converge to an acceptable solution. Instead, iteration cycles can be reduced or eliminated by creating fast analysis tools that accurately estimate key performance metrics during optimization and can quickly adjust estimates after incremental changes. Such an approach presumes a high level of integration between analysis and optimization tools, which requires a carefully designed software infrastructure. Improving the accuracy of such predictors and estimators as well as creating new ones presents a challenge in physical synthesis. Our work leverages accurate analysis techniques in new physical synthesis transformations that perform more comprehensive optimization of large, complex designs than existing transformations.

Large, Complex Designs

Moore's law describes the periodic doubling of transistor density in integrated circuits due to rapid improvements in manufacturing technology. At each new technology node, there are more transistors available in the same chip area and individual transistors are smaller than before. As of the writing of this book, 32 nm CPUs are widespread, 32 nm ICs are commercially available, and 22 nm designs are in early stages of development. New challenges stem from these trends as semiconductor technology approaches fundamental limits to circuit operation.

Some modern ICs contain over a billion transistors. Designing such a complex system presents enormous challenges in physical design. Perhaps the most obvious challenge is the overbearing amount of design effort required to complete such a design. Improvements in productivity due to automation have not kept pace with the rate of growth in the number of transistors on-chip. Hence, this *productivity gap* is a growing problem—fundamental improvements in automation or an increasing number of engineers will be required to complete the largest designs in future technologies. Our research will develop new transformations and new automation to improve the productivity of designers and address this key bottleneck.

References

1. Alpert CJ et al (2007) Techniques for fast physical synthesis. Proc IEEE 95(3):573–599
2. Alpert CJ, Chu C, Villarrubia PG (2007) The coming of age of physical synthesis. In: ICCAD, pp 246–249
3. Trevillyan L et al (2004) An integrated environment for technology closure of deep-submicron IC designs. IEEE Des Test Comput 21(1):14–22

4. Spindler P, Schlichtmann U, Johannes FM (2008) Kraftwerk2—a fast force-directed quadratic placement approach using an accurate net model. IEEE Trans CAD 27(8):1398–1411
5. Chu CCN, Wong Y-C (2008) FLUTE: fast lookup table based rectilinear Steiner minimal tree algorithm for VLSI design. IEEE Trans CAD 27(1):70–83
6. Roy JA, Markov IL (2007) Seeing the forest and the trees: Steiner wirelength optimization in placement. IEEE Trans CAD 26(4):632–644
7. Alpert CJ, Devgan A, Quay ST (1999) Buffer insertion with accurate gate and interconnect delay computation. In: DAC, pp 479–484
8. Abou-Seido A, Nowak B, Chu C (2004) Fitted Elmore delay: a simple and accurate interconnect delay model. IEEE Trans VLSI Syst 12(7):691–696
9. Alpert CJ, Devgan A, Kashyap CV (2001) RC delay metrics for performance optimization. IEEE Trans CAD 20(5):571–582
10. Alpert CJ, Liu F, Kashyap CV, Devgan A (2004) Closed-form delay and slew metrics made easy. IEEE Trans CAD 23(12):1661–1669
11. Cong J, He L, Koh C-K, Madden PH (1996) Performance optimization of VLSI interconnect layout. Integration VLSI J 21:1–94
12. Saxena P, Menezes N, Cocchini P, Kirkpatrick DA (2004) Repeater scaling and its impact on CAD. IEEE Trans CAD 23(4):451–463
13. Alpert CJ et al (2006) Accurate estimation of global buffer delay within a floorplan. IEEE Trans TCAD 25(6):1140–1146
14. Otten R (1998) Global wires harmful? In: ISPD, pp 104–109

Part II
Local Physical Synthesis and Necessary Analysis Techniques

Chapter 3
Buffer Insertion During Timing-Driven Placement

Physical synthesis tools are responsible for achieving timing closure. Starting with 130 nm designs, multiple cycles are required to cross the chip, making latch placement critical to success. We present a new physical synthesis optimization for latch placement called Rip up and move boxes with linear evaluation (RUMBLE) that uses a linear timing model to optimize timing by simultaneously re-placing multiple gates. RUMBLE runs incrementally and in conjunction with static timing analysis to improve the timing for critical paths that have already been optimized by placement, gate sizing, and buffering. The contributions in this chapter improve the detailed placement and critical path optimization stages of physical synthesis as illustrated in Fig. 3.1.

3.1 Introduction

Physical synthesis is a complex multi-phase process primarily designed to achieve timing closure, though power, area, yield and routability also need to be optimized. Starting with 130 nm designs, signals can no longer cross the chip in a single cycle, which means that *pipeline latches* need to be introduced to create multi-cycle paths. This problem becomes more pronounced for the 90-, 65- and 45-nanometer nodes, where interconnect delay increasingly dominates gate delay [1]. Indeed, for high-performance ASIC scaling trends, the number of pipeline latches increases by 2.9$\times$ at each technology generation, accounting for as much as 10 % of the area of 90 nm designs [2] and as many as 18 % of the gates in 32 nm designs [3]. Hence, the proper placement of pipeline latches is a growing problem for timing closure.

The choice of computational techniques for latch placement depends on where this optimization is invoked in a physical synthesis flow. In Chap. 2 we described the major phases of physical synthesis: global placement, electrical correction, legalization, timing analysis, detailed placement, critical-path optimization and compression, which may be iterated with timing-driven placement to improve solution quality.

D. A. Papa and I. L. Markov, *Multi-Objective Optimization in Physical Synthesis of Integrated Circuits*, Lecture Notes in Electrical Engineering 166,
DOI: 10.1007/978-1-4614-1356-1_3, © Springer Science+Business Media New York 2013

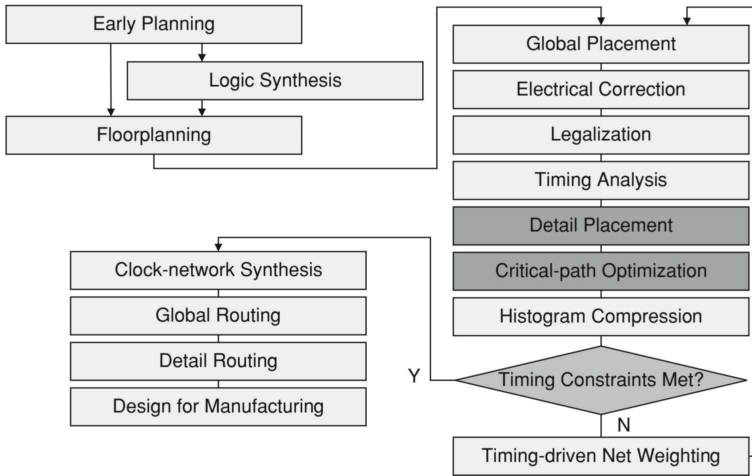


Fig. 3.1 The contributions in this chapter improve the state of the art in critical path optimization and timing-driven detailed placement

We argue that pipeline latches should be placed only after some amount of timing analysis and optimization.

Figure 3.2a–d illustrates the complications of using existing global placement techniques to solve the latch placement problem for a single two-pin net. Assume that, for all four figures, the source A and sink B are fixed in their respective locations, and that global placement must find the correct location for the latch. This example is representative of situations in which a fixed block in one corner of the chip must communicate with a block in the opposite corner, but signal delay inevitably exceeds a single clock period. All four placements have equal wirelength, therefore unless global placement is timing driven, the placement of the latch between A and B is arbitrary. Consider the following scenarios:

- Suppose the placement tool chooses (a), which is the worst location for the latch. In this case, the latch is so far from B that the timing constraint at B cannot be met. This results in a slack on the input net (n_1) of $+5$ ns and a slack on the output net (n_2) of -5 ns (even after optimal buffering).[1]
- With a second iteration of physical synthesis, timing-driven placement could try to optimize the location of this latch by adding net weights. Any net weighting scheme will assign a higher weight to net n_2 than n_1, resulting in a placement where the latch is very close to B, as in (b). While the timing is improved, there now is a slack violation on the other side of the latch with -3 ns of slack on n_1 and $+3$ ns on n_2.

[1] The nets in each scenario could include buffers without changing the trends discussed.

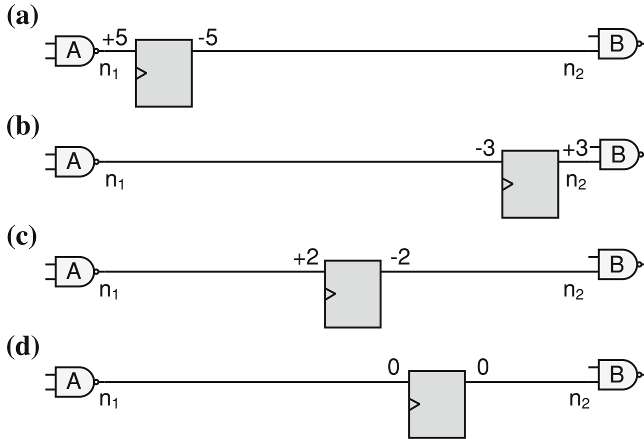


Fig. 3.2 The placement of a pipeline latch impacts the slacks of both input and output paths. A wirelength objective does not capture the timing effects of this situation, and with equal net weights a placer may choose the configuration in **a**. In trying to fix this path, timing-driven net weighting would increase the weight on net n_2, and placement would then choose the configuration in **b**. Placing the latch in the center as in **c** is also not an optimal approach. There may be only a single optimal location as shown in **d**

- A global or detailed placer could use a quadratic wirelength objective to handle these kinds of nets, giving the location (c), which, while better than (a) and (b), is suboptimal.
- To achieve the optimal location with no critical nets (0 slack on n_1 and n_2), the latch must be placed as shown in (d). In this case, there is only one location that meets both constraints.

This example suggests that wirelength optimization is not well-suited for latch placement, especially when there is little room for error. Instead, one must be able to couple latch placement with timing analysis and model the impact of buffering. The problem is more complex in practice, and some aspects are not illustrated above. In particular, many latches have buffer trees in the immediate fanin and fanout. Such complications pose additional challenges that we address in this work. We make the following contributions.

- We show that a linear-wire-delay model is sufficient to model the impact of buffering for the latch placement problem.
- We develop RUMBLE, a linear-programming-based, timing-driven placement algorithm which includes buffering for slack-optimal placement of individual latches under this model and show its effectiveness experimentally.
- We extend RUMBLE to improve the locations of individual logic gates other than latches. Further, we show how to find the optimal locations of multiple gates (and latches) *simultaneously*, with additional objectives. Incremental placement of multiple cells requires additional care to preserve timing assumptions, optimizing

a set of slacks instead of a single slack, while also biasing the solution towards placement stability. We describe how RUMBLE handles these situations.

- We empirically validate proposed transformations and the entire RUMBLE flow. We show how these techniques can be used to significantly improve initial latch placement in a reasonably optimized ASIC design. Our *do-no-harm* acceptance criteria reject solutions if any quality metrics are degraded. This key feature facilitates the use of RUMBLE later in physical synthesis.

The remainder of this chapter is organized as follows. Section 3.2 discusses background and previous work. Section 3.3 describes the timing model we use in this work. Section 3.4 describes how RUMBLE performs timing-driven placement. Section 3.5 describes the RUMBLE algorithm. Section 3.6 shows experimental results. Conclusions are drawn in Sect. 3.7.

3.2 Background

Several approaches improve IC performance by modifying wirelength-driven global placement through timing-based net weights [4–9]. Such algorithms are generally referred to as timing-driven placement, but the literature has not yet considered the impact of buffering on latch placement during global placement. Due to the lack of such algorithms, it is inevitable that some latches will be suboptimally placed during global placement. Therefore, new algorithms are needed for post-placement performance-driven incremental latch movement.

We introduce a high-level description of the incremental latch placement problem below, and elaborate on its multi-move formulation in Sect. 3.4. Given an optimized design and a small set of gates M, e.g., a single latch, find new locations for each gate in M and new buffering solutions for nets incident to M such that the timing characteristics of the design are improved.

While moving a poorly placed cell can improve adjacent interconnect delay, moving a latch has special significance since it facilitates time-borrowing: reallocating circuit delay from a longer (slow) combinational stage to a shorter (fast) combinational stage. This fact offers a particularly significant boost to our basic approach, and is enhanced even further when surrounding gates are also free to move.

An optimization that performs operations such as moving a gate or latch is called a *transformation* using the terminology of [10]. Transformations are designed to incrementally improve design objectives such as timing. Other examples of transformations include buffering a single net, resizing a gate, cloning a cell, swapping pins on a gate, etc. The way transformations are invoked in a physical synthesis flow is determined by the *controllers*. For example, a controller designed for critical path optimization may attempt a transformation on the 100 most critical cells. A controller designed for the compression stage (see Sect. 3.1) may attempt a transformation on every cell that fails to meet its timing constraints.

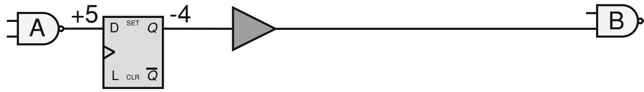


Fig. 3.3 A poorly-placed latch with buffered interconnect. In this case, the buffer must be moved or removed in order to have the freedom to move the latch far enough to fix the path

A controller has the option of avoiding transformations that may harm the design (e.g., generating new buffering solutions inferior to the original) and can then reject this solution. This *do no harm* philosophy of optimization has received significant recognition in recent work [11, 12]. The RUMBLE approach adopts this same convention which makes it more trustworthy in a physical synthesis flow.

While no previous work has attempted to solve this particular problem, other solutions do exist that may be able to help with the placement of poorly placed latches. The authors of [13] propose a linear programming formulation that minimizes downstream delay to choose locations for gates in field-programmable gate arrays (FPGAs). The authors of [14] model static timing analysis (STA) in a linear programming formulation by approximating the quadratic delay of nets with a piecewise-linear function. Their formulation's objective is to maximize the improvement in total negative slack of timing end points. The authors of both approaches conclude that the addition of buffering would improve their techniques [13, 14]. When these transformations are applied at the same point in a physical synthesis flow that we propose, they will be restricted by previous optimizations. When applied somewhat earlier (e.g., following global placement) they are incapable of certain improvements. Namely, downstream optimizations, such as buffer insertion, gate sizing, and detailed placement may invalidate the optimality of latch placement. Therefore our technique focuses on the bad latch placements that we observed in large commercial ASIC designs after state-of-the-art physical synthesis optimizations.

3.3 The RUMBLE Timing Model

We now introduce the timing model critical to RUMBLE's success.

Figure 3.3 shows an intuitive example of the problem when we try to find new locations for movable gates. Similar to Fig. 3.2, the latch has to be moved to the right to improve timing. However, since the latch drives a buffer which is placed next to it, we must move the buffer in order to improve the slack of the latch. Other challenges in latch placement are illustrated by Fig. 3.4. At the same time, the optimal new location of the latch depends on how the input and output nets are buffered. As a result, the optimal approach is to simultaneously move the latch and perform buffering, but this is computationally prohibitive because a typical multiple-objective buffering algorithm runs in exponential time. As mentioned in Sect. 3.1, we propose a sequential approach in which we first compute the new locations for a selected set

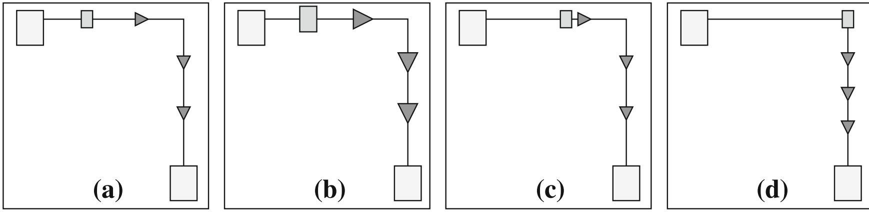


Fig. 3.4 The layout in **a** has a poorly-placed latch, and existing critical path optimizations do not solve the problem. Repowering the gates may improve the timing some in **b**, but if it cannot fix the problem, the latch must be moved. Moving the latch up to the next buffer, shown in **c**, does not give optimization enough freedom. If we move the latch but do not re-buffer in **d**, timing may degrade. Figure 3.12d shows the ideal solution to this problem

of movable gates based on timing estimation considering buffers. Then, buffering is applied to the input and output nets of the selected movable gates. Being practical, effective and efficient, this approach can be integrated into a typical VLSI physical synthesis flow. The calculation of optimal movement uses a simple but effective buffered-interconnect delay model, which is discussed next.

Linear Buffered-Path Delay Estimation

Buffering has become indispensable in timing closure and cannot be ignored during interconnect delay estimation [3, 15, 16]. Therefore to calculate new locations of movable gates, one must adopt a buffering-aware interconnect delay model that accounts for future buffers. Consider the problem of estimating the delay of an optimally-buffered net of arbitrary length L. We briefly review an analytical delay model that is well-suited to this purpose [15, 17]. Consider the delay of a long chain of buffers as shown in Fig. 3.5a. Suppose there are k buffers driving wire segments each of which are length Γ. The model is simplified by assuming the size of a buffer is negligibly small, then $\Gamma = \frac{L}{k}$. Assume that each buffer and wire segment it drives is modeled by the RC-network in Fig. 3.5b. Then the delay of the whole chain of buffers of length L is computed as k times the delay through each segment.

$$\text{delay}(L) = k\left[R_b\left(\frac{L}{k}C\right) + R_bC_b + \left(\frac{L}{k}R\right)\left(\frac{L}{k}C\right) + \left(\frac{L}{k}R\right)C_b\right] \tag{3.1}$$

Where L is the length of a 2-pin buffered net, R_b and C_b are the intrinsic resistance and input capacitance of buffers and gates while R and C are unit wire resistance and capacitance respectively.

The model is further simplified by assuming continuous gate sizes and placement sites. Then optimal buffering solutions minimize the delay function as follows.

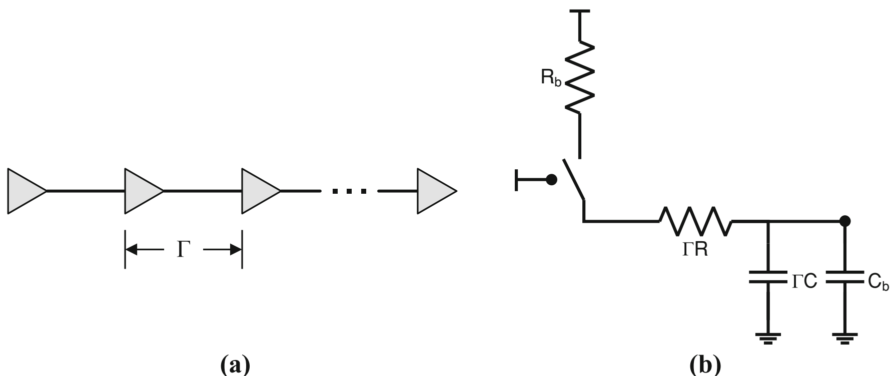


Fig. 3.5 **a** A model for buffered interconnect. Γ describes the optimal distance between buffers on a two-pin net. **b** A corresponding RC-network of a single buffer driving a wire segment. R_b and C_b represent the intrinsic resistance and gate capacitance of the buffer while R and C represent the per-unit resistance and capacitance of a metal wire

$$\frac{\delta(\text{delay}(L))}{\delta k} = 0 \tag{3.2}$$

Which leads to this relation on the optimal buffering solution.

$$\frac{L}{k} = \sqrt{\frac{R_b C_b}{RC}} \tag{3.3}$$

By subtituting Eq. 3.3 into 3.1 we can simplify the calculation of delay to the following.

$$\text{delay}(L) = L(R_b C + RC_b + \sqrt{2R_b C_b RC}) \tag{3.4}$$

Note that this equation is linear in terms of L.

Empirical results in [15] indicate that Eq. 3.4 is accurate up to 0.5 % when at least one buffer is inserted along the net. Furthermore, our own empirical results in Sect. 3.6 suggest a 97 % correlation between this linear delay model and the output of an industry timing-analysis tool.

The Timing Graph

In RUMBLE, a set of movable gates is selected, which must include fixed gates or input/output ports to terminate every path. Fixed gates and I/Os help formulate timing constraints and limit the locations of movables. In Fig. 3.6a, we assume that new locations have to be computed for the latch and the two OR gates, while all NAND gates are kept fixed.

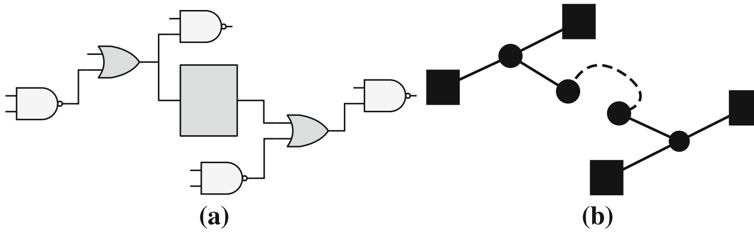


Fig. 3.6 **a** An example subcircuit and **b** corresponding timing graph used in RUMBLE. The AATs or RATs of unmovable objects (*squares*) are considered known. STA is performed on movable objects (*round shapes*)

In the timing graph, each logic gate is represented by a node, while a latch is represented by two nodes because the inputs and outputs of a latch are in different clock cycles and can have different slack values. Each edge represents a driver-sink path along a net and is associated with a delay value which is linearly proportional to the distance between the driver and the sink gate. In other words, we decompose each multi-pin net into a set of two-pin edges that connect the driver to each sink of the net. This simplification is crucial to our linear delay model and is valid because the linear relationship can be preserved for the most critical sinks by decoupling less-critical paths with buffers [15]. Therefore the two-pin edge model in the timing graph can guide the computation of new locations for the movable gates.

In the timing graph, an edge which represents a timing arc is created only for (1) each connection between the movable gates, and (2) each connection between a movable gate and a fixed gate. This is because we only care about the slack change due to the displacement of movable gates. For the subcircuit in Fig. 3.6a, the resultant timing graph is shown in Fig. 3.6b.

For each fixed gate, we assume that the required arrival time (RAT) and the actual arrival time (AAT) are fixed. The values of RAT and AAT are generated by a static timing analysis (STA) engine using a set of timing assertions created by designers. An in-depth exposition of STA can be found in [18, 19] along with algorithms to generate RAT and AAT. A movable latch corresponds to two nodes in the timing graph, one for the data input pin and one for the output pin. For the input pin, the RAT is fixed based on the clock period. Similarly, the AAT is fixed for the latch's output pin. Based on all the fixed RAT and AAT at fixed gates and latches, the AAT and RAT are propagated along the edges according to the delay of the timing arcs. The values of AAT are propagated forward to fanout edges, adding the edge delay to the AAT. On the contrary, RATs are propagated backward along the fanin edges, subtracting the edge delay from the RAT values. Details of edge delay, RAT and AAT calculation in our algorithm are covered in Sect. 3.4.

3.4 Timing-Driven Placement with Buffering

The goal of RUMBLE is to find new locations for movable gates in a given selected subcircuit such that the overall circuit timing improves. Therefore we maximize the minimum (worst) slack of source-to-sink timing arcs in the subcircuit. In contrast to other objectives used in previous work, we select this objective because we are targeting critical-path optimization. Hence, we prefer 1 unit of worst-slack improvement over 2 units of slack improvement on less-critical nets. Below we introduce the timing-driven placement technique in RUMBLE that directly maximizes minimum slack. In the following placement formulation we account for the timing impact of our changes by implicitly modeling static timing analysis in our timing graph. In this work, we estimate net length by the half-perimeter wirelength (HPWL) and then scale it to represent net delay. More accurate models are possible, but may complicate optimization.

Problem Formulation

Consider the problem of maximizing the minimum slack of a given subcircuit G with some movable gates and some fixed gates, or ports. Let the set of nets in the subcircuit be $\mathbf{N} = n_0, n_1, \ldots, n_h$. Let the set of all gates in the subcircuit (movable and fixed) be $\mathbf{G} = g_0, g_1, \ldots, g_f$. Let the set of movable gates in the subcircuit (a subset of $\mathbf{G}$) be $\mathbf{M} = m_0, m_1, \ldots, m_k$. τ is a technology-dependent parameter that is equal to the ratio of the delay of an optimally-buffered, arbitrarily-long wire segment to its length

$$\tau = \frac{\text{delay}(wire)}{\text{length}(wire)} \tag{3.5}$$

The following equations govern static timing analysis and are used in the next section. A timing arc is specified for a given net n driven by gate u and having sink v as $n_{u,v}$, as illustrated by Fig. 3.7a. The delay of a gate g is D_g.

The calculation of Required Arrival Time (RAT) and Actual Arrival Time (AAT) of a gate for combinational circuits shown in Fig. 3.7 are computed as follows. The RAT of a combinational gate g

$$R_g = \min_{o_j : 0 \le j \le m} \{R_{o_j} - \tau * \text{HPWL}(n_{g,o_j}) - D_g\} \tag{3.6}$$

The AAT of a combinational gate g is

$$A_g = \max_{i_j : 0 \le j \le l} \{A_{i_j} + \tau * \text{HPWL}(n_{i_j,g}) + D_g\} \tag{3.7}$$

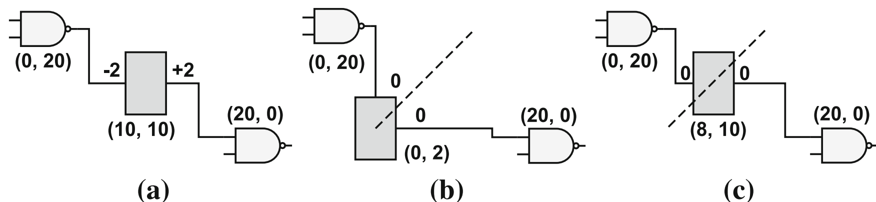


Fig. 3.7 **a** A timing arc $n_{u,v}$ connecting an arbitrary gate u to an arbitrary gate v. **b** The RAT of a gate g is the minimum of RATs of the outputs of g. **c** The AAT of a gate g is the maximum of AATs of the inputs of g

Given a clocked latch r, we assume for simplicity that the RAT (R_r) and AAT (A_r) are fixed and come from the timer. Unclocked latches are treated similarly to the combinational gates above.

The slack of a timing arc $n_{p,q}$ connecting two gates (combinational or sequential, movable or fixed) p and q is

$$S_{n_{p,q}} = R_q - A_p - \tau * \text{HPWL}(n_{p,q}) \tag{3.8}$$

The RUMBLE Linear Program

We now define a linear program that maximizes the minimum slack S of a subcircuit as follows.
VARIABLES:

For each movable object m in M we define two independent variables representing the location (x, y) of m: β_x^m, β_y^m.

In terms of these locations, we define the bounding box of each net n using four new variables representing lower-left coordinate: L_x^n, L_y^n
as well as the upper-right coordinate: U_x^n, U_y^n.

Given a gate m, the actual arrival time at the output of m is defined using the variable: A_m.

The required arrival time at the input of the gate m is similarly defined using the variable: R_m.

The slack of each net n is defined using the variable: S_n.

The minimum slack of all S_n variables is computed using the variable: S.

Objective: **Maximize** S
Constraints: For every gate g_j on net n_i

$$U_x^{n_i} \geq \beta_x^{g_j}, \quad U_y^{n_i} \geq \beta_y^{g_j} \tag{3.9}$$

$$L_x^{n_i} \leq \beta_x^{g_j}, \quad L_y^{n_i} \leq \beta_y^{g_j} \tag{3.10}$$

For every movable gate m_i and sink it drives g_j via net n_k

$$R_{m_i} \leq R_{g_j} - \tau * (U_x^{n_k} - L_x^{n_k} + U_y^{n_k} - L_y^{n_k}) - D_g \tag{3.11}$$

For every movable gate m_i and gate g_j that drives one of its inputs via net n_k

$$A_{m_i} \geq A_{g_j} + \tau * (U_x^{n_k} - L_x^{n_k} + U_y^{n_k} - L_y^{n_k}) + D_g \tag{3.12}$$

For every timing arc in the subcircuit $n_{p,q}$ associated with net n_i

$$S_{n_i} \leq R_q - A_p - \tau * (U_x^{n_i} - L_x^{n_i} + U_y^{n_i} - L_y^{n_i}) \tag{3.13}$$

For each net n_i:

$$S \leq S_{n_i} \tag{3.14}$$

Extensions to Minimize Displacement

The linear program of RUMBLE is defined to maximize the minimum slack of a subcircuit. Additional objectives can be considered as well, such as total cell displacement, which sums Manhattan distances between cells' original and new locations. We subtract the minimum slack objective from a weighted total cell displacement term to avoid unnecessary cell movement. The weight W_d for the total cell displacement objective is set to a small value. Therefore the weighted total displacement component is used as a tie-breaker and has little impact on worst-slack maximization. Instead, the combined objective is maximized by a slack-optimal solution closest to cells' original locations. During incremental timing-driven placement, minimizing total cell displacement encourages higher placement stability and often translates into fewer legalization difficulties.

Figure 3.8 shows an example of the RUMBLE formulation with and without the total displacement objectives. The only movable object in Fig. 3.8a is the latch. An input net n_1 and an output net n_2 are connected to the latch with slacks -2 and $+2$ respectively. Figure 3.8b shows the optimal LP solution without the total displacement objective. The Manhattan net length of n_1 is reduced from 20 to 18, and the net length of n_2 is increased from 20 to 22. This improves the worst slack of the subcircuit from -2 to 0. However, the latch moves a large distance. Figure 3.8c illustrates

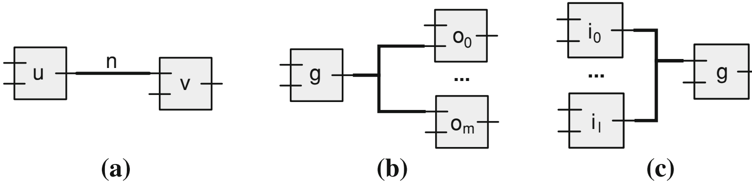


Fig. 3.8 In many subcircuits there are multiple slack-optimal placements. In RUMBLE we add a secondary objective to minimize the displacement from the original placement. This helps to maintain the timing assumptions made initially and reduces legalization issues. **a** shows the initial state of and example subcircuit, **b** a slack-optimal solution commonly returned by LP solvers, all optimal solutions lie on the dotted line and **c** a solution given by RUMBLE that maximizes worst-slack then minimizes displacement

that including the total displacement objective may preserve optimal slack, while minimizing latch displacement.

In order to minimize displacement by adding a new objective, we introduce the following variables and constraints to the linear program.

Displacement variables:
Given a gate m, define the upper bounds on the new and original coordinates in the x and y dimensions using two new variables: ϕ_x^m, ϕ_y^m.

Similarly define the lower bounds on the new and original coordinates in the x and y dimensions for the gate m using two new variables: ω_x^m, ω_y^m.

Then, in terms of ϕ and ω we define the displacement of the gate m in the x and y dimensions using two variables: δ_x^m, δ_y^m.

Displacement constraints:
For every movable gate m_i, $\alpha_x^{m_i}$ and $\alpha_y^{m_i}$ denote the original x and y coordinates. The upper and lower bounds of the new and original coordinates ϕ and ω in each dimension are:

$$\begin{aligned} \phi_x^{m_i} &\geq \beta_x^{m_i}, \quad \omega_x^{m_i} \leq \beta_x^{m_i} \\ \phi_y^{m_i} &\geq \beta_y^{m_i}, \quad \omega_y^{m_i} \leq \beta_y^{m_i} \\ \phi_x^{m_i} &\geq \alpha_x^{m_i}, \quad \omega_x^{m_i} \leq \alpha_x^{m_i} \\ \phi_y^{m_i} &\geq \alpha_y^{m_i}, \quad \omega_y^{m_i} \leq \alpha_y^{m_i} \end{aligned} \tag{3.15}$$

The displacements δ^{m_i} for a movable gate m_i are defined as

$$\delta_x^{m_i} = \phi_x^{m_i} - \omega_x^{m_i}, \quad \delta_y^{m_i} = \phi_y^{m_i} - \omega_y^{m_i} \tag{3.16}$$

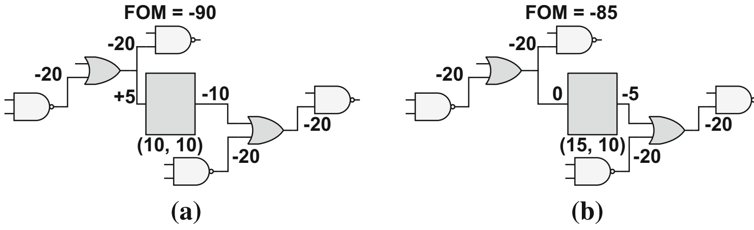


Fig. 3.9 **a** An example subcircuit with an imbalanced latch whose worst-slack cannot be improved. Nevertheless, it is possible to improve timing of the latch while maintaining slack-optimality. By including a TNS component in the objective, the total negative slack can be reduced, as shown in **b**

Extensions to Improve the Slack Histogram

The minimum slack is the worst slack in a subcircuit. For two subcircuits with identical worst slack, it is possible that one subcircuit has few critical paths with worst slack while the other one has many. A timing optimization has to improve both the worst slack and the overall total threshold slack (TTS) in a subcircuit. TTS is defined as the sum of all slacks below a threshold. If the slack threshold is zero, TTS is equivalent to the total negative slack. With the minimum slack as the only objective, a small improvement in the worst slack may cause a large TTS degradation. Therefore we must add a TTS component to the optimization objective. The balance between the minimum slack and the TTS is controlled by a parameter W_f, which is set to a relatively small value because the worst slack objective is more important.

Figure 3.9 shows another scenario where the TTS component may help. During optimization, it may not be always possible to improve the minimum slack of the subcircuit. In that case, we can still reduce the number of critical cells by improving the TTS. In Fig. 3.9, there are three movables in the subcircuit. The minimum slack of the subcircuit is -20, and it is not possible to improve the minimum slack by moving any of the gates. With the additional TTS component in the objective, the TTS of the subcircuit is improved from -90 to -85, as shown in Fig. 3.9b.

Let S_n denote the slack on net n, then the combined objective has the displacement and TTS components

Maximize:

$$S - W_d\Big[\sum_{m \in M} (\delta_x^m + \delta_y^m)\Big] + W_f\Big[\sum_{n: n \in N, S_n < T_s} S_n\Big] \tag{3.17}$$

where T_s is the small slack threshold used to compute the TTS. We have earlier assumed W_f and W_d to be small, with $W_d < W_f$. In our implementation we set W_f to 0.005 times the absolute value of the average slack in the subcircuit, and we set W_d to 10^{-6}. These additional terms change the optimal region, but because the weights are so small the combined optimal region is very near the slack-optimal region.

Preserving the TTS Objective

The primary goal of the RUMBLE linear program as presented in previous sections is to maximize the worst slack of the subcircuit. We define two additional objectives—one preserves the initial solution as much as possible, the other can improve the slack histogram when the worst slack cannot be further improved. However, it is possible that in order to improve a single worst slack path, multiple paths may degrade to the point of being critical. If RUMBLE is deployed late enough in a physical synthesis flow, the corresponding TTS degradation may be undesirable. To address this problem, we have devised an additional constraint that can prevent this type of TTS degradation, but may restrict improvement in worst slack. When TTS should not be degraded, we add the following constraints to the RUMBLE linear program to preserve TTS.

For each net n_k whose slack is greater than the slack threshold T_s, add the following constraint.

$$S_{n_k} \geq T_s \tag{3.18}$$

This addition may over-constrain the linear program, in which case it is not possible to improve the worst slack without harming TTS.

3.5 The RUMBLE Algorithm

In this section we discuss the details of the RUMBLE algorithm, which employs the linear program from the previous section to incrementally improve the timing of poorly placed latches.

Subcircuit Selection

RUMBLE identifies *imbalanced latches*, which we define as those that exhibit positive slack on their inputs and negative slack on their outputs (or vice versa). As illustrated in Fig. 3.2, the movement of any such imbalanced latch has the potential to improve timing, even if all surrounding cells are held fixed. More generally, however, the neighbors and extended neighbors of the targeted latch may also be included to form a set M of movable cells. In our technique, shown in Fig. 3.11, we adopt a basic N-hop neighborhood approach, where any gate within N steps of the imbalanced latch is included in the set of movable cells. This requires both a forward sweep (to collect sinks) and a backward sweep (to collect sources), which are performed in tandem. Those cells that are $N + 1$ steps away from the latch form a set P of fixed peripheral nodes.[2]

---

[2] Variations on this theme, such as metrics that incorporate the degree of neighbors' criticality [13, 20] and the size of the subcircuit bounding box are also possible.

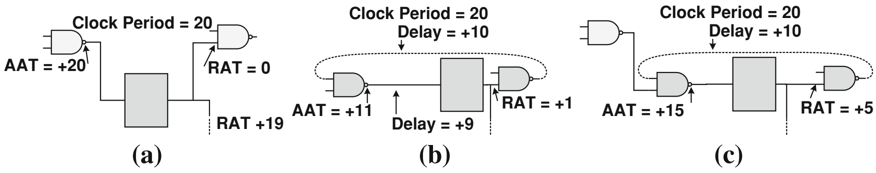


Fig. 3.10 Modeling feedback paths within logic requires a new type of gate. Pseudomovable gates have timing values that depend on the timing values of neighboring gates, but they cannot be moved. **a** Ignoring the presence of feedback paths is overly pessimistic, and it appears that the timing of the latch cannot meet its constraints. **b** Making the fixed gates along a feedback path pseudomovable allows the latch to meet its timing constraints, but doing only this can lead to the wrong placement. **c** Including all gates connected to pseudomovables as fixed timing points properly models the problem as a convex subcircuit

In contrast to prior work that has assumed operation within a pre-buffering stage, our subcircuit selection algorithm must address the presence of buffers. These buffers will be encountered in our neighborhood selection algorithm, as they are part of the current logic; however, since it is presumed that they would be ripped up when new locations are determined (a critical assumption that makes our linear-delay model possible), we must prevent their inclusion in our model of the subcircuit. Therefore, when fetching adjacent gates, we transparently skip these buffers and omit them from the set M. The recursive functions TRUE-SOURCE() and TRUE-SINK() in Fig. 3.11 provide this additional level of indirection, returning only those combinational gates that reflect the logical structure of the subcircuit. Buffers are removed and reinserted on adjacent nets by a state-of-the-art buffer insertion algorithm after RUMBLE moves gates.

Feedback Paths

As noted in [13], the process of extracting gates to form a subcircuit may suffer from complications when subpaths of combinatorial logic between peripheral nodes are not modeled. These subpaths introduce additional timing constraints that, if left absent from the model, could invalidate the optimality of the solution.

To illustrate, consider the example in Fig. 3.10, in which a single latch has been selected as a movable gate. After collecting its inputs and outputs, a simple subcircuit is constructed as shown in Fig. 3.10a, with the two endpoints shown selected as fixed gates. With the timing constraints as given in the figure, an optimal solution to this problem will place the latch equidistantly from both endpoints to ensure that the slacks on either side are balanced. However, consider a scenario where a feedback path exists from the output to the input, as shown in Fig. 3.10b; in such an event, the RAT of the output and the AAT of the input are *dependent* on the location of the latch. If this dependency is modeled, the solution may be biased toward one of the two neighbors. We loosely refer to these neighbors as *pseudomovable* gates. Although timing must be propagated through them (as it is for movable gates), their physical locations may be fixed.

```
BUILD-SUBCIRCUIT-FROM-SEED
   ▷ Input: Latch L , int N-hops
   ▷ Output: Set movables , Set pseudo , Set fixed
1  movables =
   BUILD-MOVABLES-FROM-SEED(L, N-hops)
2  pseudo =
   BUILD-PSEUDOMOVABLES-FROM-MOVABLES(movables)
3  fixed =
   BUILD-FIXED-FROM-CORE(movables ⋃ pseudo)
```

```
BUILD-MOVABLES-FROM-SEED
   ▷ Input: Latch L , int N-hops
   ▷ Output: Set movables
1  inputs = input-fringe = {L}
2  outputs = output-fringe = {L}
3  for i = 1 .. N-hops
4     input-fringe =
      ⋃ ( GET-INPUTS(input ∈ input-fringe) )
5     output-fringe =
      ⋃ ( GET-OUTPUTS(output ∈ output-fringe) )
6     inputs = inputs ⋃ input-fringe
7     outputs = outputs ⋃ output-fringe
8  movables = inputs ⋃ outputs
```

```
BUILD-PSEUDOMOVABLES-FROM-MOVABLES
   ▷ Input: Set movables
   ▷ Output: Set pseudo
1  pseudo = ∅
2  do
3     Set fan_in = INPUT-CONE(movables ⋃ pseudo)
4     Set fan_out = OUTPUT-CONE(movables ⋃ pseudo)
5     Set pseudo′ = (fan_in ⋂ fan_out ) - movables - pseudo
6     pseudo = pseudo ⋃ pseudo′
7  while pseudo′ ≠ ∅
```

```
BUILD-FIXED-FROM-CORE
   ▷ Input: Set core
   ▷ Output: Set fixed
1  fixed = ∅
2  for each Gate G ∈ core
3     Set neighbors = GET-INPUTS(G) ⋃ GET-OUTPUTS(G)
4     fixed = fixed ⋃ (neighbors - core)
```

```
GET-INPUTS
   ▷ Input: Gate G
   ▷ Output: Set inputs
1  S = ∅
2  for each pin ∈ IN-PINS(G)
3     S = S ⋃ TRUE-SOURCE(pin)
4  return S
```

```
GET-OUTPUTS
   ▷ Input: Gate G
   ▷ Output: Set outputs
1  S = ∅
2  for each pin ∈ OUT-PINS(G)
3     S = S ⋃ TRUE-SINKS(pin)
4  return S
```

```
TRUE-SOURCE
   ▷ Input: Pin p
   ▷ Output: Gate source
1  Net net = NET(p)
2  Gate G = DRIVER(net)
3  unless IS-BUFFER(G)
4     return G
5  p = IN-PIN(G)
6  return TRUE-SOURCE(p)
```

```
TRUE-SINKS
    ▷ Input: Pin p
    ▷ Output: Set sinks
 1  Net net = NET(p)
 2  Set driven = DRIVEN(net)
 3  S = ∅
 4  for each Gate G ∈ driven
 5     if IS-BUFFER(G)
 6           p = OUT-PIN(G)
 7           S′ = TRUE-SINKS(p)
 8     else S′ = G
 9     S = S ⋃ S′
10  return S
```

Fig. 3.11 Subcircuit selection transparently skips buffers when building a neighborhood of movable gates, and requires detection of *pseudomovables*

Pseudomovables are collected by intersecting the transitive cones of logic between inputs and outputs to detect feedback paths, as shown in the pseudocode of Fig. 3.11.[3] To ensure accuracy, the inputs and outputs of pseudomovables themselves must be bounded by fixed endpoints, as shown illustrated in Fig. 3.10c. These fringe nodes completely isolate the timing of the resulting *convex* subcircuit from outer cones of logic.

[3] To improve runtime, one can limit the depth of these cones to a reasonably small constant, as opposed to the exhaustive expansion in [11].

The do-no-harm Philosophy

After gates are moved, it is likely that timing has degraded due to, for example, a capacitance violation on a long wire. The subcircuit must be examined, and its interconnect improved through physical synthesis optimizations, which might include gate-sizing and buffer-insertion for delay or electrical considerations on nets.

Even though the linear program of Sect. 3.4 can be solved optimally, it does not account for all the complexities of interconnect optimization. The linear program is an abstraction of the subcircuit timing that models physical synthesis optimizations (e.g., virtual buffering) by prorating wire delay constants based on upcoming physical synthesis optimizations. Despite the high correlation to more accurate timing models in experimental results, the RUMBLE model ignores certain constraints and legalizing its solution might result in a timing degradation. For example, nets can cross blockages or congested regions with no nearby legal locations. As a result, legalization could create a timing degradation.

When running RUMBLE in our physical synthesis flow, we mitigate the harmful effects of legalization by finding legal locations for gates and buffers when moving or inserting them. Insisting on legal locations can also contribute to a degradation not anticipated by the RUMBLE model. Fortunately, RUMBLE can examine the timing implications of its changes before committing to them. It simply stores the initial state of the subcircuit, and restores it if a timing degradation occurs. In this way, RUMBLE will *do no harm* to the circuit by ensuring that whatever solution it keeps is no worse than what existed before. Such safe delay optimizations are more easily inserted into physical synthesis flows [11, 12].

The RUMBLE Algorithm

Figure 3.13 shows pseudocode for the RUMBLE algorithm, which assumes a set of movable gates given at input, and Fig. 3.12 illustrates the process. First, the subcircuit that is necessary for incremental placement is extracted (for a single movable, we extract its one-hop neighborhood of input gates). During this process, buffers are ignored (viewed as wires) as described in Sect. 3.5. Next, RUMBLE performs timing analysis so as to measure timing improvement later. Line 3 stores the state of the circuit (gates and nets) so as to possibly undo most recent transformations we are considering. Once the initial state is safely stored, lines 4–6 use the linear program of Sect. 3.4 to compute new gate locations, followed by buffer removal. If the model shows improvement we continue. Buffers are inserted on line 8, and other physical synthesis optimizations could also be applied here (e.g, repowering, Vth assignment, etc.). Lines 9–12 measure improvement, and in the case of timing degradation, restores the initial solution.

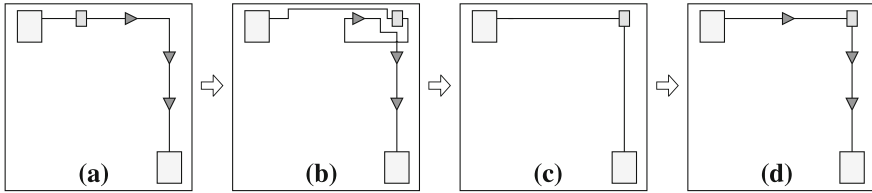


Fig. 3.12 The RUMBLE algorithm proceeds by **a** selecting a subcircuit to work on. An LP is formulated and solved, with movable gates being relocated as shown in **b**. Existing repeater trees are no longer appropriate, and are subsequently removed in **c**. Finally, the nets are re-buffered, forming the final subcircuit shown in **d**

Fig. 3.13 The RUMBLE algorithm for moving one latch

```
RUMBLE-ONE-LATCH
      ▷ Input: Gate movable
      ▷ Output: movable has optimized location and interconnect
 1    subcircuit = BUILD-SUBCIRCUIT-FROM-SEED(movable, 0)
 2    before-timing = MEASURE-TIMING(subcircuit)
 3    initial-solution = CACHE-SUBCIRCUIT(subcircuit)
 4    LP = new RUMBLE linear program for subcircuit
 5    after-locs = SOLVE(LP)
 6    SET-GATE-LOCATIONS(subcircuit, after-locs)
 7    REMOVE-BUFFERS(subcircuit)
 8    REINSERT-BUFFERS(subcircuit)
 9    after-timing = MEASURE-TIMING(subcircuit)
10    if(after-timing worse than before-timing)
11        RESTORE-GATE-LOCATIONS(subcircuit, initial-solution)
12        RESTORE-INTERCONNECT(initial-solution)
```

3.6 Empirical Validation

RUMBLE is implemented in C++ (compiled with GCC 4.1.0) and integrated into an industrial physical synthesis flow. For our experiments, we examined an already optimized 130 nm commercial ASIC with clock period 2.2 ns and 3 million objects. We first examined the most critical latches and then filtered out the ones where the latch was already well placed. We use the algorithm from [21] to perform buffering after the cells have been moved. In practice, the LP-solving technique from RUMBLE requires only 17 ms; the buffering algorithm dominates the runtime (over 75 %). Since the overall runtime is dependent on the choice of the buffering algorithm we omit the (trivial) runtimes from our tables. Note that the *do-no-harm approach* of Sect. 3.5 is applied to all experiments, preventing timing degradation in our tables (i.e., a value of 0 appears in the imprv. column).

Re-Buffering in RUMBLE

Previously published LP techniques for timing-driven placement do not allow for re-buffering during optimization. Instead, they are either applied before buffers have

Table 3.1 Keeping buffers instead of reinserting them degrades RUMBLE's performance

	Implications of keeping buffers					
	KEEP-BUFFERS Slack (ps)			RUMBLE Slack (ps)		
Subcircuit	Orig	New	Imprv.	Orig	New	Imprv.
latch A0	−1480	−1318	162	−1480	26	1506
latch A1	−1268	−1066	202	−1268	186	1454
latch A2	−1020	−939	80	−1020	−791	229
latch A3	−953	−766	187	−953	−390	563
latch A4	−897	−677	220	−897	356	1253
latch A5	−848	−746	101	−848	−278	570
latch A6	−690	−690	0	−690	395	1085
latch A7	−645	−586	59	−645	−19	626
latch A8	−633	−560	74	−633	290	923
latch A9	−610	−466	144	−610	262	872
avg	−904	−782	123	−904	4	908

been inserted, or they do not differentiate the buffers from other gates. Our first experiment is designed to show how important it is to rip up buffers before replacing gates and subsequently rebuffering.

We modified our pseudocode in Fig. 3.11 so that the function IS-BUFFER() always returns false. The effect of this is to stop *seeing through* the buffers, and instead to consider them fixed timing endpoints. This configuration models the work of [13]. We then calculate a new location for each latch with the LP in Sect. 3.4. The final change is to skip line 8 of Fig. 3.13, i.e., do not re-buffer. We call this algorithm KEEP-BUFFERS.

Table 3.1 shows the results of RUMBLE on a single latch compared with KEEP-BUFFERS. Column 1 shows the name of the benchmark and columns 2 and 5 show worst-slacks in picoseconds before optimization. Columns 3 and 6 show the slacks after optimization of KEEP-BUFFERS and RUMBLE respectively. Columns 4 and 7 show the improvements of each technique.

From the table we observe the following:

- Despite preserving buffers, KEEP-BUFFERS is still able to improve solution quality for nine out of ten testcases, though the improvement is never more than 220 ps.
- When re-buffering is allowed, RUMBLE is able to significantly outperform KEEP-BUFFERS for all ten testcases. On average the improvement is 7.4× greater.
- While KEEP-BUFFERS improves slack by an average of 123 ps, RUMBLE improves slack by 908 ps, which confirms how important it is to rip-up buffers so that they do not anchor the latch into an artificially small region.

Table 3.2 The RUMBLE model accurately predicts the solution quality improvements in the reference timing model

	Model timing versus reference timing					
	Model slack (ps)			Subcircuit slack (ps)		
Subcircuit	Orig	New	Imprv.	Orig	New	Imprv.
`latch A0`	−1799	−48	1751	−1480	26	1506
`latch A1`	−1509	65	1574	−1268	186	1454
`latch A2`	−1113	−868	245	−1020	−791	229
`latch A3`	−1147	−527	620	−953	−390	563
`latch A4`	−1090	180	1269	−897	356	1253
`latch A5`	−945	−295	650	−848	−278	570
`latch A6`	−920	320	1241	−690	395	1085
`latch A7`	−886	49	935	−645	−19	626
`latch A8`	−913	213	1126	−633	290	923
`latch A9`	−800	397	1198	−610	262	872
avg	−1112	−51	1061	−904	4	908

Accuracy of the RUMBLE Timing Model

Theoretical results published by Otten [17] and discussed in Sect. 3.3 indicate that optimal buffer insertion on a two-pin net results in a wire delay that is linearly-proportional to its length. The RUMBLE model heavily relies on these results.

Table 3.2 compares the model-predicted values for subcircuit slack to values measured by running a commercial static timing analyzer. Measurements are taken after the RUMBLE LP is solved, the latches are moved and connected nets are buffered. Columns 2–4 report the initial, final, and improvement in worst-slack of the subcircuit measured by the timing model presented in Sect. 3.3. Columns 5–7 report the same metrics measured by the STA engine.

We make the following observations:

- On average, the RUMBLE model overestimates the actual timing improvement by about 15 %. This makes sense since it assumes an optimal ideal buffering will be achievable, but this is not always the case, especially for multi-sink nets.
- However, if one compares actual improvement to model improvement, there is a 97 % correlation, suggesting that the model is reasonable enough to justify the latch location.

We now show how RUMBLE actually improves the design's timing characteristics.

RUMBLE on a Single Latch

Given that we are solving a new physical synthesis problem, existing solutions are scarce. Therefore we first consider straightforward approaches to solve this problem. One possibility is to take the *center-of-gravity* (COG) of adjacent pins. A timing-

Table 3.3 Comparison of RUMBLE's LP to a slack-weighted center-of-gravity technique

Center-of-gravity versus RUMBLE						
	COG Slack (ps)			RUMBLE Slack (ps)		
Subcircuit	Orig	New	Imprv.	Orig	New	Imprv.
latch A0	−1480	−527	953	−1480	26	1506
latch A1	−1268	−203	1065	−1268	186	1454
latch A2	−1020	−800	219	−1020	−791	229
latch A3	−953	−615	338	−953	−390	563
latch A4	−897	−78	819	−897	356	1253
latch A5	−848	−319	529	−848	−278	570
latch A6	−690	−690	0	−690	395	1085
latch A7	−645	−645	0	−645	−19	626
latch A8	−633	−633	0	−633	290	923
latch A9	−610	67	677	−610	262	872
avg	−904	−444	460	−904	4	908

driven improvement of the center-of-gravity technique weights each pin by its slack. A reasonable version of this heuristic works in the following way. For a slack threshold T_s (see Sect. 3.4), let the weight w of a pin p with slack S_p be:

$$w_p = \begin{cases} 1 + |S_p - T_s| & S_p < 0 \\ \max(0.1, 1 - |S_p - T_s|) & S_p \geq 0 \end{cases} \tag{3.19}$$

Then we compute the x coordinate of movable gate m as the weighted average of the x coordinates of the set of neighboring pins P.

$$m_x = \frac{\sum_{p \in P} w_p p_x}{\sum_{p \in P} w_p} \tag{3.20}$$

and similarly for the y coordinate.

We implemented the above COG technique within the RUMBLE framework in place of the LP solver presented in Sect. 3.4. We still allow COG the benefits of ripping up buffers, and reinserting them after the latches are moved. Table 3.3 shows a comparison between RUMBLE and slack-weighted COG on 10 latches. Column 1 shows the same latches as reported in Table 3.2. Columns 2–4 show the initial and final slacks, and improvement for COG. Columns 5–7 show the same for RUMBLE.

We observe the following:

- For all ten cases, RUMBLE generates a better solution than COG. For three of the cases, COG could not improve the latch placement. These new solutions are rejected by the controller so as not to make the design worse.

- On average, COG improves slack by 20.9 % of the 2.2 ns cycle time, whereas RUMBLE improves slack by 41.3 %. This shows that one must incorporate slack constraints on cells incident on the latch to achieve the most balanced solution.

Optimizing Multiple Gates Simultaneously

For this experiment, we show how an even better solution can be obtained when one allows cells close to the latch to move. We show the effectiveness of this technique on two sets of circuits.

- **One-hop** subcircuits include every gate (while ignoring buffers and inverters) incident to the latch of interest that shares an incident net with the latch. Typically this results in 4 or 5 gates being moved.
- **Two-hop** subcircuits in addition include all non-buffer and inverter cells incident to cells in the one-hop neighborhood. These subcircuits range from 11 to 18 movables with a mean of 14.8 movables.

We compare this technique to iterated single-move RUMBLE, where we pick each cell in the neighborhood and solve the LP for that particular cell, fix it, and then move to the next cell. The experiment is designed to show that multiple cells need to be optimized simultaneously to obtain the best results.

To measure the improvement one must now consider the slacks of all cells that may be moved, and the objective becomes to improve the worst slack of the entire subcircuit. However, when one cannot improve the most critical path, the other paths may have room for improvement. We use TTS to measure the total improvement of all the slacks in the subcircuit.

Tables 3.4 and 3.5 compare iterating RUMBLE over each gate one at a time versus RUMBLE moving multiple gates simultaneously. Columns 2–4 show the original and final slack, and the slack improvement for iterated single-move RUMBLE, while columns 5–7 show the corresponding TTS measurements for a zero-slack threshold. Columns 8–13 show the same measurements for multi-move RUMBLE. We make the following observations:

- Multi-move RUMBLE is clearly more effective than iterative RUMBLE both for one- and two-hop neighborhoods. In fact, for six out of ten one-hop subcircuits and for seven out of ten two-hop circuits, multi-move actually brought the TTS down to zero, meaning it fixed all the timing violations. Iterative single move was able to fix two and four respectively.
- On average, the worst-slack improvements were 849 and 673 ps respectively for one- and two-hop subcircuits. The diminished improvement for larger subcircuits is likely because we are including more nets, some of which cannot be improved as much as those connected to the imbalanced latch (Fig. 3.9 has an example).
- Solving the LP takes 53 ms for one-hop subcircuits and 325 ms for two-hop subcircuits, on average.

Table 3.4 RUMBLE simultaneously moving a *one-hop* neighborhood compared to iteratively moving the same gates individually

Iterated RUMBLE versus RUMBLE: 1-hop												
	Iterated single-move RUMBLE						Multi-move RUMBLE					
	Slack (ps)			TTS (ps)			Slack (ps)			TTS (ps)		
Subcircuit	Orig	New	Imp.	Orig	New	Imp.	Orig	New	Imp.	Orig	New	Imp.
subckt B0	−1542	−1542	0	−6091	−6091	0	−1542	−130	1412	−6091	−130	5962
subckt B1	−1501	−277	1223	−5924	−277	5647	−1501	55	1556	−5924	0	5924
subckt B2	−1240	−1240	0	−4354	−4354	0	−1240	−980	261	−4354	−4044	310
subckt B3	−848	−278	569	−2523	−812	1710	−848	−279	569	−2523	−813	1709
subckt B4	−690	−79	612	−4090	−79	4011	−690	202	893	−4090	0	4090
subckt B5	−690	48	739	−2053	0	2053	−690	290	980	−2053	0	2053
subckt B6	−645	−18	627	−1921	−32	1889	−645	301	945	−1921	0	1921
subckt B7	−595	86	681	−1937	0	1937	−595	503	1098	−1937	0	1937
subckt B8	−444	−444	0	−889	−889	0	−444	−92	352	−889	−191	698
subckt B9	−418	−46	372	−857	−46	811	−418	6	424	−857	0	857
avg	−861	−379	482	−3064	−1258	1806	−861	−12	849	−3064	−518	2546

Table 3.5 RUMBLE simultaneously moving a *two-hop* neighborhood compared to iteratively moving the same gates individually

Iterated RUMBLE versus RUMBLE: 2-hop												
	Iterated single-move RUMBLE						Multi-move RUMBLE					
	Slack (ps)			TTS (ps)			Slack (ps)			TTS (ps)		
Subcircuit	Orig	New	Imp.	Orig	New	Imp.	Orig	New	Imp.	Orig	New	Imp.
subckt C0	−719	−719	0	−8313	−8313	0	−719	−675	44	−8313	−5028	3285
subckt C1	−719	−719	0	−8004	−8004	0	−719	−653	66	−8004	−4386	3617
subckt C2	−690	−79	612	−4090	−79	4011	−690	314	1004	−4090	0	4090
subckt C3	−690	−79	612	−4090	−79	4011	−690	337	1027	−4090	0	4090
subckt C4	−681	−349	333	−3865	−349	3516	−681	−158	524	−3865	−158	3707
subckt C5	−645	−91	554	−3767	−306	3462	−645	371	1015	−3767	0	3767
subckt C6	−645	−33	612	−3767	−52	3716	−645	324	969	−3767	0	3767
subckt C7	−318	−318	0	−940	−940	0	−318	531	848	−940	0	940
subckt C8	−490	227	716	−966	0	966	−490	466	956	−966	0	966
subckt C9	−217	−217	0	−652	−652	0	−217	60	277	−652	0	652
avg	−581	−238	344	−3846	−1877	1968	−581	92	673	−3846	−957	2888

RUMBLE in a Physical Design Flow

In the experiments presented so far, we have compared the effects of RUMBLE to those of other techniques on the most critical latches of the design. Due to the high runtime of buffering all of the nets in multi-move subcircuits, multi-move RUMBLE for every critical latch in the design is expensive. Consequently, in this subsection, we demonstrate the cumulative effect of single-move RUMBLE when deployed in our physical synthesis flow on *all latches with a critical pin*. Table 3.6 shows two

Table 3.6 RUMBLE deployed in a physical design flow on circuits that have pipeline latch placement problems. ckt1 has 2.92M objects and 629k latches and ckt2 has 4.74M objects and 247k latches. "old" reports values before RUMBLE "new" reports results after and "diff" reports their difference. FOM is reported in nanoseconds

		Imb.	Imb. FOM	Crit.	Crit. FOM	TTS
	old	102768	−21855	7912	−2798	−22448
ckt1	new	93736	−19400	7775	−2644	−20511
	diff	−9032	2455	−137	154	1937
	old	12151	−3080	3206	−1783	−19211
ckt2	new	11037	−2351	2997	−1667	−18170
	diff	−1114	271	−209	116	1041

circuits that each contain a significant number of poorly placed latches. For each circuit, we report 5 statistics. An imbalanced latch is defined as one that has slack on the input pins that is greater than the slack threshold, T_s (see Sect. 3.4), and slack on the output pins that are lower than T_s, or vice versa. The Imb. column reports the number of imbalanced latches found in the design. Let the set of imbalanced latches be I, and for each latch l let $ws(l)$ be the worst slack of any pin on l. We define imbalance FOM to be

$$\sum_{l \in I} T_s - ws(l) \tag{3.21}$$

The Imb. FOM column reports this value. A critical latch is defined as one that has pins on both sides that are below T_s. The Crit. column reports the number of critical latches found in the design. Similarly to imbalance FOM, if C is the set of critical latches and for each latch c let $ws(c)$ be the worst slack of any pin on c, then we define the critical FOM to be

$$\sum_{c \in C} T_s - ws(c) \tag{3.22}$$

The Crit. FOM column reports this value.

Finally, the TTS column reports the TTS for the entire design. We make the following observations:

- RUMBLE reduces the number of imbalanced latches by 8.8 and 9.2 % on ckt1 and ckt2, respectively.
- RUMBLE has a harder time optimizing critical latches than imbalanced ones.
- RUMBLE reduces circuit TTS by 8.6 and 5.4 % on ckt1 and ckt2, respectively.
- RUMBLE improves the characteristics of all columns, and does no harm to the circuit metrics.

In addition to these observations, we point out that the two most common reasons for being unable to fix a particular latch are (1) there is a high-fanout net in the subcircuit, which would degrade the performance of buffering, and we therefore skip this case or (2) the gates are moved to a fixed endpoint, which indicates that RUMBLE does

not have enough freedom to solve the problem entirely. The addition of RUMBLE to our design flow adds about 4 % to the total runtime in these experiments.

3.7 Conclusions

In this work we observe that wirelength-driven placement leads to particularly poor timing of *pipeline latches* in modern physical design flows, which is especially problematic at sub-130 nm technology nodes. To address this challenge, we developed RUMBLE–a linear-programming based, incremental physical synthesis algorithm that incorporates timing-driven placement and buffering. The latter justifies RUMBLE's linear-delay model which exhibited a 97 % correlation to the reference timing model in our experiments. Empirically this delay model is accurate enough to guide optimization; RUMBLE improves slack by 41.3 % of cycle time on average for a large commercial ASIC design.

The linear program (LP) used in RUMBLE is general enough to optimize multiple gates and latches simultaneously. However, when moving multiple gates considering only the slack objective, we encountered two challenges: placement stability and TTS degradations. We present our extensions to address these problems directly in our LP objective. With these additions, moving several gates simultaneously improves upon RUMBLE used iteratively on the same movables.

References

1. International Technology Roadmap for Semiconductors, 2001 edition. http://public.itrs.net
2. Cocchini P (2002) Concurrent flip-flop and repeater insertion for high performance integrated circuits. ICCAD 2002:268–273
3. Saxena P, Menezes N, Cocchini P, Kirkpatrick DA (2004) Repeater scaling and its impact on CAD. IEEE Trans CAD 23(4):451–463
4. Halpin B, Chen CYR, Sehgal N (2001) Timing driven placement using physical net constraints. DAC 2001:780–783
5. Jackson MAB, Kuh ES (1989) Performance-driven placement of cell based IC's. DAC 1989:370–375
6. Kahng AB, Markov IL (2002) Min-max placement for large-scale timing optimization. ISPD 2002:143–148
7. Kong TT (2002) A novel net weighting algorithm for timing-driven placement. ICCAD 2002:172–176
8. Marek-Sadowska M, Lin SP (1989) Timing driven placement. ICCAD 1989:94–97
9. Ou SL, Pedram M (2000) Timing-driven placement based on partitioning with dynamic cut-net control. DAC 2000:472–476
10. Trevillyan L et al (2004) An integrated environment for technology closure of deep-submicron IC designs. IEEE Design Test Comput 21(1):14–22
11. Chang K-H, Markov IL, Bertacco V (2007) Safe delay optimization for physical synthesis. ASP-DAC 2007:628–633
12. Ren H et al (2007) Hippocrates: first-do-no-harm detailed placement. ASP-DAC 2007:141–146

13. Wang Q, Lillis J, Sanyal S (2005) An LP-based methodology for improved timing-driven placement. ASP-DAC 2005:1139–1143
14. Chowdhary A et al (2005) How accurately can we model timing in a placement engine? DAC 2005:801–806
15. Alpert CJ et al (2006) Accurate estimation of global buffer delay within a floorplan. IEEE Trans TCAD 25(6):1140–1146
16. Cong J, He L, Koh C-K, Madden PH (1996) Performance optimization of VLSI interconnect layout. Integration VLSI J 21:1–94
17. Otten R (1998) Global wires harmful? ISPD 1998:104–109
18. Nair R, Berman C, Hauge P, Yoffa E (1989) Generation of performance constraints for layout. TCAD 8(8):860–874
19. Sapatnekar S (2004) Timing. Springer, New York
20. Luo T, Newmark D, Pan DZ (2006) A new LP based incremental timing driven placement for high performance designs. DAC 2006:1115–1120
21. Alpert CJ et al (2004) Fast and flexible buffer trees that navigate the physical layout environment. DAC 2004: 24–29

Chapter 4
Bounded Transactional Timing Analysis

Modern physical synthesis flows operate on very large designs and perform increasingly aggressive timing optimizations. Traditional incremental timing analysis now represents the single greatest bottleneck in such optimizations and is lacking in features necessary to support them efficiently. We describe a paradigm of transactional timing analysis, which, in addition to incremental updates, offers an efficient, nested *undo* functionality that does not require significant timing calculations. This paradigm extends traditional incremental Static timing analysis (STA) and enables necessary infrastructure for multiple physical synthesis optimizations in this book.

Transactions offer significant performance benefits when working with highly-optimized netlists, where most candidate transformations are retracted after evaluation. Another context, where our techniques offer speed-ups of two orders of magnitude, is compound optimizations where incremental updates are amortized over a tree of further possible optimizations. We describe efficient implementations of *update*, *begin*, *commit* and *undo* functionalities by bounding their impact throughout the netlist.

4.1 Introduction

Achieving timing closure for large modern ASIC designs requires the use of *physical synthesis*—a series of performance-driven optimizations that simultaneously alter the layout, the netlist and electrical parameters of logic gates.

Physical synthesis tightly couples *analysis* with *optimization* in an automated flow that iteratively improves design parameters. Such flows rely on *Static Timing Analysis* (STA) in two essential ways. First, STA identifies the sections of the design that are most critical to the overall performance. Second, STA assesses the impact of every potential change on circuit performance, before the change is committed. Circuit optimizations are bundled into *transformations* that implement common operations such as relocating a gate, buffering a net, etc. [1]. Recent state-of-the-art design methodologies consider *compound transformations* to simultaneously perform many

D. A. Papa and I. L. Markov, *Multi-Objective Optimization in Physical Synthesis of Integrated Circuits*, Lecture Notes in Electrical Engineering 166,
DOI: 10.1007/978-1-4614-1356-1_4, © Springer Science+Business Media New York 2013

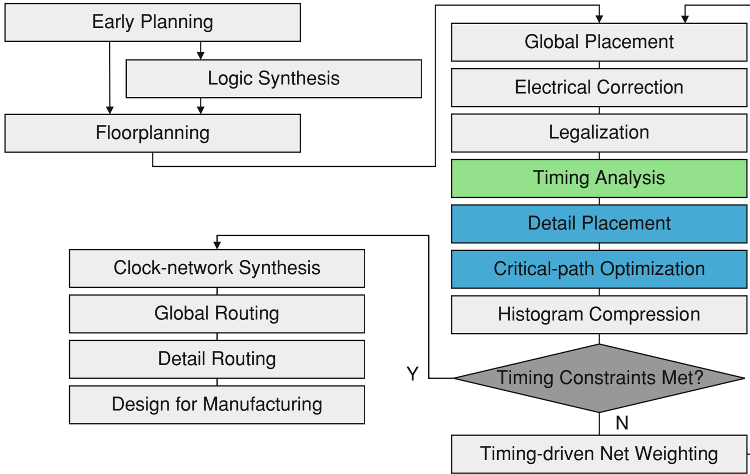


Fig. 4.1 This chapter improves the results of timing analysis as it is used in physical synthesis

simpler transformations that would not have improved overall performance if applied individually [2] (Fig. 4.1).

Advanced technology nodes require complex timing models that cannot be captured analytically with sufficient accuracy, often making timing analysis the single major bottleneck in physical synthesis. Therefore we take a closer look at the conceptual role of STA and its interfaces with optimization. Mathematically, circuit optimizations often interact with STA by obtaining *arrival times* and *required arrival times* at timing points throughout the design [3, 4]. However, running STA on the entire design to evaluate each potential change is impractical. Therefore, STA can be used (i) in *batch mode* to evaluate the compound impact of many changes, (ii) in *incremental mode*, where the impact of a single change is efficiently propagated through the netlist, and (iii) with *lazy updates*, where timing data are propagated only in response to queries, batching the changes that occur between queries.

Multi-objective optimizations now increasingly rely on *do-no-harm* methodologies that carefully evaluate each change and commit only those that provide tangible improvements [5–7]. The more aggressive algorithms have very high rejection rates in this loop, making the speed of incremental STA a major factor in improving physical synthesis. However, batched mode and lazy updates are of limited use when evaluating individual impact of multiple candidate changes.

The major impact of STA on overall runtime tempts physical synthesis developers to assume the responsibility for some aspects of timing analysis and shortchange STA engines for handcrafted local delay models, which offer significant opportunities for runtime improvement. However, this practice risks subtle timing mistakes and also increases the development effort by lowering reuse. Therefore, we propose improvements to reusable STA engines that better account for the bounded scope of physical-synthesis transformations.

We present an extension to the interface of static timing analysis to accommodate transaction histories. Our technique employs a *timing change history* datastructure that stores changes to the timing graph to support an efficient restore operation in the event of a retraction. This approach is specifically designed to allow nesting events that spur timing changes. To further improve worst-case complexity, we limit changes to the timing graph by way of *bounded timing analysis*, an enhancement that works in conjunction with transactional timing analysis to allow for the rapid exploration of circuit search space. Finally, we provide an empirical evaluation of bounded transaction histories for both classical and lazy STA, demonstrating an improvement in performance by up to two orders of magnitude.

The remainder of this chapter is organized as follows. In Sect. 4.2 we describe the state of the art in timing analysis as it applies to physical synthesis and transformation-driven optimization. We go on to classify several types of physical synthesis transformations that pose problems to existing timing analysis engines. Section 4.3 presents bounded transactional timing analysis, along with appropriate details for embedding it into modern static timing analysis. Section 4.4 provides empirical evidence demonstrating that bounded transactional timing analysis greatly improves the speed of transformations that rely on repeated retractions. Conclusions are drawn in Sect. 4.5. A review of static timing analysis appears in the appendix.

4.2 Background

Timing analysis and its integration into the physical design flow have long been key topics in design automation. To this end, we review the basics of STA in this section. Modern static timing engines are products of sophisticated engineering, and have evolved substantially over recent decades. Yet, dramatic changes to basic timing models continue to drive the need for further innovation. For instance, multi-mode timing has become increasingly popular—wherein several timing points are maintained at each node of the global timing graph, each corresponding to a different corner of design operation. While these corners enable modern optimization techniques to evaluate the effect of their actions on many scenarios at once, they also serve as a multiplier of basic computation that the timing engine must perform. Statistical timing engines that reflect the variance of design performance require the maintenance of complex distribution models that also significantly expand the amount of work placed on the timing engine. These elaborate models, in conjunction with a stronger emphasis on local transformation-driven operations, have increased the responsibility of timing engines to provide a much higher degree of incremental maintenance of internal timing state.

Previous Work

The problem of updating only a subset of timing analysis values in response to a local change is explored in [8], where a depth-first propagation of timing values is executed

until no change is observed. This process was later refined [9], to reduce the amount of incremental recalculation needed. A distinction between the propagation cost of positive delay changes and negative delay changes is described in [10], demonstrating that the expense of executing an operation may differ from that of its inverse. The algorithm of [11] avoids excessive computation by propagating only along paths that are influenced by altered inputs. A query language based on temporal logic is proposed in [12], along with an algorithm to efficiently retrieve answers to those queries. Algorithms for incremental timing analysis [13] and incremental criticality updates [14] have been proposed in the context of FPGAs. The authors of [15] explore an extension of static timing analysis to model coupling, and exploit circuit structure to find a good node ordering during incremental iterative analysis.

Relatively little attention has been given to the *explicit* support for the retraction of local design changes. The recent work of [16] provides support for transactional operations such as *begin*, *commit*, and *undo*. However, these operations are restricted only to the resurrection of previously cached *routing* data, and are not communicated to the timing engine. Indeed, the decision to revert one or more timing properties to their original state is typically cast as just another sequence of incremental changes to the system; this forces the wasteful recomputation of timing data, which may be exacerbated when an inverse operation takes much longer to execute than the original operation [10]. Other choices in the design flow—such as the decision to compute Steiner trees for delay estimation—also compound the effort required to restore timing information to a previously known state. The savings, that can be achieved by efficiently rolling back recent changes, are likely to escalate in coming years, as compound transformations become increasingly dominant in physical synthesis and routinely thrash the timer with multiple hypothetical changes.

Incremental Static Timing Analysis

In static timing analysis [17], a *timing graph* $G = (V, E)$ is extracted from a combinational logic circuit. Each vertex $v \in V$ is a timing point, and corresponds to an input or output pin of a gate or a global input or output pin. A pair of vertices, $u, v \in G$, are connected by a directed edge $e(u, v) \in E$ if there is a timing relationship (i.e., a connection) between the pins u and v. This connection can occur within a gate, as in between an input pin and an output pin, or it can correspond to a wire connecting two gates. Each edge has an associated delay $\delta(u, v)$ indicating the delay between u and v.

To determine the worst path of the circuit, a topological traversal is performed on the graph beginning at the sources. The actual arrival time $AAT(v)$ at a timing point v in the circuit is the latest arrival time of any of its predecessors after considering delay:

$$\mathrm{AAT}(v) = \max_{\{u|e(u,v)\}} (\mathrm{AAT}(u) + \delta(u, v)) \tag{4.1}$$

The required arrival time $RAT(u)$ at a timing point u in the circuit is computed in a similar fashion, traversing backwards from the primary outputs of the circuit:

$$\text{RAT}(u) = \min_{\{v|e(u,v)\}} (\text{RAT}(v) - \delta(u, v)) \tag{4.2}$$

A pair of topological traversals are made to determine these values, after which the *slack* of any timing point v is calculated as the difference between required arrival time and actual arrival time:

$$\text{slack}(v) = \text{RAT}(v) - \text{AAT}(v) \tag{4.3}$$

Early STA engines always processed an entire design, which is impractically expensive when evaluating optimization transformations [18]. This expense can be avoided by using stale timing information or crude estimations, neither of which are acceptable in modern high-precision physical synthesis [6]. Another alternative is to maintain accurate timing information throughout the automated flow, but to do so in an incremental fashion. Research in *incremental static timing analysis* aims to provide efficient techniques for the updating of values within a timing network in response to local and partial modifications. Several varieties of incremental STA have appeared over the past decade, and are responsible for decreasing timing runtime from hours to minutes following incremental circuit changes on large ASICs [19].

Further extensions to incremental analysis include *level-limited* and *dominance-limited* schemes to reduce the amount of work performed [20, 21]. *Lazy evaluation* [9, 22], in which propagation is delayed until triggered by a relevant query, represents a particularly important improvement in throughput of static timing analysis engines.

The boost in throughput offered by incremental analysis allows an optimization algorithm (as well as a designer) to explore several hypothetical (or "what-if") scenarios, a task unaffordable in earlier tools [19]. Such hypothetical scenarios are typically communicated to the timing engine as if committing changes. If the results are unacceptable and the scenarios are rejected, another set of changes must be committed. This requires new timing calculations, even though the needed timing values have previously been known. While a single layer of "what-if" support can be added to STA easily, this is insufficient to handle the evaluation of multiple nested scenarios and their retraction. Detailed use cases for retraction are discussed in the following section.

Types of Transformation-Driven Optimization

Recall that timing-driven placement and synthesis seek non-overlapping locations for all cells such that the performance of the design meets objectives [23]. Timing optimization during physical synthesis is typically accomplished by gradually modifying and refining an initial netlist and placement image [24].

Table 4.1 Types of transformations with embedded retraction. Illustrative values in the "Undo frequency" column suggest that some cases require many more retractions that other cases

Type	Undo frequency	Undo purpose
Bind & test	(0.1) Already optimal	Return to initial state
Fallible	(0.1) Upon degradation	Error correction
Candidate	(1.0) For each candidate	Metric-indep. changes
Compound	(10.0) Nested candidates	Joint evaluation

We distinguish *controllers* and optimization *transformations*. A *transformation* applies a particular local optimization to gates and/or nets selected by a *controller*. For instance, IBM's Placement Driven Synthesis (PDS) [1] makes use of several transformation templates, including buffering, re-powering, connection reordering, cloning, etc. A *controller* selects nets and/or gates for optimization, ordering them and judging the impact of optimizations. Both controllers and transformations can query timing engines. For example, transformations often make several queries to STA, not only to construct a basic model of the neighboring region (with appropriate arrival times and required arrival times), but also to verify improvement after optimization is complete. Controllers implement optimization strategies with sophisticated reasoning to handle the feedback received from STA.

Despite extensive support for incremental propagation and lazy evaluation, existing timing engines often perform unnecessary computation in the context of sophisticated optimizations. In this section we illustrate several opportunities for improvement that motivate our research, and summarize them in Table 4.1.

Case 1: Inefficiencies in Fallible Transformations

The simplest transformations first identify feasible changes and then rely on the timer to evaluate the impact on performance. For example, a *cell-movement transformation* trying to relocate a cell on a critical path may identify several vacant nearby locations, and a *repowering transformation* may bind a critical cell to every power level available in the cell library. In either case, a timing query must be independently executed after each change is made, to select the one with greatest slack improvement. We term such methods *bind-and-test* transformations.

More advanced transformations attempt to predict the impact of their changes in advance, so as to quickly weed out unpromising options, then use the timer to select among few finalists, and verify improvement. In the case of a repowering transformation, the slew rate at input pins and the load capacitance offer sufficient information to estimate slack at the output pin for each power level. Such a transformation could guestimate the best power level, bind it, then verify its slack improvement. If the estimate is too inaccurate, the new power level may worsen slack, requiring the change to be rolled back. In other words, such transformations sometimes fail, and we therefore term them *fallible*. Aggressive use of fallible transformations requires *error correction* in the form of an *undo* functionality.

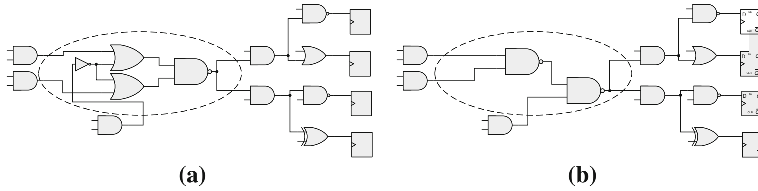


Fig. 4.2 Evaluating the timing impact of the physical synthesis transformation in Fig. 4.3 (output side only). **a** Traditional static timing analysis with lazy evaluation will mark the fanout cone of the change dirty. **b** If the change is found to have a negative impact on timing, it will be reversed. This reversal will be treated as another change, and the fanout cone will be marked dirty for a second time

Though simple, both fallible and bind-and-test transformations are inherently slow because repeated changes and timing queries require laborious propagation and updates of timing information. In our example of bind-and-test repowering, the evaluation of each power level triggers timing updates for the fanin cones (for AATs) and fanout cones (for RATs) of the gate. As we point out next, some of this effort could be deferred and ultimately avoided if the timing engine adopts a philosophy of *lazy evaluation*. Namely, after forward-propagating AATs to evaluate the impact of a change, there is no need to back-propagate RATs, unless the change is *committed*.

A stand-alone reusable STA engine must ensure consistency of its database without necessarily trusting its clients. Therefore, AATs whose values are not current, would be marked as *dirty*. Timing propagation to update dirty data would be invoked *only* in the event of a query (and even then, only to the portion of logic needed to answer the query). However, if the original location (or power level) is optimal—as it is likely to be if estimation routines and detailed placement have done their job properly—the demarcation of these cones as dirty is unnecessary, since the original arrival times stored within these cones are in fact a correct representation of the current state. Figure 4.2 illustrates the work performed by traditional static timing analysis with lazy evaluation when a transformation is applied to a circuit and subsequently retracts its changes.

Case 2: Candidate Selection Transformations

Are those that employ multiple strategies to generate several alternatives, or *candidate solutions*. In doing so, they try each optimization, and select the best candidate, rejecting the rest. Such transformations leverage the fact that different strategies work well in different contexts. For example, consider a transformation that generates candidates by repowering as well as moving a gate. Often, moving a gate has greater impact, but if the design has too little whitespace, there may be no open location where the gate can move to improve timing. Instead, a higher power level may be available for the same footprint, or enough whitespace may be available nearby to increase the footprint.

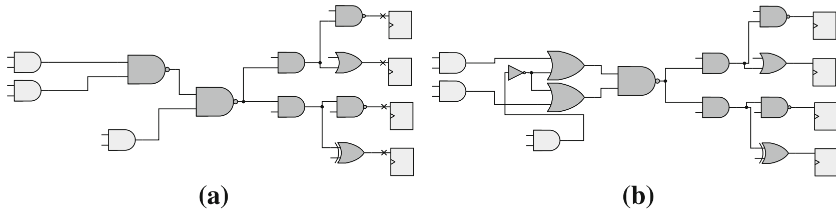


Fig. 4.3 A physical synthesis transformation improves the subcircuit in **a** by resynthesizing the logic, resulting in the circuit shown in **b**. The traditional way of evaluating the timing impact of such transformations can be improved considerably

While fallible transformations may occasionally invoke undo for *correction* (e.g., when they degrade circuit performance due to approximation inaccuracy), a candidate selection transformation requires undo *by construction*—after each candidate is computed, the initial state must be restored so that the next candidate can be generated *independently* based on the initial conditions. In the example of repowering or moving a gate, retraction must restore the gate to its original power level after repowering so that the movement decision can be based on the timing of the initial power level. Timing queries for interrogating initial conditions of each candidate generation strategy can avoid the unnecessary work of timing updates if undo can restore the initial timing state.

Case 3: Compound Transformations

Not only consider multiple strategies for generating candidates, but also do so for *multiple objects*. Such transformations may even consider *composing* overlapping optimizations to generate a single candidate. For example, consider simultaneously moving and/or repowering two connected gates in a discrete domain [2, 5]. In this situation, a very large number of candidates can be generated and evaluated, where each successive decision may depend on the previous. For example, the resynthesis transformation illustrated in Fig. 4.3 can be thought of as a compound transformation consisting of merging the inverter gates with the OR gate followed by swapping logically equivalent pins.

The increasingly popular compound transformations stress timing analysis tools much more heavily than other use cases, in that the construction of a local model requires the *search* of a large, conditional solution space. Modifications are typically made in *nested pairs* to generate appropriate timing arcs; indeed, the authors of [2] observe that the expense of generating their disjunctive timing graph is often more costly than the branch-and-bound search used to solve it optimally, a consequence of the propagation efforts of the timer. When *undo* can efficiently restore the previous timing state, *combinatorially many* timing updates can be saved in compound transformations.

4.3 Transactional Timing Analysis

In the presence of retractions, the state-of-the-art STA engines perform a large amount of unnecessary work, as we have demonstrated in Sect. 4.2. In this section, we present the details of *bounded transactional timing analysis*, which serves to substantially reduce the computation needed to support *undo* functionality. We first consider its application to classical STA, and then to the more advanced version that supports lazy evaluation.

Support for Transactions

By definition, a retraction restores the design to a previously known state. Current techniques (which view retraction as a separate incremental change) discard the original timing values during propagation. In contrast, transactional timing analysis *caches* timing data that becomes invalidated during the execution of a change.

Specifically, when a modification is made to the design, the timer is notified through a monitoring mechanism that the delay at a particular timing point has changed. That notification triggers a corresponding propagation to the transitive fanin and fanout cones. During transactional timing analysis propagation, prior values are not simply overwritten (as is commonly done within STA engines), but are rather stored in a change stack as new values are written in their place. Therefore, if and when change is retracted, the old values may be restored by "replaying" the timing updates in reverse.

In the case that a sequence of nested transactions are executed (as may occur with compound transformations), each individual change stack serves as a distinct checkpoint of the design state. These checkpoints are themselves stored on a transaction stack of all change stacks. A new change stack is pushed onto the transaction stack when a transformation requests a new checkpoint. The current state of timing is stored in the timing graph as usual. When a transformation backtracks and retracts its circuit modifications, changes to the timing graph may be rolled back to the most recent checkpoint by copying all values in the current head of the transaction stack back into the timing graph. Changes may be committed simply by clearing the transaction stack.

Figure 4.4 illustrates one possible implementation of transactional timing analysis. Several variations of this code are useful in different circumstances. For instance, if a change is likely to have significant impact on the state of the design, the caching of old timing values could be performed once, prior to rather than during propagation.

Integrating these ideas into a high-performance timing engine requires a sophisticated interface for optimization transformations. In particular, transformations are required to communicate their intent, e.g., whether a change request is truly new or seeks to restore a previous state. This information allows the timer to take appropriate actions on behalf of each transformation for each change.

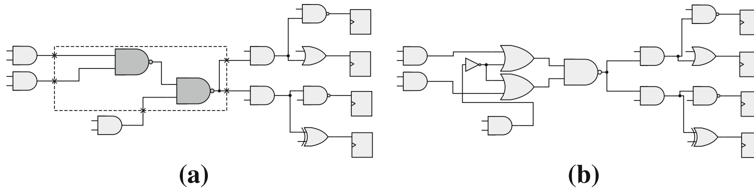


Fig. 4.4 One possible implementation of transactional timing analysis. The functions PROPAGATE-FORWARD and PROPAGATE- BACKWARD shown here using recursion for brevity are best implemented without recursion

Ensuring Consistency and Compatibility

As noted earlier, it is common for static timing engines to defer timing updates until needed by a relevant timing query. In many cases, this avoids work when timing values are invalidated multiple times before they are actually used. The notification of a change in delay during such *lazy execution* will not trigger timing propagation; instead, the fanin and fanout cones of a modified edge are simply marked *dirty*, indicating that they must be recomputed.

To accommodate transactional timing analysis with lazy execution, dirty bits must also be considered as part of the state of a timing point. Whereas traditional engines will leave nodes marked as dirty in the event of a retraction, bounding timing analysis will revert them back to their state prior to the change. Though not shown in Fig. 4.4, this extension is relatively straightforward: all actions that alter the dirty bit of a timing point are recorded, and are subsequently restored if a retraction is issued by the transformation.

Finally, support for transaction histories in the presence of logic changes (such as in our example) requires the careful caching of topological modifications to the graph itself (in addition to the timing values associated with these elements). The creation, deletion, and modification of graph connectivity can be achieved though a reference labeling of timing points; changes to structural elements, such as edges and nodes, are recorded with respect to these unique identifiers, and thus may subsequently be restored. While the implementation required to properly maintain this bookkeeping is complex and nuanced, the basic framework we outlined so far encounters no substantial obstacles.

Bounded Timing Windows

When evaluating the impact of a transformation, it is common to query timing at specific relative locations to the change. For example, one can query the slack of the output pin of a gate after repowering, or the slack of an input pin at the next circuit level after moving a gate. When possible query points can be limited to a *window of interest* known in advance, one can reduce the maintenance requirements for timing information and the update effort. This window may be expanded slightly for safety,

and we call the resulting local region a *bounded timing window*.[1] Limiting propagation to such windows provides cost savings, as it is only necessary to propagate arrival times (and/or dirty bits, in the case of lazy evaluation) to the boundaries of the window. Likewise, in the event of a rollback, the data required to restore the graph to its original state is also reduced. Since immediate timing queries are assumed to be made within the timing window, all values outside the region are considered to be fixed timing endpoints. Bounded transactional timing analysis of an example transformation is illustrated by Fig. 4.5.

Selecting an appropriate window size for a particular transformation may require some care. The effect on timing of an optimization depends on the nature of the optimization; therefore, choosing a static window size is best done when the transformation is designed and tested. In particular, differences in slew rate can greatly affect timing for the whole path in ways that are difficult to predict while only considering slack [25]. For this reason, timing analysis tools support a mode to limit slew rate propagation to a constant number of levels. This mode provides a convenient way to limit the scope of timing changes and improves the speed of timing analysis in physical synthesis tools. Any window larger than the scope of slew rate propagation can provide faster queries with no accuracy loss. Furthermore, in the context of bounded transactional timing analysis, timing queries are only required to decide if a retraction is necessary. Typically, the effect of an optimization on the timing of a path is known with enough accuracy to make a decision to retract or not after a signal is propagated through only a few levels of logic. An additional dynamic approach runs a small number of trial transformation applications and samples several window sizes to determine how much accuracy is lost for various window sizes. It then chooses the smallest window size with tolerable error to be used on the majority of transformation applications.

Facilitating Parallelism

Since a bounded timing window delimits the scope of a local change, it also provides a guarantee of the mutual independence of disjoint timing islands. This independence meets the requirements set forth for *distributed* static timing analysis [26, 27] and could, in theory, be exploited to easily decompose timing optimization into several parallel processes. Although we do not evaluate such a parallel architecture in this work, we emphasize that significant runtime savings could be gained if these techniques are integrated with other components of the design flow, e.g., the placement engine, the data model, etc.

[1] Some static timing engines—such as IBM's EinsTimer^{TM}—provide similar level-limiting features that serve to circumscribe the scope of local changes; they are not, however, integrated with any form of transaction management.

```
CHANGED-DELAY
   ▷ Input: arc − > timing arc that changed
1  PROPAGATE-FORWARD(arc.input)
2  PROPAGATE-BACKWARD(arc.output)
```

```
COMMIT-CHANGES
   ▷ Existing changes no longer need to be tracked
1  ChangeHistory.clear()
```

```
UNDO-CHANGES
1  AATStack = ChangeHistory.top().AATStack
2  while not AATStack.empty()
3     AATStack.top().node.aat =
                         AATStack.top().aat
4     AATStack.pop()
5  RATStack = ChangeHistory.top().RATStack
6  while not RATStack.empty()
7     RATStack.top().node.aat =
                         RATStack.top().aat
8     RATStack.pop()
9  ChangeHistory.pop()
```

```
PUSH-CHANGES
1  ChangeHistory.push()
```

```
PROPAGATE-FORWARD
   ▷ Input: timing-point
1  foreach successor succ of timing-point
2     if UPDATE-AAT(timing-point, succ)
3           PROPAGATE-FORWARD(succ)
```

```
PROPAGATE-BACKWARD
   ▷ Input: timing-point
1  foreach predecessor pred of timing-point
2     if UPDATE-RAT(pred, timing-point)
3           PROPAGATE-BACKWARD(pred)
```

```
UPDATE-AAT
   ▷ Input: pred, succ timing points
1  delay = COMPUTE-DELAY(pred, succ)
2  if (succ.aat < pred.aat + delay)
3     ChangeHistory.top().AATStack.push(TC(succ, succ.aat))
4     succ.aat = pred.aat + delay
5     return True
6  return False
```

```
UPDATE-RAT
   ▷ Input: pred, succ timing points
1  delay = COMPUTE-DELAY(pred, succ)
2  if (pred.rat > succ.rat - delay)
3     ChangeHistory.top().RATStack.push(TC(pred, pred.rat))
4     pred.rat = succ.rat - delay
5     return True
6  return False
```

Fig. 4.5 Evaluating the timing impact of the physical synthesis transformation in Fig. 4.3 (output side only). **a** Bounded transactional timing analysis will not propagate the change outside of a specified window. **b** In the event of a reversion, gates with dirty timing will have their timing data restored

Complexity Analysis

Let $\mathscr{C}$ denote a fanout cone affected by a given logic transformation, and let $\mathscr{W}$ represent the bounded timing window used in bounded transactional timing analysis for that change. In traditional incremental static timing analysis with lazy evaluation, all of the timing points in $\mathscr{C}$ are marked dirty upon the change. If the change is

Table 4.2 Performance of bounded transactional timing analysis, with and without lazy evaluation

$\mathscr{P}$[undo] (%)	Window Size	Nodes expanded				Runtime (seconds)					
		Classical STA		Lazy STA		Classical STA			Lazy STA		
		w/o tr	w/ tr	w/o tr	w/ tr	w/o tr	w/ tr		w/o tr	w/ tr	
0	∞	12638	12638	22905	22905	0.28	0.33	(0.8×)	2.09	2.36	(0.8×)
	40	12638	12638	19356	19356	0.32	0.33	(0.9×)	1.65	1.86	(0.8×)
	20	10186	10186	7400	7400	0.25	0.26	(0.9×)	0.41	0.47	(0.8×)
	10	3641	3641	1967	1967	0.08	0.08	(1.0×)	0.1	0.11	(0.9×)
10	∞	346170	12646	22895	22605	14.5	0.32	(45.3×)	2.07	2.29	(0.9×)
	40	202821	12646	19346	19056	6.65	0.32	(20.7×)	1.69	1.82	(0.9×)
	20	41251	10194	7380	7013	1.3	0.25	(5.2×)	0.41	0.43	(0.9×)
	10	5957	3649	1955	1793	0.14	0.07	(2.0×)	0.09	0.1	(0.9×)
30	∞	1124067	12693	22888	21960	46.66	0.32	(145.8×)	1.98	2.28	(0.8×)
	40	510642	12693	19339	18320	14.84	0.32	(46.3×)	1.63	1.76	(0.9×)
	20	75128	10233	7353	6282	2.13	0.25	(8.5×)	0.39	0.37	(1.0×)
	10	8716	3649	1948	1599	0.19	0.07	(2.7×)	0.1	0.09	(1.1×)
50	∞	1733287	12693	22886	9939	73.11	0.32	(228.4×)	2	0.67	(2.9×)
	40	799207	12693	19335	9405	24.5	0.32	(76.5×)	1.62	0.63	(2.5×)
	20	105003	10233	7351	4012	3.12	0.25	(12.4×)	0.41	0.25	(1.6×)
	10	11570	3649	1944	1085	0.26	0.08	(3.2×)	0.09	0.06	(1.5×)
70	∞	1855924	12705	22872	6483	76.47	0.31	(246.6×)	2.02	0.48	(4.2×)
	40	913461	12705	19321	5848	27.52	0.32	(86.0×)	1.65	0.44	(3.7×)
	20	133800	10245	7339	1882	4.12	0.25	(16.4×)	0.4	0.14	(2.8×)
	10	15257	3661	1932	397	0.34	0.07	(4.8×)	0.1	0.03	(3.3×)
90	∞	1947548	12705	22850	5551	76.81	0.33	(232.7×)	2.11	0.4	(5.2×)
	40	995711	12705	19299	4769	29.02	0.33	(87.9×)	1.62	0.36	(4.5×)
	20	157328	10245	7315	1078	4.51	0.25	(18.0×)	0.4	0.07	(5.7×)
	10	17019	3661	1910	173	0.37	0.07	(5.2×)	0.09	0.02	(4.5×)

retracted, all of the timing points in $\mathscr{C}$ are again marked dirty. Subsequent queries in the area may need to recompute previously known timing data for those timing points that are left dirty. When using bounded transactional timing analysis, $|\mathscr{C} \cap \mathscr{W}|$ nodes are marked dirty upon a change. If the change is retracted, no more timing points are recomputed, but $|\mathscr{C} \cap \mathscr{W}|$ timing points are copied back into the timing graph. No timing points are left dirty.

We use the following notation to estimate the impact of proposed techniques. Let L be the depth of a fanout cone $\mathscr{C}$. Let $L_{\mathscr{W}}$ denote the depth of $\mathscr{C}$ in the window $\mathscr{W}$. Let B be the average branching factor of $\mathscr{C}$ and let R be the average reconvergence factor. Then $|\mathscr{C}|$ is approximately $(B - R)^L$. The size of the fanout cone within the window $|\mathscr{C} \cap \mathscr{W}|$ is approximately $(B-R)^{L_{\mathscr{W}}}$. Therefore, the number of timing points that do not need to be marked dirty due to bounding is approximately $(B - R)^L - (B - R)^{L_{\mathscr{W}}}$. The number of timing points restored upon a retraction when using bounded transactional timing analysis is approximately $(B - R)^{L_{\mathscr{W}}}$,

versus approximately $(B - R)^L$ timing points left dirty in traditional incremental static timing analysis.

4.4 Empirical Validation

In order to evaluate the computational benefits of bounded transactional timing analysis, we have implemented the aforementioned techniques in a new static timing analysis tool that supports both classical STA (i.e., the academic variety that immediately performs propagation of modified timing values) and lazy evaluation (e.g., the more popular variety that performs propagation only on demand). For evaluation of the former, we discount the runtime required for initial propagation of a change, as that time is shared by "with-transaction" and "without-transaction" runs. All incarnations of our timing engine employ some form of incremental propagation.

We modified a simple timing-driven gate movement transformation within a state-of-the-art industrial physical synthesis flow to query our static timing analyzer when deciding whether or not to retract the change. Changes to delay values in the timing graph of a real 65nm design were simulated and profiled to determine the runtime incurred by STA. Two parameters were adjusted in these experiments; first, the probability that a delay change is retracted ($P(undo)$), and the size of our bounded timing window (where a size of ∞ indicates the absence of this technique). Since the frequency of finding timing-driven placement improvements strongly depends on the circuit and the state of optimization, our experimental transformation uses the $P(undo)$ parameter to determine when to retract changes. Thus, we can vary $P(undo)$ independently to study the impact on runtime of any retraction frequency.

In experiments, we exercised established physical synthesis transformations that introduced changes in delay values of the timing graph. We then profiled those changes in an STA engine to compare several configurations of timing analysis and measure runtime savings. Two parameters were varied in these experiments:

- $P(undo)$, the probability that a delay change is retracted.
- $L_{\mathscr{W}}$, the number of levels of logic, both upstream and downstream, in the bounded timing window $\mathscr{W}$, where a size of ∞ indicates the absence of this technique.

The results of these tests are presented in Table 4.2. For each setting of $P(undo)$ and $L_{\mathscr{W}}$, we report the number of nodes expanded and runtime incurred by all solver variants. Please note that for evaluation of classical STA, we discount the runtime required for initial propagation of a change, as this time will be incurred by "with-transaction" and "without-transaction" runs.

We observe the following:

- Transactioning is at a slight disadvantage due to the overhead of state-recording. As one would expect, benefit is observed only when retractions are performed. The worst overhead is about 20 % and occurs when using large windows with no chance of undo. In practice, such a transformation should not enable transaction histories.

- For classical STA, a speedup of up to 246× is observed. The greatest speedups occur for the largest windows and greatest probability of undo.
- For lazy evaluation, a speedup of up to 5.2× is achieved. Compared to classical STA without transaction histories, lazy evaluation improves runtime in all configurations that have a non-zero chance of undo. When transaction histories are introduced to both, the runtime improvement of lazy evaluation is reduced. In several cases, classical STA with transaction histories is faster than lazy STA with transaction histories.
- The use of bounded windows dramatically reduces the amount of work, especially when lazy evaluation is disabled. For example, runtime goes from 76.81 to 0.37s for $P(undo) = 90\,\%$.
- For all parameter settings and STA variants, when transaction histories are used, higher frequencies of retraction generally lead to stronger improvements in runtime and nodes expanded.

These results confirm that even with a moderate amount of undo, the computational savings can be substantial. It can also be observed that bounded timing windows (which can be exploited independently of transaction histories) are generally effective at reducing runtime. Indeed, best results are achieved when both transaction histories and bounding approaches are used in concert.

While the use of lazy evaluation alone prevents a fair amount of thrashing (hence its adoption in all modern timing engines), its performance can nevertheless be further improved with these techniques. We expect that most physical synthesis flows will realize the combined benefits of lazy evaluation, transaction histories, and bounded timing windows.

4.5 Conclusions

In this chapter, we have presented *bounded transactional timing analysis*, described our implementation, and validated it in a production physical synthesis flow.

Our work has been motivated primarily by deficiencies in static timing analysis that result in poor runtime for several common physical synthesis operations. Specifically, we have categorized several types of physical synthesis transformations that utilize retraction in different ways. Then we have presented an extension to static timing analysis to accommodate *transaction histories*, in which a history of network delay propagations is tracked and cached so that the state of the timing graph may be efficiently restored in the event of a retraction. This approach was further generalized to allow for the nesting of timing changes. Changes to the timing graph were limited by way of *bounded timing analysis*, an enhancement that works in conjunction with transactional timing analysis to allow for the rapid exploration of circuit search space. The incremental timing concepts presented in this paper are not unique to physical synthesis; they are equally applicable to the efficient support of logic synthesis transformations, and some of them may have been in use for this purpose since the mid

90s. However, conventional logic synthesis does not stress timing infrastructure as much as modern physical synthesis does, therefore relevant techniques were not given as much attention and, to this day, remain poorly documented. We conclude that as transformation-driven optimizations in physical synthesis continue to increase in complexity, the need to efficiently accommodate hypothetical timing queries is likely to grow.

References

1. Trevillyan L et al (2004) An integrated environment for technology closure of deep-submicron IC designs. IEEE Design Test Comput 21(1):14–22
2. Moffitt MD et al (2008) Path smoothing via discrete optimization. DAC 2008, pp 724–727
3. Kannan LN, Suaris PR, Fang H-G (1994) A methodology and algorithms for post-placement delay optimization. DAC 1994, pp 327–332
4. Ren H, Pan DZ, Kung DS (2005) Sensitivity guided net weighting for placement-driven synthesis. IEEE Trans CAD 24(5):711–721
5. Papa DA et al (2008) RUMBLE: an incremental, timing-driven. Physical-synthesis optimization algorithm. ISPD 2008, pp 2–9
6. Ren H et al (2007) Hippocrates: first-do-no-harm detailed placement. ASP-DAC 2007, pp 141–146.
7. Chang K-H, Markov IL, Bertacco V (2007) Safe delay optimization for physical synthesis. ASP-DAC 2007, pp 628–633
8. Drumm AD, Itskin RC, Todd KW (1991) US Patent 5,003,487: method and apparatus for performing timing correction transformations on a technology-independent logic model during logic synthesis, 1991
9. Abato RP, Drumm AD, Hathaway DJ, van Ginneken LPPP (1996) US Patent 5,508,937: incremental timing analysis, 1996
10. Lee J-F, Tang DT (1995) An algorithm for incremental timing analysis. DAC 1995, pp 696–701
11. Sapatnekar SS (1996) Efficient calculation of all-pairs input-to-output delays in synchronous sequential circuits. ISCAS 1996, pp 724–727
12. Mondal A, Chakrabarti PP, Mandal CR (2004) A new approach to timing analysis using event propagation and temporal logic. DATE 2004, pp 1198–1203
13. Singh DP, Manohararajah V, Brown SD (2005) Incremental retiming for FPGA physical synthesis. DAC 2005, pp 433–438
14. Eguro K, Hauck S (2008) Enhancing timing-driven FPGA placement for pipelined netlists. DAC 2008, pp 34–37
15. Das D et al (2006) FA-STAC: a framework for fast and accurate static timing analysis with coupling. ICCD 2006
16. Kazda MA et al (2008) US Patent application 20080209376: system and method for sign-off timing closure of a VLSI chip, 2008
17. Sapatnekar SS (2004) Timing. Kluwer Academic Publishers, Boston
18. Goering R (2005) Timing analysis needs overhaul. Speaker says. EE Times (February, 2005)
19. Bronnenberg D (1999) Static timing analysis increases ASIC performance. Integr Sys Design
20. Wang ARR (1989) Algorithms for multilevel logic optimization. PhD thesis, University of California, 1989
21. Scheffer L, Lavagno L, Martin G (2006) EDA for IC implementation, circuit design, and process technology. CRC Press, Boca Raton
22. Mains RE et al (1994) Timing verification and optimization for the power PC processor family. ICCD 1994, pp 390–393

23. Burstein M, Youssef MN (1985) Timing influenced layout design. DAC 1985, pp 124–130
24. Donath WE, Kudva P, Stok L, Villarrubia P, Reddy LN, Sullivan A, Chakraborty K (2000) Transformational placement and synthesis. DATE 2000, pp 194–201
25. Vygen J (2006) Slack in static timing analysis. IEEE Trans CAD 25(9):1876–1885
26. Donath WE, Hathaway DJ (2001) US Patent 6,202,192: distributed static timing analysis, 2001
27. Donath WE, Hathaway DJ (2003) US Patent 6,557,151: distributed static timing analysis, 2003

Chapter 5
Gate Sizing During Timing-Driven Placement

A fundamental challenge addressed by physical synthesis is reducing circuit delay by altering timing-critical paths. Several techniques can be applied to achieve this optimization: buffer insertion, gate sizing, cell movement, etc. In this work, we propose a powerful new technique that moves and resizes multiple cells simultaneously to straighten critical paths, thereby reducing delay and improving worst negative slack. Our approach offers several key advantages over previous formulations, including the accurate modeling of objectives and constraints in the true timing model, and a guarantee of legality for all cell locations, thereby avoiding overlap with large fixed blockages and the need for subsequent legalization. We formulate the path smoothing problem in terms of a *disjunctive timing graph*, and develop a computation of optimal locations by incorporating a generalization of static timing analysis into an efficient branch-and-bound framework. Empirically, our approach consistently improves solution quality in a large-scale modern industrial benchmark. Experimental results indicate that the techniques used in this chapter are accurate enough to improve the critical path optimization and slack-histogram compression stages of physical synthesis, as illustrated by Fig. 5.1.

5.1 Introduction

Timing-driven placement [1–3] is a critical step in any physical synthesis flow, and has received steadily increased attention in recent years [4]. Due to its computational expense and complexity, several algorithms optimize timing objectives indirectly by relying on edge- or net-weighting methods to cast the problem into one of weighted wirelength-driven placement. Whether such approaches can truly be considered *timing-driven*—or instead, merely *timing-influenced*—remains a matter of debate.

A great deal of focus has been given specifically to the construction of cheap, incremental methods for improving timing along critical paths in an optimized design,

D. A. Papa and I. L. Markov, *Multi-Objective Optimization in Physical Synthesis of Integrated Circuits*, Lecture Notes in Electrical Engineering 166,
DOI: 10.1007/978-1-4614-1356-1_5, © Springer Science+Business Media New York 2013

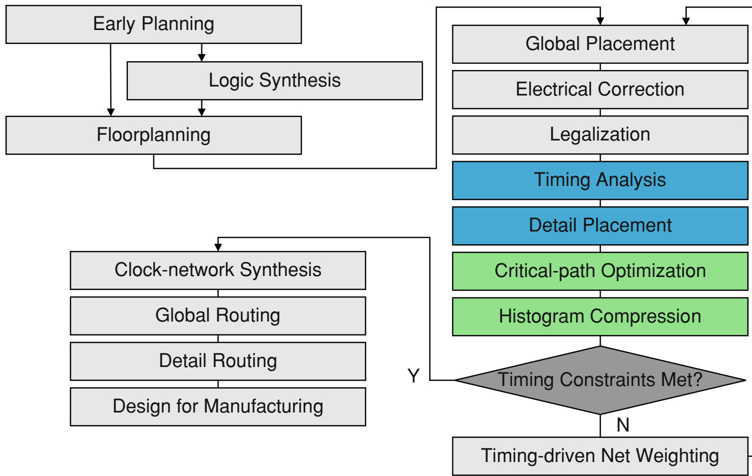


Fig. 5.1 The contributions in this chapter improve the results of the critical path optimization and slack-histogram compression stages of physical synthesis

a problem we loosely refer to as *path-smoothing*. Whether a design simply remains poorly optimized after running existing P&R tools, or whether one needs to close on timing after the application of ECOs, there remains a high demand for efficient and automated techniques for timing-driven path smoothing.

Prior work on this topic has varied widely in the treatment of *model accuracy*, including various assumptions about physical properties (e.g., gate delay and wire delay) as well as constraints to be enforced in the solution (e.g., must have a legal placement, or be subsequently buffered, or repowered, etc.). They also differ in the specific computational frameworks used to achieve the optimization (e.g., a local search, greedy algorithm, or dynamic program). These two considerations—choice of model and choice of algorithm—are strongly coupled, as a particular model often gives rise to a specific search space or methodology.

One of the more popular approaches to incremental timing-driven placement in the literature is *linear programming* (LP) [5]. While several flavors exist, a conventional LP formulation typically involves the association of decision variables with the coordinate(s) of each gate or pin, and the expression of pairwise timing dependencies between these variables using linear constraints. Since the relationship between pin-to-pin wire delay and Manhattan distance is quadratic rather than linear, the inaccuracy of this linear model has been addressed in various ways. For instance, Choi and Bazargan [6] consider an objective function that minimizes total cell displacement to prevent cases where large cell movement invalidates the linear model. The model of Wang et al. [7] assumes that LP-based optimization is followed by perfect buffer insertion. A piecewise-linear approximation of the quadratic function is employed by Chowdhary et al. [8]. Luo et al. [9] optimize a weighted slack objective in which Elmore delays computed from the original placement are scaled linearly by

a coefficient. Chapter 3 also contains an LP formulation to be used after stripping all repeaters out from combinational logic and subsequently re-buffering long wires as a post-processing step.

Despite these efforts, linear programming formulations suffer from additional complications aside from their inability to capture a faithful delay model. Among these deficiencies includes the potential to create cell overlap; although several post-placement legalization techniques have been adopted in academia and industry [10, 11], there is no guarantee that these procedures will preserve improvements made to timing. Other solutions, including the restriction of cell movement to geometrically disjoint bounding boxes [9, 12], severely overconstrains the problem by preventing large and potentially beneficial leaps. Furthermore, a trend in modern ASIC designs is the presence of large fixed macros that serve as blockages and limit the possible legal locations for movable logic. For such designs, an accurate model should avoid solutions that place gates on top of fixed obstacles. Finally, optimizing other discrete design parameters such as gate sizes and placement simultaneously requires an approach that accounts for decisions with finitely-many alternatives, since solutions produced by continuous gate-sizing [13] may degrade unacceptably when mapped to a standard cell library. Such continuous-to-discrete mappings present challenges for any of the aforementioned mathematical programming approaches.

In this chapter, we introduce a new direction for incremental, timing driven placement under models with high-fidelity to an industrial static timing analysis engine. In contrast to prior efforts that approximate timing objectives using weighted wirelength driven metrics (and approximate discrete decision variables using lossy, continuous models), our approach maintains a high degree of accuracy by explicitly encoding placement alternatives into a fully discretized graph-based representation, matching the true timing objectives as computed by an industrial static timing analysis engine. Specifically, we consider a formulation in which a finite set of *pre-legalized* candidate locations and power levels are identified for each movable gate, allowing a more faithful and accurate encoding of pairwise delay, as well as enabling the avoidance of large fixed macros that serve as blockages. This formulation gives way to a *disjunctive timing graph*, a compact structure that captures all possible conditional timing arcs for a given problem instance. We then propose a means to compute optimal solutions to this model using an efficient branch-and-bound framework that considers the simultaneous placement of multiple gates. To obtain upper bounds on worst negative slack (WNS), we develop a means to perform *Generalized Static Timing Analysis* (GSTA), an extension of traditional static timing analysis that produces optimistic slack values even when only a subset of gates have been assigned to their respective candidates.

The remainder of this chapter is organized as follows. In Sect. 5.2, we present a brief review of static timing analysis and timing-driven placement. In Sect. 5.3, we describe our problem formulation in detail, including the selection of movable gates and candidate assignments. In Sect. 5.3, we formally define the *Disjunctive Timing Graph*, and describe our optimization algorithm in Sect. 5.4. Finally, in Sect. 5.5, we present experimental results of our system–named RATCHET- followed by concluding thoughts.

5.2 Background

Timing-driven placement seeks non-overlapping locations of the cells of a circuit such that the worst slack in the design is maximized. This is in contrast to wirelength-driven placement wherein the objective is to minimize total half-perimeter wirelength (HPWL).

The problem that *incremental* timing-driven placement aims to solve is the following: given an optimized design, select a subset of gates M from G (where M may just consist of a single gate) and find a new location for each gate in M such that the worst negative slack (WNS) in the entire subcircuit is improved:

$$\text{WNS}(G) = \min_{v \in V(G)} (\min(0, \text{slack}(v))) \tag{5.1}$$

For tie-breaking, a total negative slack (TNS) component may also be optimized, which is equal to the sum of all negative slacks:

$$\text{TNS}(G) = \sum_{v \in V(G)} (\min(0, \text{slack}(v))) \tag{5.2}$$

An algorithm that solves this problem is called a *transformation*, using the terminology of [14, 15]. More generally, a transformation is any optimization procedure designed to incrementally improve timing while preserving the logical correctness of a circuit. Other examples of transformations include: buffering a single net, resizing a gate, cloning a cell, swapping equivalent pins on a gate, etc. Transformations are invoked in a physical synthesis flow by *controllers*. For example, a controller for critical path optimization may attempt a transformation on the 100 most critical cells. A controller designed for compression may consider every cell that fails to meet its timing constraints.

5.3 Problem Formulation

In formulating our problem, we require three steps to be performed in sequence. The first identifies the set of gate(s) that should be considered for movement, such as the most critical gates and their adjacent neighbors. Next, a set of candidate assignments is computed for each movable gate; if desired, these candidates can satisfy current constraints in the physical synthesis flow, such as avoidance with obstacles, keep-out regions, etc. Finally, a timing arc is extracted for each pair of candidate assignments.

Selection of Movables

The task of selecting a set of movable gates is shared by many timing-driven placement algorithms. Since our transformation can be enacted by any high-level con-

troller, we are free to assume that an external mechanism chooses individual gates for relocation (e.g., such as all imbalanced latches [16]). In expanding the movable logic to include additional gates, various heuristics have been proposed that incorporate the degree of neighbors' criticality [7, 9]. We combine the criticality adjacency network of [9] with an N-hop neighborhood, in which any gate within N steps of the targeted gate is included in the set of movable cells; however, we stress that our core timing-driven placement engine can be parametrized with any well-formed gate-selection strategy. All peripheral gates connected to the movable logic are collected to form a set of fixed nodes.

Selection of Candidate Assignments

After the set of movable gates has been determined, we precompute a discrete set of *candidate assignments* for each. Our method imposes no restrictions on how these candidates are obtained, as there are several possible strategies ranging from simple to exotic. In the case of placement, examples include the following:

- For a gate whose current coordinate is (x, y), consider the candidates:

$$(x + \Delta x, y)$$
$$(x - \Delta x, y)$$
$$(x, y + \Delta y)$$
$$(x, y - \Delta y)$$

  for a given $(\Delta x, \Delta y)$, in addition to the current coordinate of the gate. Such a set corresponds to the directions *up*, *down*, *left*, and *right*.
- The closest *feasible* locations to each of the candidates in the above set (i.e., respecting blockages and large fixed macros).
- The n nearest feasible locations closest to the gate's current coordinate, for some specified number n.
- A set of m or more locations obtained by m other incremental timing-driven placement algorithms for single gates.

The precomputation of candidate assignments bears some resemblance to graph-based approaches to buffer insertion [17]; however, it reflects a significant deviation from the vast majority of existing incremental timing-driven placement approaches that assume a continuous (and globally feasible) geometric plane. Refer to Fig. 5.2 for an example in which each of five movable gates (*b*, *c*, *d*, *e*, and *f*) has between two and four candidates each. The presence of a single large macro prevents candidate locations from appearing toward the center of the subcircuit.

Although our experiments are limited to multi-move placement, it is important to note that candidate assignments need not necessarily be new physical locations; for instance, cell f is shown to have two possible sizes, indicating different candidate power levels for the gate. Similar assignments can be obtained if considering dual

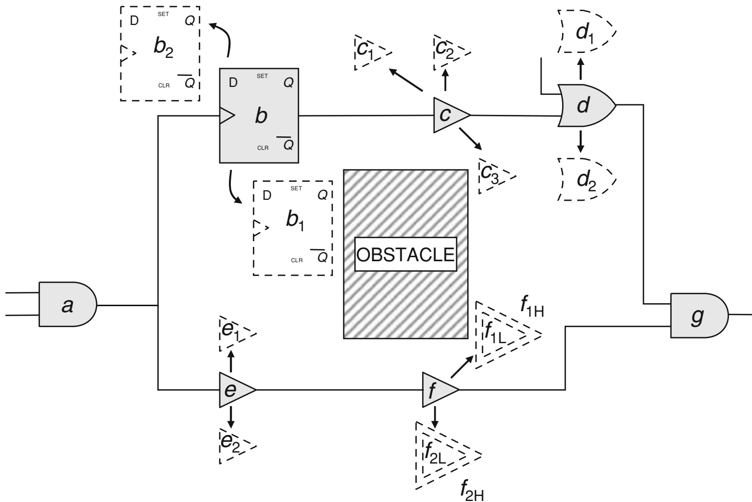


Fig. 5.2 Gates *a* and *g* are fixed. Alternate candidate locations for movable gates *b*, *c*, *d*, *e*, and *f* have been determined. Gate *f* also has two candidate power levels

threshold voltage (V_t) levels [18]. As will be demonstrated later, this generalization permits the simultaneous optimization of placement and other transformations, in a similar spirit to [13] but imposing discrete (rather than continuous) values.[1]

Disjunctive Timing Model

The final step in our problem formulation is to construct a *conditional timing arc* for each pair (l_i, l_j) of candidate assignments between source and sink, which specifies the delay that would occur between them. We refer to the arcs between these nodes as being *conditional* since they depend on the chosen candidate(s). Our algorithm makes no assumptions about the correlation between the values of these timing arcs, and any delay model may be used. For instance, half-perimeter wirelength (HPWL) may be used to create a linear-delay model if rebuffering will be performed as a postprocessing step. In this case, delay is a pure function of geometric location:

$$\text{delay}(l_i, l_j) = \tau * \text{dist}(l_i, l_j) \tag{5.3}$$

where τ is a technology dependent parameter equal to the ratio of the delay of an optimally-buffered, arbitrarily-long wire segment to its length:

[1] In practice, a discrete set of candidate values is more appropriate when working with a predefined cell library, and discretization from continuous values is NP-complete in general [19].

$$\tau = \frac{\text{delay}(wire)}{\text{length}(wire)} \tag{5.4}$$

Alternatively, if rebuffering will not occur, more elaborate and accurate timing models are appropriate. For instance, the Elmore delay model captures a quadratic function of wirelength on 2-pin nets:

$$\text{delay}(l_i, l_j) = K_D * r * \text{dist}(l_i, l_j) * \left(\frac{c * \text{dist}(l_i, l_j)}{2} + C_{pin_j}\right) \tag{5.5}$$

The delay between gates on higher degree nets may be obtained by querying a full-blown industrial timing engine, reconstructing Steiner trees from scratch [20] or via topological repair [21], or instead by cheaper methods of estimation [22].

The Disjunctive Timing Graph

In the previous paragraphs, we identified the three major components in our formulation of incremental timing-driven placement: selection of movable gates, selection of candidate assignments, and generation of conditional timing arcs. We now formally define an extension of the classical timing graph that captures these attributes:
Definition: A *disjunctive timing graph* G is defined by a tuple (V, C, E), where (as in the traditional timing graph) each element $v \in V$ corresponds to a logic gate in the circuit, and a pair of vertices, $u, v \in G$, are connected by a directed edge $e(u, v) \in E$ if there is a connection from the output of gate u to the input of gate v. The additional parameter C is a mapping from any gate $v \in V$ to a set of candidate assignments $\{v_1, \ldots, v_{C_v}\}$. Each edge has an associated *conditional delay function*, $\delta(u_i, v_j) \rightarrow \Re^+$, indicating the delay between any pair of candidates u_i and v_j. □

The disjunctive timing graph encodes all combinations of pairwise net delays, with each vertex corresponding to a *meta-node* representing a set of candidates. See Fig. 5.3 for an illustration corresponding to our example. In subsequent sections, it will be useful to refer to a solution to a disjunctive timing graph, which is obtained by selecting a candidate for each gate and extracting the appropriate timing arcs.
Definition: A *solution* S to a disjunctive timing graph G is a mapping $V \rightarrow C(V)$, in which a single candidate is selected from the domain of each gate v in V. A solution corresponds to a traditional timing graph $G' = (V', E')$, in which the vertices V' of G' correspond to the candidates selected from G, and the weight of each edge $e'(u, v) \in E'$ is taken from $\delta(u_i, v_j)$, where u_i and v_j are the candidates chosen for gates u and v (respectively). □

A solution S to a disjunctive timing graph is deemed *optimal* with respect to an objective function O (e.g., worst negative slack, or delay) if the value $O(S)$ is as good or better than $O(S')$ for every other solution S'. Observe that, in contrast to a traditional timing graph, a simple longest path calculation through the disjunctive graph does not suffice, even if optimizing for delay; such a computation maximizes

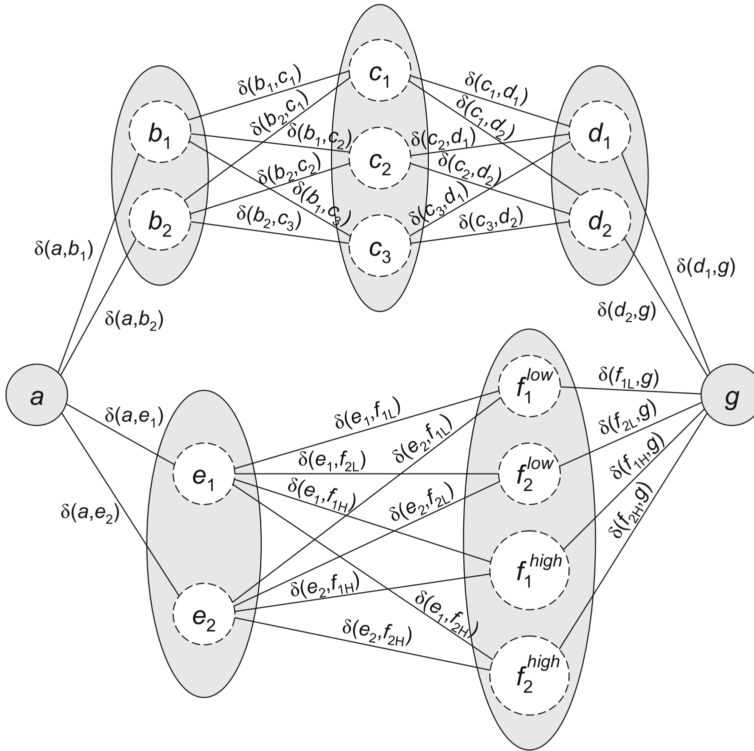


Fig. 5.3 The *disjunctive timing graph* for our running example. Each timing arc between a pair of candidate assignments has a distinct value; the actual arc between any two meta-nodes in a complete solution depends on the candidates chosen

the longest path, whereas we instead seek to select a set of candidates such that the longest path is *minimized*.

5.4 Our Simultaneous Placement and Gate-Sizing Algorithm

The previous section alludes to one possible algorithm for the optimization of a disjunctive timing graph: generate every possible solution S, evaluate its cost, and return the best solution, an approach generally referred to as *exhaustive enumeration*. However, when considering even moderately-sized problems, the computational expense of this brute-force procedure may be prohibitively expensive. In particular, given M movable gates and C candidates per gate, a total of C^M solutions will be considered, with each requiring a full pass of Static Timing Analysis to determine worst negative slack.

Of course, if strict optimality is not required, other possibilities exist. A simple greedy strategy could consider the movement of each gate individually, choosing the location that maximizes worst slack assuming all other gates are held fixed (requiring the generation of $M \times C$ solutions). However, in many practical cases, it is impossible to improve timing by moving only a single gate. For instance, suppose a large gate is being driven by a relatively weak driver, in which case neither gate can be moved a significant distance from the other without imposing an electrical violation. To accommodate a wide range of instances, our algorithm must consider the simultaneous movement of multiple gates. In response, we turn to the well-known algorithmic framework of branch-and-bound.

Recursive Branch-and-Bound Search

Branch-and-bound is a widely-studied, commonly used depth-first-search optimization technique. Rather than explore all possible combinations of assignments, branch-and-bound prunes partial solutions based on estimates of the objective function calculated during search. Backtracking occurs whenever the upper bound on the value of a partial solution is no better than that of the best found. Recent work in the coupling of graph-based procedures with branch-and-bound have demonstrated runtime reductions from days to seconds in floorplanning domains [23], although such advances have yet to be extended toward problems in timing-driven placement.

In Fig. 5.4, we display a possible search tree for our running example that has been pruned as a result of bounding. The partial solution $S = \{(b \leftarrow b_1), (c \leftarrow c_1), (d \leftarrow d_1)\}$ is eventually extended to form a complete solution; however, in exploring the partial solution $S' = \{(b \leftarrow b_1), (c \leftarrow c_1), (d \leftarrow d_2)\}$, search is aborted. By visual inspection of Fig. 5.2, the distance between candidates c_1 and d_2 is relatively large, and contributes to an excessively long delay in S'.

In order to make branch-and-bound effective, one must choose intelligent metrics to guide the process of node expansion. We identify two selection strategies for the branching schedule: the *gate ordering*, used to determine which gates should be instantiated earliest in search, and the *candidate ordering*, used to determine which partial solutions should be attempted before others. For the former strategy, gates that fall along the critical path are given highest priority; since it is the placement of these gates that has the highest impact on worst negative slack, their assignment should not be postponed. For the latter strategy, we order candidates by determining their effect on the bounding calculation, as described in the next section.

Generalized Static Timing Analysis

One question raised by the backtracking framework is how to compute upper bounds on worst negative slack when only a subset of candidate assignments have been chosen. Traditional versions of Static Timing Analysis assume that all timing arcs have been fixed, whereas in our model, a disjunctive set of choices remains until a leaf node (i.e., a fully instantiated solution) in search is encountered.

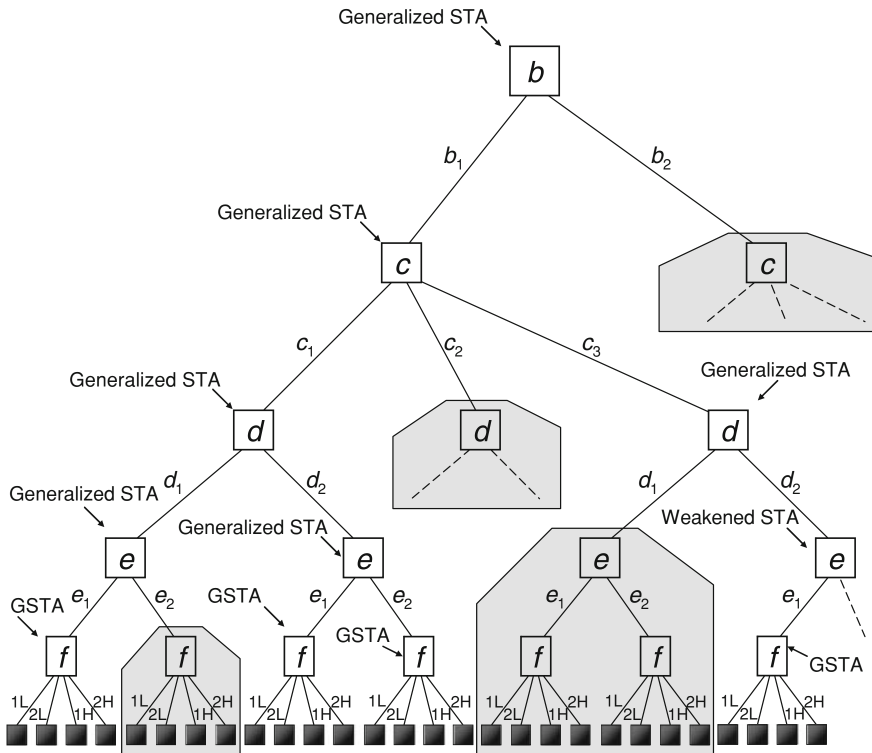


Fig. 5.4 *Branch-and-bound* computes an upper bound on the worst negative slack at every node in search. Any partial solution that cannot improve upon the best known is pruned

We resolve this by performing a generalized version of Static Timing Analysis, which we call *Generalized Static Timing Analysis* (GSTA). In GSTA, each edge in the graph corresponding to a source/sink pair is replaced with the most optimistic (or least constraining) possible timing arc. These weakened values may be safely propagated through the graph in place of any particular timing arc. Actual arrival times, required arrival times, and slacks are computed as is typically done in STA, using these weakened values during propagation. More formally, the actual arrival times and required arrival times for a partial solution S are computed by the following expressions:

$$\mathrm{AAT}(v) = \max_{\{u|e(u,v)\}} (\mathrm{AAT}(u) + \min_{u_i \in C(u), v_j \in C(v)} (\delta_S(u_i, v_j))) \tag{5.6}$$

$$\mathrm{RAT}(u) = \min_{\{v|e(u,v)\}} (\mathrm{RAT}(v) - \min_{u_i \in C(u), v_j \in C(v)} (\delta_S(u_i, v_j))) \tag{5.7}$$

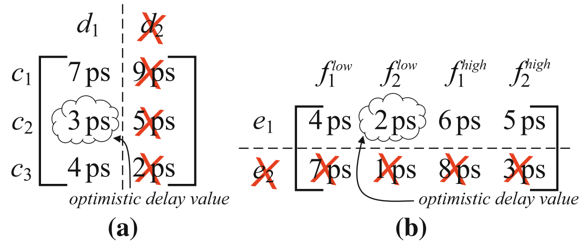


Fig. 5.5 The delay functions $\delta(c, d)$ and $\delta(e, f)$. Here we show the case where the partial solution S includes the decisions $(d \leftarrow d_1)$ and $(e \leftarrow e_1)$. The weakened delay values are $\delta_S(c, d) = 3$ ps and $\delta_S(e, f) = 2$ ps

Since these weakened delay values must hold in any fully instantiated solution, the soundness of the procedure is preserved. Although the worst slack estimate calculated from this procedure may not be achievable in any complete solution,[2] we are guaranteed that no extension of the partial solution can improve upon it.

If a candidate assignment for one movable gate has been chosen, some entries in the conditional delay function may be disregarded. For instance, in Fig. 5.5, we consider the case when the partial solution S includes the decisions $\{(d \leftarrow d_1), (e \leftarrow e_1)\}$. Since no extension of this particular search node will consider the selection of candidate d_2, an entire column of entries can be ignored, raising the optimistic delay of the conditional function $\delta_S(c, d)$ up to 3 ps from 2 ps. A similar effect is observed for $\delta_S(e, f)$. If both gates have been instantiated with candidate assignments, the actual timing arc between those specific candidates may be used.

To address issues such as resource contention (i.e., when two different gates attempt to take the same location), one may check for such conflicts during search, backtracking accordingly. Alternatively, such locations may be pre-processed prior to search, so that only one location appears as the candidate of any cell.

Observe that in the case that all gates have only a single candidate assignment (or, equivalently, that a single candidate has been chosen for each movable gate), Generalized STA reduces to traditional STA. It should also be noted that our branch-and-bound technique is an *anytime* algorithm, and may be interrupted prior to completion to obtain a suboptimal solution (e.g., based on a timeout limit, maximal number of nodes, etc.).

The Complete Flow

In Fig. 5.6, we present the full pseudocode for our algorithm, named RATCHET. After selecting the targeted gate (line 1) and its surrounding movable neighbors (line 2), the current location of each gate is stored into the best known solution (*BestSol*). The algorithm then repeats the remaining steps for a given number of *Iterations* (line 4). Within each iteration, candidate assignments for each movable gate are computed (lines 5–6), as well as the appropriate timing arcs for pairs of candidate

[2] Interestingly, for subcircuits whose topology is that of a tree, a slight variation of GSTA can provide provably achievable upper bounds; however, due to space limitations, we omit the details in this 6-page submission.

Fig. 5.6 Pseudocode for the RATCHET algorithm. It begins by exploring a subcircuit from a seed and constructing the disjunctive timing graph. The final step is to solve for the best solution in disjunctive timing graph

```
RATCHET(Design D, int Iterations)
1. Gate G ← SELECTTARGETEDGATE(D)
2. Set⟨Gate⟩ M ← SELECTMOVABLES(D, G)
3. Set⟨(Gate,Loc)⟩ BestSol ← CURRENTASSIGNMENTS(M)
4. for iter = 1,2,... Iterations
5.    for each u ∈ M
6.       Set⟨Loc⟩ C_u ← GETCANDIDATEASSIGNMENTS(u)
7.    for each pair of adjacent gates u, v ∈ M
8.       for each candidate u_i ∈ C_u
9.          for each candidate v_j ∈ C_v
10.            arcs_{u,v}(u_i, v_j) ← GETTIMINGARC(u_i,v_j)
11.   SOLVE(∅, M, C, arcs)
12. return BestSol
```

```
SOLVE(Set⟨(Gate,Loc)⟩ S, Set⟨Gate⟩ U, Set⟨Set⟨Loc⟩⟩ C, arcs)
1. if (WORSTSLACKUB(S, arcs) ≤ WORSTSLACK(BestSol, arcs))
2.    return
3. if (TERMINATIONCRITERIONREACHED()) // timeout, # nodes, ...
4.    return
5. if (U = ∅)
6.    BestSol ← S; return
7. u ← CHOOSEMOVABLE(U)
8. Set⟨Gate⟩ U′ ← U − {u}
9. for each candidate u_i ∈ C_u
10.   Set⟨(Gate,Loc)⟩ S′ ← S ⋃ {(u, u_i)}
11.   COMPUTEDAG(S′, arcs)
12.   SOLVE(S′, U′, C, arcs)
```

assignments between adjacent gates (lines 7–10). These data are passed to the recursive function SOLVE(line 11). Upon its return (line 12), the optimized solution will be stored in *BestSol*.

Function SOLVE is given the current partial solution of candidate assignments to gates (S), the unassigned gates (U), the candidate assignments (C), and the timing arcs ($arcs$). If branch-and-bound detects that worst slack cannot be improved in any extension of this node, search is aborted (lines 1–2). Similarly, if any other termination criteria have been reached (such as a timeout limit, or a maximal number of search nodes), the function return as well (lines 3–4). If a leaf node in the search tree has been reached (line 5), the fully instantiated solution is recorded as the best known (line 6). Otherwise, a movable gate is selected heuristically (line 7), removed from the set of unassigned gates (line 8), and each of its candidate assignments is attempted (line 9). For each location, the partial solution is extended appropriately (line 10), and the DAG is recomputed to reflect the new assignment (line 11). The function then recurses (line 12) and returns when all candidates have been attempted.

RATCHET is meant to be applied in an iterative fashion; each call perturbs the location of movable gates, and a fresh set of candidate assignments are generated from this new solution. This process continues until a maximal number of iterations are attempted, or a threshold on minimal improvement cannot be met. In the unlikely event that a solution is found to degrade timing (for instance, if delay values for the

Table 5.1 Statistics for path smoothing benchmarks

Name	# gates	# mov.	# nets	init slack	init FOM
ibm-ps-01	3	1	2	−549 ps	−549 ps
ibm-ps-02	4	2	3	−522 ps	−801 ps
ibm-ps-03	6	3	5	−260 ps	−477 ps
ibm-ps-04	8	4	6	−758 ps	−1516 ps
ibm-ps-05	15	7	15	−943 ps	−1986 ps
ibm-ps-06	18	9	16	−411 ps	−1174 ps
ibm-ps-07	19	10	17	−1171 ps	−3513 ps
ibm-ps-08	21	13	18	−288 ps	−2537 ps
ibm-ps-09	34	15	33	−307 ps	−2726 ps
ibm-ps-10	58	21	57	−782 ps	−1863 ps
ibm-ps-11	96	29	103	−297 ps	−2927 ps
ibm-ps-12	164	49	205	−252 ps	−2149 ps

The # gates and # nets columns shows the total number of gates and nets, while the # mov. column shows how many of the gates are movable. The init slack column shows the worst initial slack and init FOM shows the sum of negative slacks

model had been inaccurately estimated), we adopt a *do-no-harm philosophy* [24, 16] by reverting the design back to its pre-transformation state.

5.5 Empirical Validation

In order to evaluate the efficacy of RATCHET, we extracted twelve subcircuits from a large, modern 65nm industrial design that contains several macros, keep-out regions, and other blockages. A summary of these subcircuits is given in Table 5.1.

Since the disjunctive nature of our problem formulation escapes the expressive power of LP formulations in previous work, we compare our full implementation of RATCHET against a simple variation on the aforementioned brute-force approach of exhaustive enumeration. For this set of experiments, we limit run RATCHET with a controller that selects imbalanced latches, and vary the number of movable gates to measure scalability. For candidate selection, we select four locations around the chip (effectively, the legalized positions corresponding to coordinates to the right, the left, above, and below each movable gate). Any duplicate locations after the legalization process are lumped into a single candidate. Exhaustive enumeration is, as expected, capable of producing optimal solutions, but with a significant runtime penalty. Our branch-and-bound algorithm is able to improve worst negative slack and TNS on all subcircuits with comparatively negligible runtime.

Table 5.2 Experimental Results on a large industrial design with a 2.2ns clock

Name	old slack	old FOM	Exhaustive Enumeration			RATCHET (B&B)		
			new slack	new FOM	cpu (s)	new slack	new FOM	cpu (s)
ibm-ps-01	−549 ps	−549 ps	0 ps (24.95%)	0 ps	0.01	0 ps (24.95%)	0 ps	0.01
ibm-ps-02	−522 ps	−801 ps	−231 ps (13.23%)	−450 ps	0.03	−231 ps (13.23%)	−450 ps	0.05
ibm-ps-03	−260 ps	−477 ps	−25 ps (10.68%)	−36 ps	0.2	−25 ps (10.68%)	−36 ps	0.04
ibm-ps-04	−758 ps	−1516 ps	−153 ps (27.50%)	−307 ps	0.52	−153 ps (27.50%)	−307 ps	0.03
ibm-ps-05	−943 ps	−1986 ps	−704 ps (10.86%)	−1388 ps	0.92	−704 ps (10.86%)	−1388 ps	0.05
ibm-ps-06	−411 ps	−1174 ps	−180 ps (10.50%)	−571 ps	3.2	−180 ps (10.50%)	−571 ps	0.08
ibm-ps-07	−1171 ps	−3513 ps	−897 ps (12.45%)	−2690 ps	7.4	−897 ps (12.45%)	−2690 ps	0.2
ibm-ps-08	−288 ps	−2537 ps	−62 ps (10.27%)	−200 ps	14	−62 ps (10.27%)	−200 ps	0.43
ibm-ps-09	−307 ps	−2726 ps	−148 ps (07.23%)	−870 ps	68	−148 ps (07.23%)	−870 ps	0.69
ibm-ps-10	−782 ps	−1863 ps	−513 ps (12.23%)	−1492 ps	129	−513 ps (12.23%)	−1492 ps	0.58
ibm-ps-11	−297 ps	−2927 ps	−132 ps (07.50%)	−2293 ps	290	−132 ps (07.50%)	−2293 ps	1.52
ibm-ps-12	−252 ps	−2149 ps	−19 ps (10.59%)	−77 ps	430	−19 ps (10.59%)	−77 ps	1.55
avg.	−545 ps	−1852 ps	−255 ps (13.17%)	−865 ps	78.61	−255 ps (13.17%)	−865 ps	0.44

5.6 Conclusions

The path smoothing problem in timing-driven placement is one that fundamentally admits a discrete solution space, and requires a corresponding methodology to efficiently perform discrete optimization. In response, we have proposed a new direction for incremental, timing-driven physical synthesis that directly optimizes timing objectives using accurate, high-fidelity models. RATCHET couples the graph-based techniques of static timing analysis with a powerful branch-and-bound strategy to achieve efficient optimization of critical paths in late stages of refinement. In contrast to prior efforts that approximate timing objectives using weighted-wirelength driven metrics, our approach maintains a high degree of accuracy by explicitly encoding placement alternatives into a *disjunctive timing graph*. We have also developed a method of *Generalized Static Timing Analysis* necessary to obtain upper bounds on worst negative slack (WNS) when only a subset of gates have been assigned to their respective locations, leading to an efficient branch-and-bound algorithm shown to improve the solution quality of large industrial designs.

References

1. Burstein M, Youssef MN (1985) Timing influenced layout design. In: DAC, pp 124–130
2. Marquardt A, Betz V, Rose J (2000) Timing-driven placement for FPGAs. Proceedings of FPGA 2000, pp 203–213
3. Swartz W, Sechen C (1995) Timing driven placement for large standard cell circuits. In: DAC, pp 211–215
4. Alpert CJ, Chu C, Villarrubia PG (2007) The coming of age of physical synthesis. In: ICCAD 2007, pp 246–249
5. Jackson MA B, Kuh ES (1989) Performance-driven placement of cell based IC's. In: DAC, pp 370–375
6. Choi W, Bazargan K, (2003) Incremental placement for timing optimization, In: ICCAD 2003, pp 463–466
7. Wang Q, Lillis J, Sanyal S (2005) An LP-based methodology for improved timing-driven placement. In: ASP-DAC 2005, pp 1139–1143
8. Chowdhary et al A (2005) How accurately can we model timing in a placement engine?. In: DAC, pp 801–806
9. Luo T, Newmark D, Pan DZ (2006) A new LP based incremental timing driven placement for high performance designs, In: DAC , pp 1115–1120
10. Brenner U, Pauli A, Vygen J (2004) Almost optimum placement legalization by minimum cost flow and dynamic programming. In: ISPD 2004, pp 2–9
11. Hill D (2002) Method and system for high speed detailed placement of cells within an integrated circuit design. US patent 6370673, 2002
12. Halpin B, Chen CYR, Sehgal N (2001) Timing driven placement using physical net constraints. In: DAC, pp 780–783
13. Chen W, Hsieh C-T, Pedram M (2000) Simultaneous gate sizing and placement. IEEE Trans CAD 19(2):206–214
14. Donath WE, Kudva P, Stok L, Villarrubia P, Reddy LN, Sullivan A, Chakraborty K (2000) Transformational placement and synthesis. In: DATE 2000, pp 194–201
15. Trevillyan L et al (2004) An integrated environment for technology closure of deep-submicron IC designs. IEEE Des Test Comput 21(1):14–22

16. Papa DA et al (2008) RUMBLE: an incremental, timing-driven. physical-synthesis optimization algorithm. In: ISPD, pp 2–9
17. Gao Y, Wong DF (2001) A graph based algorithm for optimal buffer insertion under accurate delay models. In: DATE 2001, pp 535–539
18. Lee D, Blaauw D, Sylvester D (2005) Static leakage reduction through simultaneous V_t/T_{ox} and state assignment. IEEE Trans CAD 24(7):1014–1029
19. Lee WN (1993) Strongly NP-hard discrete gate sizing problems. In: ICCD 1993, pp 468–471
20. Chang K-H, Markov IL, Bertacco V (2007) Safe delay optimization for physical synthesis. In: ASP-DAC, pp 628–633
21. Ajami AH, Pedram M (2001) Post-layout timing-driven cell placement using an accurate net length model with movable steiner points. In: ASP-DAC, pp 595–600
22. Alpert CJ, Devgan A, Kashyap CV (2001) RC delay metrics for performance optimization. IEEE Trans CAD 20(5):571–582
23. Moffitt MD, Pollack ME (2006) Optimal rectangle packing: a meta-CSP approach. In: ICAPS 2006, pp 93–102
24. Ren H et al (2007) Hippocrates: first-do-no-harm detailed placement. In: ASP-DAC 2007, pp 141–146

Part III
Broadening the Scope of Circuit Transformations

Chapter 6
Physically-Driven Logic Restructuring

In a complete physical synthesis flow, many optimizations are applied to critical paths that are already optimized by a series of powerful transformations, as described in Chap. 2. Transforms that can further improve the timing of such paths are invaluable for timing closure. Finding such transformations and applying them efficiently is challenging. To this end, we explore new techniques for logic cloning (gate duplication) to improve timing closure in a physical synthesis environment.

With a buffer-aware interconnect timing model, new polynomial-time optimal algorithms are proposed for timing-driven cloning, including finding appropriate sink partitions (fanout identification) for the original and the duplicated gates, as well as optimized physical locations for both gates. In particular, we present an $O(m)$-time optimal algorithm to maximize the worst slack if the original gate is movable, and an $O(m \log m)$-time optimal algorithm if the original gate is fixed, where m is the number of fanouts. To the best of our knowledge, this work is the first to consider the timing-driven cloning problem under a buffer-aware interconnect delay model.

6.1 Introduction

Physical synthesis is a complex process that combines physical design with netlist restructuring to achieve design closure. As described in Chap. 2, physical synthesis typically consists of several stages including placement, legalization, critical-path optimization, etc. Among these stages, the critical-path optimization stage is particularly important. It takes a design that is legally placed and initially optimized for timing, and restructures critical paths by applying a multitude of different transformations, such as gate sizing, V_{th} tuning, and buffering. It is usually not difficult to improve timing early in a physical synthesis flow. However, it is more challenging to improve timing if the circuit has been optimized by a series of powerful transformations in a physical synthesis flow.

D. A. Papa and I. L. Markov, *Multi-Objective Optimization in Physical Synthesis of Integrated Circuits*, Lecture Notes in Electrical Engineering 166,
DOI: 10.1007/978-1-4614-1356-1_6, © Springer Science+Business Media New York 2013

Timing closure requires a variety of netlist transformations, each addressing certain problematic structures. In this chapter, we design several highly efficient cloning techniques, also known as cell replication techniques, to improve delay along critical paths. Cloning is not a new synthesis optimization; Brglez [1] and Hwang et al. [2] use cloning as a mechanism to reduce net-cut during partitioning, and cloned gate placement has been studied in the FPGA domain [3, 4]. Since cloning helps in reducing the total capacitance loading of a high-fanout net, many existing techniques focus on technology-independent delay optimization [5–7]. A variant cloning problem that considers a load-dependent gate delay model and zero-wire delay is known to be NP-complete [7]. Under the same delay model, a cloning in sink-to-source order can improve the timing of a technology-mapped circuit [8]. Due to the computational complexity of the problem, heuristics are often proposed to speed up the technique. However, all of these techniques neglect two key features of the problem: interconnect delay and the placement of the duplicated gate. Thus, these models can be used in the logic synthesis stage of design but will be less applicable during the core stages of a physical synthesis flow.

For modern technologies, previous cloning algorithms are largely ineffective for critical-path optimization because they ignore wire delay, buffering and placement. This is explained in part by interconnect scaling, which has only recently necessitated that buffers be inserted on nearly all global nets to overcome wire resistance [9]. Consequently, when one wants to apply cloning to improve path delay, buffers that have been inserted previously limit the scope of cloning for timing improvement. To make cloning effective, one must account for buffers by considering only non-buffer sinks, and re-buffering the resulting circuit.

To the best of our knowledge, the only work which handles both cloning and buffer insertion in the placement stage is BufDup [10]. Unfortunately, they consider cloning and buffer insertion separately. In addition, BufDup uses a timing-oblivious, simple k-means based clustering algorithm to partition the fanout gates. It does contain a timing-driven post-processing step, but it can only be used to balance the capacitance loading of the two partitions and is not designed to improve timing. In contrast to [10], our cloning is based on a linear-delay model [11, 12] with the knowledge that buffered interconnect delay is linearly-proportional to its length (see Chap. 3). This model handles simultaneous buffering and cloning in an abstract and unified way. Adoption of such a delay model also helps to reduce the complexity of the gate cloning problem. This work reveals that cloning with a buffer-aware linear-delay model can be accomplished very efficiently (in polynomial time).

Other works on simultaneous timing-driven gate placement and buffering are related to this problem. RUMBLE (see Chap. 3) uses a linear-delay buffering model and linear programming techniques to solve the timing-driven latch and gate placement problem considering practical constraints. Pyramids uses computational geometry techniques to efficiently solve a one gate placement problem with a similar delay model [13]. Note that the timing-driven gate placement problem is subsumed by the timing-driven gate cloning problem, since a fixed sink partitioning reduces the cloning problem to the gate placement problem. Thus, the cloning problem is complicated by the need to find sink partitions and gate placements simultaneously.

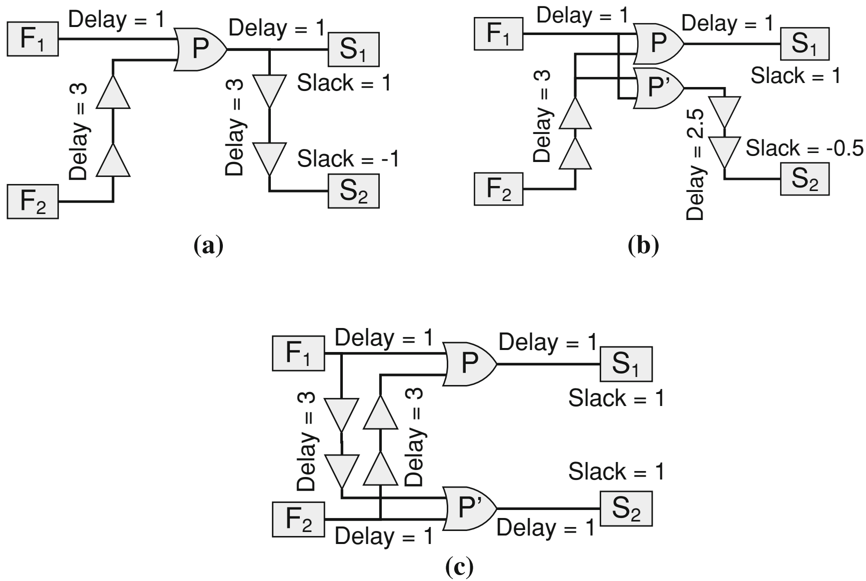


Fig. 6.1 Example of interconnect-driven cloning. The arrival times of F_1 and F_2 are 0. The required arrival times of S_1 and S_2 are 5. For simplicity, this example uses gate delays of 0. **a** Original circuit. **b** New circuit after cloning leaving buffering intact. **c** New circuit after cloning considering buffering

An example of simultaneous cloning and buffering is shown in Fig. 6.1. The arrival times of F_1 and F_2 are 0, and the required arrival times of S_1 and S_2 are 5. Consider the situation in Fig. 6.1a where we consider cloning gate P. There are two sinks S_1 and S_2 with slacks $+1$ and -1. The delays from fanins F_1 and F_2 to P are 1 and 3 respectively, as are the delays from P to S_1 and S_2, including the delay of buffers and wires along the path. If we clone P to P' while leaving the original buffer trees intact, we may get the result shown in Fig. 6.1b in which P' is placed very close to P, and the slack only improves to -0.5. Here the new location of P' is restricted by the buffers that must drive both P and P'. However, if one restructures the buffering solution to eliminate this constraint, one can obtain the superior solution in Fig. 6.1c which increases both slacks to $+1$ and obtains the physically shortest possible paths from F_1 and F_2 to S_1 and S_2. This example suggests that one must consider buffering and cloning together to effectively reduce delay.

Timing-driven buffering alone can be computationally expensive when used excessively [14]. It is also difficult to use it to derive any guidance for simultaneous cloning and buffering. To be most accurate, one should explore all possible partitionings of sinks for each net, find gate placements (i.e, with the technique in Chap. 3), re-buffer with dynamic programming, and legalize the design. The whole process is too expensive for modern designs with hundreds of thousands of nets. It

may also waste the majority of its runtime, because in many cases the new solution may be worse than the old solution, and will therefore be retracted.

Unlike the above approach, we use abstract timing models and build a theoretical guide on top of them. In our approach, the effect of buffering is modeled by a linear-delay model, introduced in Chap. 3. Our algorithms guarantee optimality under this delay model, and can also be used as a filter to identify a group of critical gates that may benefit from cloning. Even if our solution does not fix all timing problems, one can still apply more accurate gate placement techniques based on our sink partitioning and re-buffer on a small group of nets. In that way, success rate and the total turn-around-time will be improved.

The main contributions of this chapter are summarized as follows.

- We propose several polynomial-time optimal algorithms for simultaneous timing-driven cloning and buffering under a linear-delay model. Our algorithms "see through" buffer trees in the original circuit.
- For circuits surrounding a movable object, an $O(m)$-time algorithm to compute the optimal cloning that maximizes worst slack is proposed, where m is the number of fanouts.
- For circuits surrounding a fixed object, we present an $O(m \log m)$-time algorithm to compute the optimal cloning.

For the remainder of this chapter, we assume that load-based cloning techniques have already been applied during logic synthesis or an early design stage, and we will not focus on the problem of reducing capacitive load. Also, buffering should have processed all high-fanout nets before the cloning we propose. The techniques in this chapter are designed primarily for gates driving substantial interconnect delay (medium-length and long nets).

6.2 Background and Preliminaries

We outline our problem formulation as follows.

Linear Buffered-Path Delay Model. Recall the linear-delay model introduced in Chap. 3. The delay along an optimally buffered interconnect of length l is given by $\text{delay}(l) = \tau \cdot l$, where τ is a technology dependent constant. In general, τ depends on the buffer library size and the input slew rate. In this chapter, we refer to $\tau = \text{delay}(\mathit{wire})/\text{length}(\mathit{wire})$.

Problem Formulation. The circuit for the cloning problem is a directed graph $G = (V, E)$, where $V = \{P\} \cup F \cup S$, and $E = (F \times \{P\}) \cup (\{P\} \times S)$. Vertex P is the *target* gate to be duplicated, F is the set of *fanin* gates that drive P with size n, and S is the set of *fanout* gates that P drives with size m.[1] Every gate $g \in V$ is a logic gate performing certain logic functions, such as AND, OR, XOR but not buffers or

[1] Without loss of generality, we assume n and m are of the same order for simplicity of the complexity analysis.

inverters, and is associated with physical coordinates $(X(g), Y(g))$. If there are any buffers/inverters in the circuit that are fanins or fanouts of P, we will look through them to find the first non-repeater logic gate. Each fanout gate $S_i \in S$, is associated with required arrival time $RAT(S_i)$ at its input pin, and each fanin gate $F_i \in F$ is associated with arrival time $AAT(F_i)$ at its output pin.

The location of each gate in S and F can not be changed in our problem formulation, and we refer them as *fixed* gates. Note that these gates may be allowed to move during other transformations (e.g., legalization after cloning) but their locations are constrained during cloning to simplify the analysis. It may also be the case that they are fixed by designers who want to keep certain gates in specified locations, or in a late stage of the design flow, one prefers minimal perturbation to the design for stability. Gate P may be movable or fixed.

After cloning, we create a duplicated gate for P, denoted by P'. Finding a location for P' is one objective of this work. The graph G becomes $G' = (V', E')$, where $V' = P \cup P' \cup F \cup S_P \cup S_{P'}$, $E' = (F \times P) \cup (P \times S_P) \cup (F \times P') \cup (P' \times S_{P'})$. In G', each fanin gate F_i is also connected to the duplicated gate P', but fanout gates S are divided into two disjoint sets S_P and $S_{P'}$ such that $S_P \cup S_{P'} = S$, and $S_P \cap S_{P'}$. S_P is the set of fanout gates that P drives, and $S_{P'}$ is the set of fanout gates that P' drives. We refer to the division of S into S_P and $S_{P'}$ as a *sink partitioning*, and S_P and $S_{P'}$ as *sink partitions*. All other notations pertaining to G are valid for G'.

For each edge $e = (g_1, g_2)$ in G and G', the Manhattan length of edge e is $dis(e) = |X(g_1) - X(g_2)| + |Y(g_1) - Y(g_2)|$, where $g_1 \in F \cup P \cup P'$, and $g_2 \in P \cup P' \cup S$. Recall that all multi-pin nets will be broken into 2-pin nets with a linear-delay model. For each edge e, edge delay is $D(g_1, g_2) = \tau \cdot dis(e)$. Each edge is also referred as a “net” where g_1 is the driver, and g_2 is the sink.

For gates P and P', we denote their gate delays by $D(P)$ and $D(P')$, respectively. In this chapter, we treat these gate delays as constants. This is fairly accurate since we maintain that buffering must be performed with cloning, and after that, the load of P and P' will remain almost the same. Gate sizing can be performed before or after cloning if the original driver is too weak or strong, which will further control the error of this constant gate delay model.

For a gate g in $P \cup P'$, the required arrival time at the output pin of g is $RAT(g) = \min_{S_i \in S}\{RAT(S_i) - D(P, S_i)\}$, where S is the set of its fanout gates. The arrival time at the output pin of g is $AAT(g) = \max_{F_i \in F}\{AAT(F_i) + D(F_i, P)\} + D(g)$, where F is the set of its fanin gates. The slack of a gate g is $Q(g) = RAT(g) - AAT(g)$.

Without loss of generality, we set gate delays $D(P)$ and $D(P')$ to zero in the following discussion to simplify the analysis. All algorithms are still valid as long as gate delays are constants.

It is easy to see that the slack of P and P' determines the slack of the circuit G and G'. For circuit G, we have $Q(G) = Q(P)$, and for G', we have $Q(G') = \min\{Q(P), Q(P')\}$. For each edge (net) $e = (g_1, g_2) \in E \cup E'$, we define the slack of e as

$$Q(e) = RAT(g_2) - D(g_1, g_2) - AAT(g_1). \tag{6.1}$$

Note that $Q(G') = \min_{e \in E'} Q(e)$ and $Q(G) = \min_{e \in E} Q(e)$.

Cloning Problem: Given a graph $G = (V, E)$, where P is the target gate, RAT for all fanouts S_i, AT for all fanins F_i, and a linear-delay constant τ, create a cloned gate P' for P, which induces a new graph G', find S_P, $S_{P'}$ and locations of P' and P (if P is movable) such that $Q(G')$ is maximized.

In contrast to most previous work which only identifies the partitions S_P and $S_{P'}$, our algorithms will not only provide a partitioning of fanouts, but also the placement of P and P' [6, 7]. If the solution is worse than the original circuit, no cloning will be performed.

6.3 Fast Timing-Driven Gate Cloning

In this section, we present our algorithms for the cases where P is movable and P is fixed. We start with several new concepts.

Best Region and Best Arrival Arc Segment. Recall that the set of fanin gates F is connected to both the original gate P and the duplicate gate P' after cloning. The set of fanout gates S is split into two disjoint sets (partitions) S_P and $S_{P'}$ such that $S = S_P \cup S_{P'}$.

Treat the whole circuit image as a 2D plane H. For each fanin gate F_i in F, the arrival time at any point v in H is $AAT(F_i) + D(F_i, v)$, and $D(F_i, v) = \tau \cdot dis(F_i, v)$. Therefore, if we place a gate at v with the fanin set F, according to static timing analysis

$$AAT(v) = \max_{F_i \in F} \{AAT(F_i) + D(F_i, v)\}.$$

Clearly, $AAT(v)$ is a 2D function, parametrized by the location v. Define the set of points minimizing $AAT(v)$ on the plane H as

$$K(F) = \{a \in H | AAT(a) \leq \underset{\forall v \in H}{AAT(v)}\}.$$

So $K(F)$ is the set of points which have minimum arrival time for all fanins. In the following, we will show that $K(F)$ is either a single point or a line segment with 45° slope. Refer to Figs. 6.2 and 6.3 for examples of $K(F)$.

If there is only a single fanin F, it is obvious that $K(F)$ is the same point as the location of F itself, with $AAT(K(F)) = AAT(F)$. If there are two fanins F_1 and F_2, then there are three cases,

$$K(F) = \begin{cases} \{a \in H | AAT(F_1) + D(F_1, a) = AAT(F_2) + D(F_2, a)\}, \\ \text{ if } |AAT(F_1) - AAT(F_2)| \leq D(F_1, F_2); \\ v_{F_1}, \text{ if } AAT(F_1) > AAT(F_2) + D(F_1, F_2); \\ v_{F_2}, \text{ if } AAT(F_2) > AAT(F_1) + D(F_1, F_2); \end{cases}$$

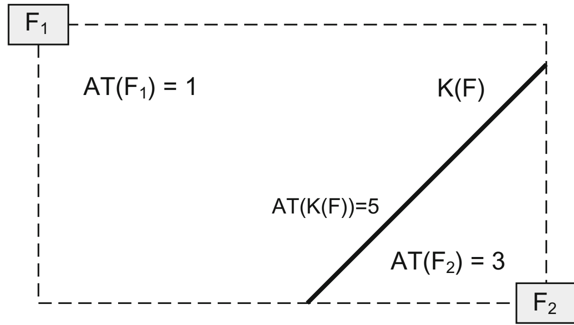


Fig. 6.2 An example of *arrival time arc* K(F). dis(F_1, K(F)) = 4, dis(F_2, K(F)) = 2, $\tau = 1$

Here v_{F_i} refers to the location of fanin gate F_i. In the first case, where the difference between the arrival time at F_1 and F_2 is smaller than $D(F_1, F_2)$, $K(F)$ is a *Manhattan Arc*, which is a segment with slope 45 or −45° in the bounding box of F_1 and F_2. This slope will always be 45 or −45° as long as technology dependent coefficient τ is a constant. Note that when F_1 and F_2 are either horizontally or vertically aligned, $K(F)$ is a point, which is a degenerate case of a *Manhattan Arc*. For the other two cases, where one of the arrival times dominates the other, $K(F)$ is at the location of one of the fanin gates.

Denote the set of points minimizing $AAT(v)$ for fanins $F_1, \ldots, F_i$ by $K(F_i)$. If we have more than two fanins, we will first form $K(F_2)$ for F_1 and F_2, and then merge $K(F_2)$ with F_3 to get $K(F_3)$. $K(F_3)$ will be another *Manhattan Arc* or a single point, depending on the relationship among $AAT(K(F_2))$, $AAT(F_3)$, and $dis(K(F_2), F_3)$ which is the shortest Manhattan distance between F_3 and $K(F_2)$. Repeating this procedure for all fanins, we can find the final $K(F)$. This bottom-up merging process is very similar to the Deferred-Merge Embedding (DME) algorithm in clock tree construction [15] though the goal there is to get a zero skew arc. With a similar procedure to the one shown in [15], it is not hard to prove that $K(F)$ is always a *Manhattan Arc* or a single point, and our merging process guarantees that $K(F)$ will have minimum arrival time for all fanins.

In the rest of this chapter, we denote the *arrival time arc* by $K(F)$ (a point can be considered a degenerate case of an arc), and the arrival time on this arc as $AAT(K(F))$. An example of $K(F)$ for two fanins is shown in Fig. 6.2.

Similarly, we can find $K(S)$, the set of points maximizing $RAT(v)$ on the plane H, for the set of fanouts S. We denote the *required arrival time arc* by $K(S)$ and the required arrival time on this arc by $RAT(K(S))$. Refer to Fig. 6.3 for an illustration of $K(S)$ in an example. With a procedure similar to that in [15], it is easy to prove that computation of $K(F)$ and $K(S)$ takes $O(m)$ time assuming m and n are of the same order. Also, given any order of fanins and fanouts, denote the *arrival time arc* for the set of gates $F_1, \ldots, F_i$ by $K(F_i)$, and the *required arrival time arc* for the set of gates $S_1, \ldots, S_i$ by $K(S_i)$. We can compute all values of $K(F_i)$ and $K(S_i)$ in $O(m)$ time with dynamic programming by incrementally updating and storing all

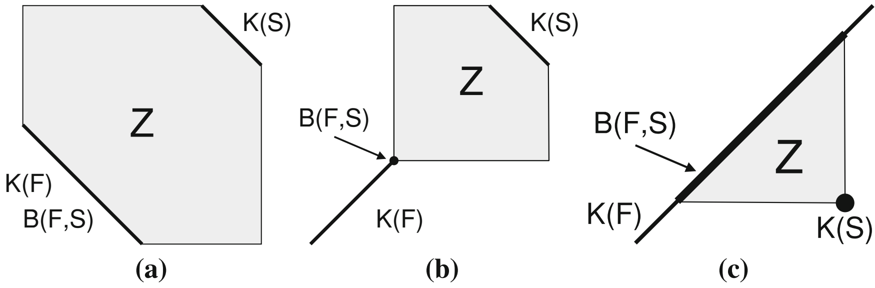


Fig. 6.3 Examples of the region Z. **a** Both $K(F)$ and $K(S)$ are $-45°$ line segments; **b** $K(F)$ is a $45°$ line segment and $K(S)$ is a $-45°$ line segment; **c** $K(F)$ is a $45°$ line segment and $K(S)$ is a single point

arcs. Therefore, the amortized cost for computing each $K(F_i)$ and $K(S_i)$ is constant. We introduce the following lemma to be used in Sect. 6.3.

Lemma 6.1 *It takes $O(m)$ time to compute $K(F)$, $K(S)$, and all values of $K(F_i)$ and $K(S_i)$. The amortized cost of computing each $K(F_i)$ and $K(S_i)$ is $O(1)$.*

The next lemma states that for any point in the plane, its arrival time (required arrival time) can also be represented by the arrival time at $K(F)$ ($K(S)$) and the shortest Manhattan distance between the point and $K(F)$ ($K(S)$). The proof is straightforward, it is based on the merging process and the fact that computation of arrival time (or required arrival time) is a max (min) operation.

Lemma 6.2 *For any point v in the plane H, $AAT(v) = AAT(K(F)) + \tau \cdot dis(K(F), v)$, and $RAT(v) = RAT(K(S)) - \tau \cdot dis(K(S), v)$.*

Now we will introduce the concept of *Best Region*. Define $Z(F, S)$ as a region formed by $K(F)$ and $K(S)$,

$$Z(F, S) = \{v \in H | dis(v, K(F)) + dis(v, K(S)) = dis(K(F), K(S))\},$$

where $dis(v, K(F))$, $dis(v, K(S))$ and $dis(K(F), K(S))$ are the shortest distances between a point or *Manhattan Arc* and another point or *Manhattan Arc*. When $K(F)$ and $K(S)$ are both single points, then $Z(F, S)$ is the rectangle bounding box formed by the two points. Other examples of the region $Z(F, S)$ for different shapes of $K(F)$ and $K(S)$ are shown in Fig. 6.3.

It is easy to show that for any point v outside region Z, it will have $dis(v, K(F)) + dis(v, K(S)) > dis(K(F), K(S))$, and no point exists in H with $dis(v, K(F)) + dis(v, K(S)) < dis(K(F), K(S))$. Also, all points in region Z will have the same slack

$$Q(Z(F, S)) = RAT(K(F)) - AAT(K(S)) - \tau \cdot dis(K(F), K(S)).$$

The following theorem states the slack optimality of the region $Z(F, S)$.

Theorem 6.1 *Given the location of fanin gates F, fanout gates S, if the gate P is placed inside a region $Z(F, S)$ formed with the above process, it achieves the maximum slack.*

Proof If P is located outside of region Z with a bigger slack, then based on Lemma 6.2 we have

$$\begin{aligned} Q(P) &= RAT(P) - AAT(P) \\ &= RAT(K(S)) - AAT(K(F)) - \tau \cdot (dis(K(F), v) + dis(K(S), v)) \\ &< RAT(K(S)) - AAT(K(F)) - \tau \cdot dis(K(F), K(S)) \\ &< Q(Z(F, S)), \end{aligned}$$

which contradicts the assumption.

We refer to region Z as the *Best Region* since it gives the region with the best slack. We also refer to the above procedure to find Z as FIND- BEST- REGION. The runtime complexity of FIND- BEST- REGION is $O(m)$ since the only cost is to compute $K(F)$ and $K(S)$.

Theorem 6.2 FIND- BEST- REGION *finds Best Region Z in $O(m)$ time for a net with m fanouts.*

Now we introduce the concept of *Best Arrival Time Arc*. We define *Best Arrival Time Arc* $B(F, S)$ as the intersection of $K(F)$ and $Z(F, S)$. $B(F, S)$ is part of $K(F)$, while the detailed shape is decided by $K(F)$ and $K(S)$. In examples illustrated in Fig. 6.3, $B(F, S)$ is $K(F)$ in Fig. 6.3a, a single point in Fig. 6.3b, and a partial segment of $K(F)$ in Fig. 6.3c. From Theorem 6.1, we know that every point on $B(F, S)$ still achieves the maximum slack. Define the slack on $B(F, S)$ as $Q(B(F, S))$, and we have $Q(B(F, S)) = Q(Z(F, S))$. In next section, the concept of $B(F, S)$ is used to design our algorithm.

The case of movable original gate. In this section, we present the algorithm for the case when the original gate P is movable. The main idea is to limit the solution search space to $K(F)$, and then find *Best Arrival Time Arc* $B(F, S_P)$ and $B(F, S_{P'})$ efficiently by dividing the plane into six regions (Fig. 6.4) and using the unique properties of fanout slack of each region to find the best locations of P and P'.

When P is movable, we are free to place both P and P'. From Sect. 6.3, given a partitioning S_P and $S_{P'}$, we can simply place P and P' on the best arrival time arc $B(F, S_P)$ and $B(F, S_{P'})$ to achieve the optimal solution. The goal is to find the partitioning, S_P and $S_{P'}$, which gives best slack among all possible partitionings. However, without knowing S_P and $S_{P'}$, $B(F, S_P)$ and $B(F, S_{P'})$ are not apparent.

An important observation is that both arcs must coincide with $K(F)$, which is known. Therefore, rather than trying all partitionings, we will limit our solution search space for both P and P' to $K(F)$, which enables efficient algorithms. This is

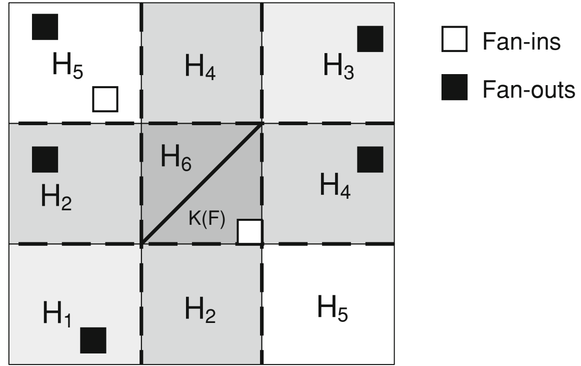


Fig. 6.4 The region division for the *arrival time arc* $K(F)$

the key observation used to derive the partitioning and computation of *best arrival time arcs*. By limiting P and P' on $K(F)$, we have $AAT(P) = AAT(P') = AAT(K(F))$.

Lemma 6.3 *If arrival time arc $K(F)$ is a single point, no cloning is needed.*

Proof If $K(F)$ is a single point, then $B(F, S_P) = B(F, S_{P'})$ is a single point. One can place P at $B(F, S_P)$ and achieve the maximum worst slack without cloning.

As stated in Sect. 6.1, we assume that the capacitive load of the gate is reasonable and no capacitance-based cloning is needed.

Now we discuss the case when $K(F)$ is a *Manhattan Arc*. Since both P and P' are movable, we use P as an example in the following discussion. Without loss of generality, we assume $K(F)$ is a 45° line segment (analysis for the −45° case is similar), as shown in the Fig. 6.4. Denote the lower-left and upper-right endpoint of $K(F)$ as i and j, respectively. The plane H is divided into six regions, H_1, H_2, H_3, H_4, H_5, and H_6, based on i and j as shown in Fig. 6.4. Note that some fanins may be located outside region H_6 as shown in Fig. 6.4 since the arrival time of all fanin gates could be different. One can also refer Fig. 6.2 as an example. For any fanout gate S_i in each region, we analyze the relation between the slack of the edge (net) $e = (P, S_i)$ and the location of P on $K(F)$. Note that $Q(e)$ is purely determined by $RAT(S_i) - D(P, S_i)$ since $AAT(P) = AAT(K(F))$.

Figure 6.5 shows the typical curves of edge slack versus location of P on $K(F)$ for each region. The horizontal coordinate is the distance along the line segment from point i to point j. For example, if a fanout is located in region H_1, then when P is located at i, we will get the maximum slack for this net, and when P is located at j, we will get the minimum slack for this net. When a fanout is located in H_2, then when P is located from i to a certain point on $K(F)$, the slack will stay constant, and begins to decrease when P moves towards j.

If we intersect all slack curves in the set S_P ($S_{P'}$), a minimum slack curve can be generated by taking the minimum slack among all slack curves for each point on $K(F)$. The segment with maximum slack on this new curve will be the best slack

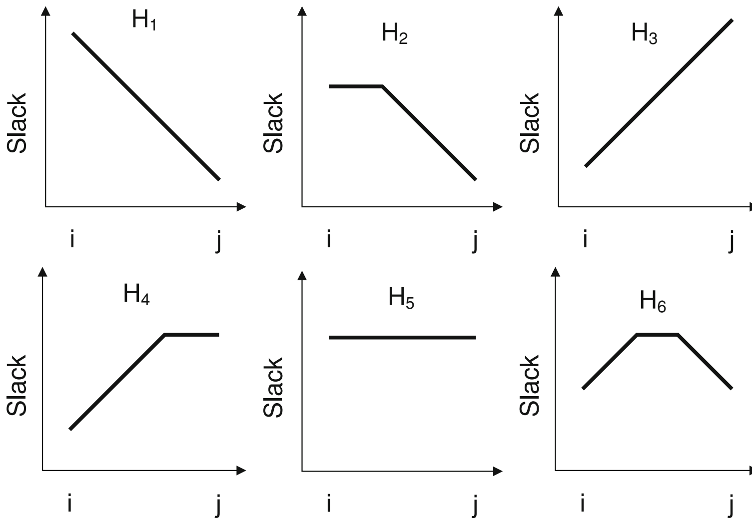


Fig. 6.5 The slack versus $K(F)$ curves for each region

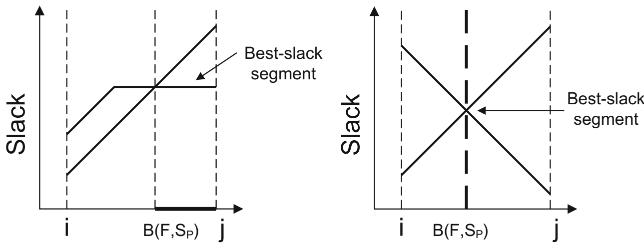


Fig. 6.6 Examples of best slack segment

we can achieve for this set of fanouts. This segment is either a level segment or a single point, as illustrated in Fig. 6.6. Let us refer to this segment as the *Best Slack Segment*. Then the corresponding segment for the *Best Slack Segment* on $K(F)$ is $B(F, S_P)$ ($B(F, S_{P'})$). Clearly, we seek the partitioning with the greatest *Best Slack Segment*, and if we find it, the *Best Slack Segment*, $B(F, S_P)$ and $B(F, S_{P'})$ are also determined. Two examples of *Best Slack Segments* are illustrated in Fig. 6.6.

We now consider two cases: fanouts may be present in region H_6 or absent.

The case when there are no fanouts in region H_6. Consider Fig. 6.5. By putting all fanout gates from H_1 and H_2 in one set (say S_P), and all fanout gates from H_3 and H_4 in another set (say $S_{P'}$), the *Best Slack Segment* for each set is the maximized since it avoids potential intersection (i.e. fanouts from H_1 and H_3). In that case, $B(F, S_P)$ is a line segment on $K(F)$ starting from i, and $B(F, S_{P'})$ is a line segment on $K(F)$ starting from j. Fan-out gates from H_5 can be put in either set and do not affect the results since for every point in region H_5, the distances to all locations on

$K(F)$ are equal. This partitioning is one of the best partitionings and achieves the best slack.

Lemma 6.4 *If both P and P' are movable, and no fanout gates are located in the region H_6, the cloning problem can be solved optimally in $O(m)$ time.*

Proof If we put all fanout gates from H_1 and H_2 in S_P, all fanout gates from H_3 and H_4 in $S_{P'}$, and all fanout gates from H_5 in either set, we have an optimal partitioning. One of the optimal placement solutions places P at i and P' at j. The time complexity is $O(m)$, which is the time of computing $K(F)$. The case when the slope of $K(F)$ is $-45°$ can be proved similarly.

From Lemma 6.4, it follows that

Lemma 6.5 *If P is movable, and no fanout gates are located in region $H_6 \cup H_1 \cup H_2$ (or $H_6 \cup H_3 \cup H_4$), no clone is needed and optimal slack can be achieved by placing P at j (or i).*

Now we present the general algorithm.

The case when there are fanouts in region H_6. A slack curve as shown in Fig. 6.5 for any region H_1, H_2, H_3, H_4, H_5 and H_6 can be regarded as a trapezoid-like curve (referred to as trapezoids for notational convenience henceforth) or a degenerate case (e.g., a line segment) of a trapezoid. Consider a graph containing slack curves corresponding to all fanout gates. In the following, a side of a trapezoid will be called a line segment. The slope of any such line segment is one of 0°, τ or $-\tau$. A 0° line segment in a trapezoid is called a level segment. In the degenerate case where the slack curve is a single line segment, the level segment is defined as the end point with maximum slack.

In all trapezoids, we first find the rightmost τ-slope line segment and the leftmost $-\tau$-slope line segment. For example, the left (right) side of the dotted t_1 (t_2) in Fig. 6.7a shows the rightmost τ-slope (leftmost $-\tau$-slope line segment). The line segment of a trapezoid is rightmost (leftmost) if no line segment of the slope is to the right (left) of the line segment. The leftmost and rightmost line segments can be found in linear time.

First note that any point in a slack curve for fanout S_i refers to the net slack $Q(P, S_i)$ when placing P along i, j as defined in Fig. 6.4. Given a single slack curve t_1, the best slack it can achieve is the slack corresponding to the level segment. To achieve it, one can place the gate anywhere along that level segment.

Case 1: When the rightmost τ-slope line segment and the leftmost $-\tau$-slope line segment do not intersect, as shown in Fig. 6.7b, the lower level segment of all trapezoids, which is the *Best Slack Segment*, determine the maximum worst slack and no gate duplication is needed. Note that in this case, pure line segments in regions H_1, H_2, H_3 and H_4 are considered as well, since they are degenerate cases of trapezoids. One can just place P anywhere on that level segment and this achieves the best slack.

Case 2: When the rightmost τ-slope line segment and the leftmost τ-slope line segment intersect, first find the trapezoids that these two line segments belong to.

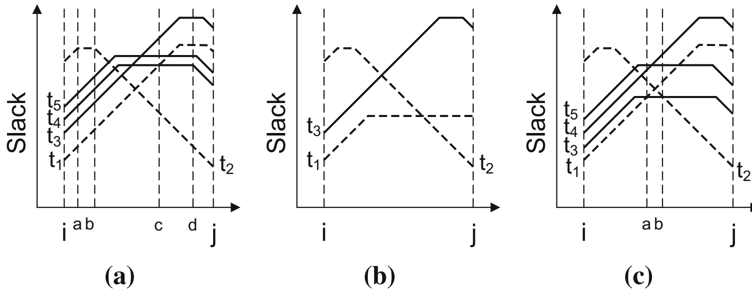


Fig. 6.7 Examples of slack curves versus locations: **a** an example that needs gate duplication; **b** an example in which the rightmost and leftmost segments do not intersect; **c** an example that does not need gate duplication

Without loss of generality, the identified trapezoids are as t_1 and t_2, respectively, in Fig. 6.7a. We compute the intersections of all other trapezoids with t_1 and t_2, and put them into the sets S_P and $S_{P'}$ formed by t_1 and t_2, respectively. All other trapezoids can be divided into three groups.

Group A: For any trapezoid intersecting neither t_1 nor t_2, called a *zero-intersecting trapezoid*, we arbitrarily assign it to a set. The zero-intersecting trapezoids will not impact the worst slack. Note that if all trapezoids other than t_1 and t_2 are zero-intersecting trapezoids, the lowest level segment in each of t_1 and t_2 is the *Best Slack Segment* in each set.

Group B: For any trapezoid intersecting only one of t_1 and t_2, called a *one-intersecting trapezoid*, we can always assign it to the opposite set (formed by the line segment not intersecting with it). For example, the trapezoid t_3 in Fig 6.7a only intersects t_2 and it is assigned to S_P formed by t_1. The one-intersecting trapezoids will not impact the worst slack as long as they are assigned appropriately. Note that if all trapezoids other than t_1 and t_2 are one-intersecting trapezoids, the lowest level segment in each of t_1 and t_2 is the *Best Slack Segment* of each set.

Group C: For any trapezoid intersecting both of t_1 and t_2, called a *two-intersecting trapezoid*, we have two intersecting points. A two-intersecting trapezoid will be assigned to the set containing the higher intersecting point. For example, both t_4 and t_5 are assigned to the S_P formed by t_1. One then needs to find the two-intersecting trapezoid with lowest level segment, such as t_4 in Fig. 6.7a. Subsequently, the lowest level segment in t_1, t_2 and t_4 determines the *Best Slack Segment*. In Fig. 6.7a, the *Best Slack Segment* for P is in t_2. This means that P can be anywhere between a, b and P' can be anywhere between c, d. For the partitioning of the set of fanout gates S, P will connect to S_P which contains all the trapezoids assigned to S_P determined by t_1, and P' will connect to $S_{P'}$ which contains all the trapezoids assigned to $S_{P'}$ determined by t_2. Note that the lowest level segment of a two-intersecting trapezoid can be lower than the intersection of t_1 and t_2, see, e.g., t_5 in Fig. 6.7c. However, it will not impact our algorithm. This just means that one cannot improve the slack by gate duplication since the worst slack is determined by the level segment of t_5.

```
CLONING-MOVABLE
      ▷ Input: Graph G
      ▷ Output: Location of P and P', S_P and S'_P
 1    Find arrival time arc K(F) given the set of fanin gates F
 2    if K(F) is a point
 3       Move P to K(F) and return
 4    Divide the placement region by K(F) into 6 regions
 5    if no S_i in H_6
 6       Compute S_P, S'_P, P, P' according to Lemma 6.4, return
 7    Put all slack curves into a single graph
 8    Find the rightmost τ-slope and leftmost −τ-slope line segment and trapezoids
 9    if they do not intersect
10       Slack is determined by the lower level segment
11    else
12       Compute intersections between remaining trapezoids
               with the above trapezoids and assign them to S_P and S'_P
               accordingly. Compute P and P' as above.
13    return the location of P and P', S_P and S'_P
```

Fig. 6.8 Our simultaneous cloning and placement algorithm for a movable gate

The algorithm is optimal since the above two cases cover all possible situations and in each situation, it is easy to see that the optimal solution is computed. In the algorithm, one needs to first compute the rightmost τ-slope and the leftmost $-\tau$-slope line segment. If they do not intersect, the slack is determined by the lower level segment. Otherwise, for each of the remaining $m - 2$ trapezoids, compute its intersections with t_1 and t_2. Assign the trapezoids to partitions accordingly based on their groups. For a two-intersecting trapezoid, one also needs to record its higher intersection point. Next, find the trapezoid with lowest higher intersecting point (e.g., t_4 in Fig. 6.7a), which takes linear time. One can then immediately find the maximum possible worst slack the circuit can achieve by comparing it with the level segment of t_1 and t_2. The above algorithm runs in linear time.

Theorem 6.3 *The optimal cloning can be computed in O(m) time if the original gate is movable.*

Pseudo-code of the algorithm appears in Fig. 6.8.

The case of fixed original gate. When the original gate P is fixed, the algorithm in Fig. 6.8 does not work since we can not expect P to be placed on the *arrival time arc* $K(F)$. Let us assume all fanouts in S are sorted in a non-increasing order of $RAT(S_i) - D(P, S_i)$.

Lemma 6.6 *There are at most m unique* $Q(P)$ *values if* P *is fixed.*

Proof Since P is fixed, $AAT(P)$ and $D(P, S_i)$ are constant. Then for all possible partitionings, $Q(P)$ can only be one of the values among $RAT(S_1) - D(P, S_1) - AAT(P), RAT(S_2) - D(P, S_2) - AAT(P), \ldots, RAT(S_m) - D(P, S_m) - AAT(P)$.

Fig. 6.9 Our simultaneous cloning and placement algorithm for a fixed gate

```
CLONING-FIXED
   ▷ Input: Graph G, Original Slack Q_ori
   ▷ Output: Location of P', S_P and S'_P
1  SORT S in non-increasing order of RAT(S_i) − D(P, S_i)
2  for i = 1 to m-1
3     S_P = {S_1 ... S_i}, S'_P = S − S_P
4     Call FIND-BEST-REGION to get the location of P'
          Compute Q(P) and Q(P')
5     if Q(P') ≥ Q(P)
6          break
7  Compare the solution with Q_ori and
      return the location of P', S_P and S'_P
```

The above lemma states that if fanout S_i is in S_P, then we can put all fanouts S_j, where $j < i$ into S_P, and $Q(P)$ does not change. With Lemma 6.6, we can start with putting S_1 in $S(P)$, while putting all other gates in $S(P')$, and get the worst slack of $Q(P)$ and $Q(P')$. If $Q(P') \geq Q(P)$, we can stop since we have found the possible best slack. If not, we can put S_1 and S_2 in $S(P)$, which will decrease $Q(P)$, but may increase $Q(P')$. Again, if $Q(P') \geq Q(P)$, this will be the best possible slack since further additions to $S(P)$ can only decrease $Q(P)$. The pseudo-code of the algorithm is shown in Fig. 6.9.

The sorting of S takes $O(m \log m)$ time. After S is sorted, we can compute all $K(S_{P'})$ for all possible m cases in $O(m)$ time based on Lemma 6.1. Each FIND-BEST- REGION then takes $O(1)$ time since $AAT(K(F))$ is a constant and $K(S_{P'})$ has been precomputed. The total complexity is $O(m \log m)$.

An interesting corollary is that one solution to this problem involves disconnecting all sinks from P and letting the cloned gate P' drive all fanouts, then placing P' optimally. If permissible, this case is similar to RUMBLE (see Chap. 3), and we can compare the solution with the above results and choose the best one. If this is undesirable behavior, we can constrain the solution to include at least one sink driven by P.

Theorem 6.4 *The optimal cloning can be computed in $O(m \log m)$ time if the original gate is fixed.*

6.4 Empirical Validation

To show the effectiveness of cloning and compare it to other optimizations, we first create 100 testcases at the 45 nm process node. We randomly created circuits with different fanins and fanouts and placed them in a region with the bounding box size ranging from 1 to 15 mm. The number of fanins range from 2 to 4, and the number of fanouts range from 2 to 8. We choose 16 buffers and inverters for the buffer insertion.

We implemented four different optimizations including cloning as follows, to show the benefit of our techniques. They are

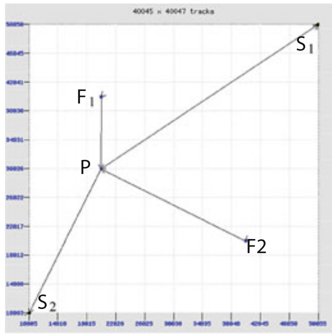
- Buffering: Timing-driven buffer insertion [16]. This optimization is treated as the baseline to which all other optimizations are compared.
- RUMBLE: Moving the original gate and rebuffering as described in Chap. 3.
- Clone1: Our cloning algorithm when the original gate is fixed.
- Clone2: Our cloning algorithm when both the original and duplicated gates can be moved.

Before the optimizations RUMBLE, Clone1, and Clone2, we always perform buffer insertion to fix slew rate violations and begin with reasonable timing. The results are also compared to buffer insertion results (which means Buffering is the baseline). This is to guarantee that any improvement we see from our techniques is due to cloning instead of merely buffering the original net. In addition, we also use the RUMBLE algorithm inside our cloning algorithms to determine the best gate location after a partitioning is fixed. For each partition, we will perform RUMBLE to find the gate location and slack, and then choose the best solution for all partitions derived from our algorithm. Note that this is only for comparison purposes, and one can apply our algorithm first to find the best partitioning and only apply the RUMBLE algorithm once.

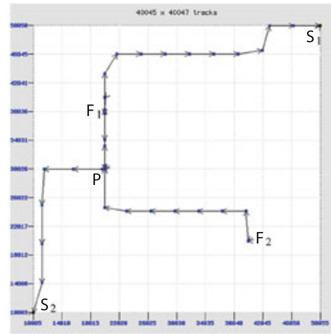
All algorithms including buffering and RUMBLE are implemented in C++ and tested on an AMD Opteron computer with 2.8 GHz CPU and adequate memory. For cloning, we apply all optimization steps, including ripping up buffer trees for the circuits, duplicating and placing the gates, re-buffering and legalization. For RUMBLE, we also rip up buffer trees and place the original gate in the new location. We use an industrial static timing analysis (STA) engine for timing analysis. For rebuffering, we implement the buffering algorithm in [16] to get the best timing-area trade-off, and the buffer tree is constructed to be placement-congestion aware.

To clearly illustrate the impact of each optimization, we first choose one circuit and show its layout after each optimization from Fig. 6.10b–d, where Fig. 6.10a shows the original circuit without buffering. The Manhattan distance between S_1 and S_2 is 13 mm. The timing information after each optimization algorithm is shown in Table 6.1. It clearly shows the benefit of the Buffering, RUMBLE, Clone1 and Clone2 approaches. Clone2 gives the best results in terms of worst slack and total negative slack. Clone1 is still better than RUMBLE and achieves the same worst slack as Clone2, but can not do better for S_2. RUMBLE achieves better slack than pure buffering by placing the original gate in the middle, however, it sacrifices the slack at S_1 for S_2. Note that the slack of S_1 and S_2 are not exactly the same for RUMBLE and Clone2. This is explained by slew rate differences; the buffering topology chosen by the placement congestion aware buffer-tree algorithm considers placement density, as well as the order of buffer insertion for all the nets which results in asymmetric timing constraints.

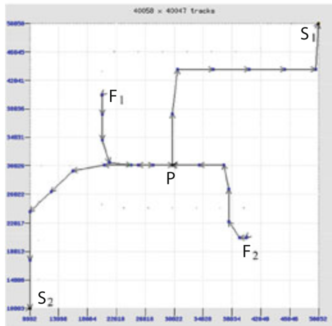
For the rest of the circuits, we list the top 10 circuits with the best improvement due to cloning with detailed information. The results are shown in Table 6.2. For all experiments, we present worst slack (WSLK) improvement over “Buffering”, total



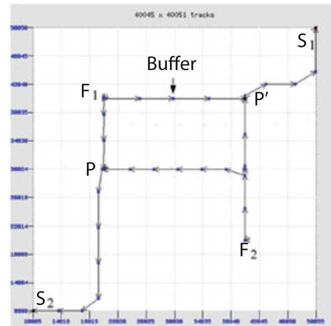
(a) Originalcircuit.



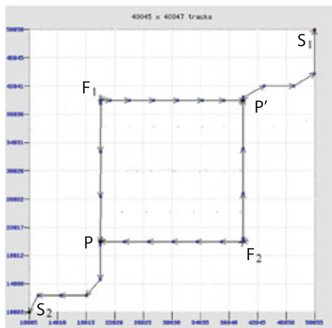
(b) Newcircuitafterbufferinsertion.



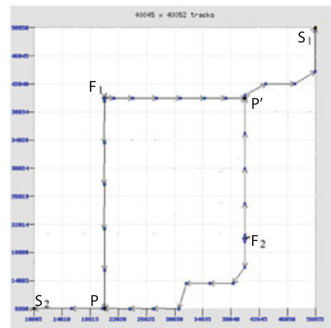
(c) Newcircuitaftergatereplacing andbufferinsertion(RUMBLE,see Chapter3).



(d) Newcircuitaftercloningandre-placingnewgateonly(Clone1).



(e) Newcircuitaftercloningandre-placingbothgates(Clone2).



(f) NewcircuitafterClone2with wirelength-suboptimalsolution.

Fig. 6.10 Examples of different optimizations, including buffering, RUMBLE and cloning. F_1 and F_2 are fanins with same arrival time and S_1 and S_2 are fanouts with same required arrival time. P is the original gate, and P' is the new duplicated gate. **a** Original circuit. **b** New circuit after buffer insertion. **c** New circuit after gate replacing and buffer insertion (RUMBLE, see Chap. 3). **d** New circuit after cloning and re-placing new gate only (Clone1). **e** New circuit after cloning and re-placing both gates (Clone2). **f** New circuit after Clone2 with wirelength-suboptimal solution

Table 6.1 Experimental results comparing cloning to other optimization techniques for the circuit shown in Fig. 6.10

Optimization	Slack at S_1 (ns)	Slack at S_2 (ns)
Buffering (Fig. 6.10b)	−2.855	−2.206
RUMBLE (Fig. 6.10c)	−2.410	−2.403
Clone1 (Fig. 6.10d)	−1.606	−2.076
Clone2 (Fig. 6.10e)	−1.606	−1.590

negative slack (TNS, the sum of all negative paths) improvement over "Buffering", final area and wirelength, where Buffering serves as the baseline. The area includes the original fanin gates, fanout gates, cloned gate and buffering area. We also list the summary results of all 100 circuits in Table 6.2 by averaging all metrics. The runtime for all testcases is less than 5 s, including all static timing analysis, buffer insertion, linear programming inside RUMBLE, I/O processing and model build time.

The table clearly shows the same trend as shown in Fig. 6.10. In terms of worst slack, Clone1 and Clone2 are better than RUMBLE, which is better than buffering. Clone2 gives the best timing results in general, although with the cost of area and wirelength increase. We also notice that for all cases, Clone1 and Clone2 both achieve better TNS improvement than buffering. Note that our algorithms may not get the best TNS, especially Clone1, which does not entail movement of the original gate. The summary data show that Clone2 and Clone1 still outperform RUMBLE and buffering on average.

6.5 Extensions

Our algorithms naturally accommodate several additional objectives that we briefly summarize in this section.

Wirelength optimization. Note that in our formulation, we do not directly consider wirelength. However, our approach can be extended to consider wirelength while not sacrificing slack. For example, in the case where both gates are movable and no gates are placed in region H_6, after we determine the partitioning, and put P at i and P' at j, we can still find the best region Z which is bounded by i and S_P for P (similarly for P' with a region bounded by j and $S_{P'}$). When region Z is not a single point, it may be possible to find a solution with same slack but better wirelength. We briefly summarize the $O(m^3)$-time algorithm as follows. Consider the Hanan grid H composed of the coordinates of all fanins and fanouts of some gate P. Each rectangular region of H will have some distinct function of wirelength in terms of the location of P. Begin by finding the slack-optimal region Z for the gate P. Then iterate over all regions R of H and compute the minimum wirelength value for locations in $R \cap Z$. Skip this region if $R \cap Z = \emptyset$. Record the best wirelength for each region R of H, then choose the best wirelength solution among all recorded.

Table 6.2 Experimental results comparing cloning to other optimization techniques for 100 circuits

Ckt	Transforms	WSlk (ns) improvement	TNS (ns) improvement	Area	WL
1	Buffering	0	0	1158	181960
	RUMBLE	1.548	3.630	609	117637
	Clone1	1.553	3.645	799	117632
	Clone2	1.581	3.747	601	141977
2	Buffering	0	0	1110	166546
	RUMBLE	1.111	2.895	859	162461
	Clone1	1.175	3.091	889	162419
	Clone2	1.542	4.660	1026	164254
3	Buffering	0	0	942	142242
	RUMBLE	0.956	1.859	722	131794
	Clone1	1.030	2.298	850	145908
	Clone2	1.073	2.611	765	135896
4	Buffering	0	0	709	95520
	RUMBLE	1.050	1.113	636	88441
	Clone1	1.022	1.092	636	88441
	Clone2	1.050	1.113	636	88441
5	Buffering	0	0	1758	253393
	RUMBLE	0.839	6.128	1120	194261
	Clone1	0.814	6.262	1109	194260
	Clone2	1.028	5.413	1410	241818
6	Buffering	0	0	1604	225577
	RUMBLE	0.773	3.282	998	177139
	Clone1	1.014	1.041	1529	241626
	Clone2	1.017	2.152	1293	233053
7	Buffering	0	0	1781	268237
	RUMBLE	0.302	0.189	1583	257903
	Clone1	0.830	1.049	1990	315835
	Clone2	0.815	1.121	2047	330826
8	Buffering	0	0	1578	227047
	RUMBLE	0.262	4.270	1153	195097
	Clone1	0.681	2.118	1836	272108
	Clone2	0.732	4.866	1633	251854
9	Buffering	0	0	998	140556
	RUMBLE	0.685	1.512	861	122344
	Clone1	0.687	1.514	848	122411
	Clone2	0.718	1.530	871	122360
10	Buffering	0	0	998	159705
	RUMBLE	0.269	1.312	831	140127
	Clone1	0.672	1.759	916	150529
	Clone2	0.673	1.754	899	150490
Average of 100 circuits	Buffering	0	0	1407	205891
	RUMBLE	0.192	0.797	1337	198247
	Clone1	0.279	1.050	1472	216617
	Clone2	0.309	1.267	1471	220089

Buffering timing-driven buffering, *RUMBLE* timing-driven gate placement followed by buffering, *Clone1*, gate duplication with the original gate fixed, *Clone2* gate duplication with the original gate movable

Because this coordinate is within Z it is guaranteed to have the optimal slack, and because we exhaustively searched H, it is guaranteed to have the best wirelength of all locations within Z. Note that the wirelength optimal region may be contained within Z, in which case the wirelength optimal solution is also slack optimal. This algorithm runs in $O(m^3)$-time because there are $O(m^2)$ rectangular regions within H and evaluating each region requires $O(m)$ time for the wirelength calculation.

A wirelength-suboptimal Clone2 example is shown in Fig. 6.10f. It has the same slack and TNS as Fig. 6.10e, however, Fig. 6.10e clearly shows smaller wirelength (and fewer buffers), and it can be proved that the location of P in Fig. 6.10e is a wirelength optimal solution.

TNS Optimization. Though our algorithms can improve the TNS objective (see Eq. 5.2) by improving worst slack, our algorithms do not directly optimize the TNS objective. It can, for example, hurt TNS by reducing slack on two paths, while seeking to improve the slack on a third worst-slack path. In the late stages of the flow, this may be unacceptable, and we may wish not to harm TNS, or to directly optimize TNS or the number of negative paths. When both gates are movable and there is no fanout in region H_6, it is easy to prove that our solution gives the best solution in terms of TNS. When there are gates in region H_6, one can tune the algorithm CLONING- MOVABLE to be TNS aware. When we assign trapezoids, even if it does not change worst slack, we can assign based on its own slack and achieve better TNS. Finally, we can prevent harm to the TNS objective by incorporating it into the acceptance criteria for any cloning solution.

Placement Blockages. When there are placement blockages in the design, such as IP, macros, or high-gate-density regions, one may not be able to place gates in optimal locations. Our algorithms can be extended to handle blockages as follows. When the best region Z is not a single point, and not completely blocked by placement blockages, we place P (or P') in the region inside Z with free space and still achieve the optimal slack. If Z is completely blocked, then P is placed at the legal location with the minimum Manhattan distance to the region Z.

6.6 Conclusions

This chapter revisits timing-driven cloning under a linear interconnect-delay model that accounts for buffering during physical synthesis. We present several highly efficient algorithms for timing-driven cloning to optimize the worst slack of a circuit. The primary contribution of this work is an optimal method for simultaneously determining which sinks will be driven by the which copy of a gate, as well as the locations of a gate and its replica under the given delay model. We also describe several extensions to the algorithm for accommodating additional objectives. Our empirical results demonstrate improved circuit performance as a result of increased optimization scope.

References

1. Kužnar R, Brglez F (1995) PROP: a recursive paradigm for area-efficient and performance oriented partitioning of large FPGA netlists. In: ICCAD, pp 644–649
2. Hwang J, El Gamal A (1992) Optimal replication for min-cut partitioning. In: ICCAD, pp 432–435
3. Chen G, Cong J (2005) Simultaneous timing-driven placement and duplication. In: ISFPGA, pp 51–59
4. Kim H, Lillis J, Hrkic M (2006) Techniques for improved placement-coupled logic replication. In: GLSVLSI, pp 211–216
5. Chen C, Tsui C (1999) Timing optimization of logic network using gate duplication. In: ASP-DAC, pp 233–236
6. Lillis J, Cheng CK, Lin TY (1996) Algorithms for optimal introduction of redundant logic for timing and area optimization. In: ISCAS, pp 452–455
7. Srivastava A et al (2001) On the complexity of gate duplication. IEEE Trans CAD 20(9):1170–1176
8. Srivastava A et al (2004) Timing driven gate duplication. IEEE Trans VLSI 12(1):42–51
9. Saxena P, Menezes N, Cocchini P, Kirkpatrick DA (2004) Repeater scaling and its impact on CAD. IEEE Trans CAD 23(4):451–463
10. Bañeres D, Cortadella J, Kishinevsky M (2007) Layout-aware gate duplication and buffer insertion. In: DATE, pp 1367–1372
11. Alpert CJ et al (2006) Accurate estimation of global buffer delay within a floorplan. IEEE Trans TCAD 25(6):1140–1146
12. Otten R (1998) Global wires harmful. In: ISPD, pp 104–109
13. Luo T, Papa DA, Li Z, Sze CN, Alpert CJ, Pan DZ (2008) Pyramids: an efficient computational geometry-based approach for timing-driven placement. In: ICCAD, pp 204–211
14. Shi W, Li Z, Alpert CJ (2004) Complexity analysis and speedup techniques for optimal buffer insertion with minimum cost. In: ASP-DAC, pp 609–614
15. Chao TH et al (1992) Zero skew clock routing with minimum wirelength. IEEE Trans CAS 39(11):799–814
16. Li Z, Sze CN, Alpert CJ, Hu J, Shi W (2005) Making fast buffer insertion even faster via approximation techniques. In: ASP-DAC, pp 13–18

Chapter 7
Logic Restructuring as an Aid to Physical Retiming

The impact of physical synthesis on design performance is increasing as process technology scales. Current physical synthesis flows generally perform a series of individual netlist transformations based on local timing conditions. However, such optimizations lack sufficient perspective or scope to achieve timing closure in many cases. To address these issues, we develop an integrated transformation system that performs multiple optimizations simultaneously on larger design partitions than existing approaches. Our system, SPIRE, combines physically-aware register retiming, along with a novel form of logic cloning and register placement. SPIRE also incorporates a placement-dependent static timing analyzer (STA) with a delay model that accounts for buffering and is suitable for physical synthesis.

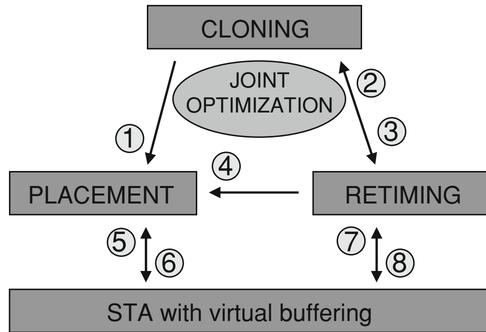
7.1 Introduction

Recall from Chap. 2 that the physical synthesis process begins by computing a tentative cell placement and proceeds to restructure timing-critical paths. Traditional physical-synthesis flows in the industry [1, 2] apply a series of local, mostly greedy transformations such as inserting individual buffers on particular nets, or relocating individual gates in the limited context of their neighboring gates. Several iterations of such transformations may be required for timing closure [1, 2]. However, growing reliance on physical synthesis for timing closure motivates the development of transformations that are more powerful in two specific ways.

- **Greater optimization scope**: the ability to effect larger changes in the circuit in terms of simultaneously moving or altering several objects in order to achieve timing closure.
- **Larger optimization window size**: the ability to consider temporal and spatial constraints from partitions of a design.

Increasing the optimization scope and window sizes can help avoid local minima in the solution space that trap individual, local transformations. Additionally,

D. A. Papa and I. L. Markov, *Multi-Objective Optimization in Physical Synthesis of Integrated Circuits*, Lecture Notes in Electrical Engineering 166,
DOI: 10.1007/978-1-4614-1356-1_7, © Springer Science+Business Media New York 2013



1:	CLONING changes the netlist and influences PLACEMENT
2:	RETIMING helps select combinational gates for CLONING
3:	CLONING creates new opportunities for RETIMING (see Fig. 7.2)
4:	RETIMING relocates netlist registers, causing new PLACEMENT
5:	PLACEMENT changes interconnect delays used in STA
6:	Register PLACEMENT after retiming is performed based on STA
7:	RETIMING relocates netlist registers, changing paths in STA
8:	STA computes min slack — the optimization goal for RETIMING

Fig. 7.1 Interactions between optimizations in SPIRE's joint optimization

this circumvents the *ordering problem* of individual transformations, since different sequences can yield different results.

We facilitate more powerful optimizations through retiming. Unlike traditional gate- and net-centric timing optimizations that aim to satisfy given stage-timing constraints, retiming can optimize the constraints themselves to better fit a given netlist. Therefore, we propose a **S**ystem for **P**hysically-aware **I**ncremental **R**etiming and **E**nhancements, or *SPIRE*, that performs register-retiming with accurate delay models, buffering, placement, and logic cloning to seamlessly integrate retiming into physical synthesis. Key features of SPIRE are:

- Multiple degrees of freedom to optimize the circuit, including *gate placement*, *register retiming*, and *gate cloning*.
- A mixed-integer linear programming (MILP) framework for joint optimization that emphasizes synergies between point optimizations as shown in Fig. 7.1.
- An embedding of placement-dependent STA computations with virtual buffering into the MILP, which allows for efficient and accurate consideration of timing constraints from large design partitions.

SPIRE allows for placement, retiming, and cloning to simultaneously optimize a circuit, as shown in Fig. 7.1. In physical synthesis, such a joint optimization problem is often considered intractable. Instead, one chains individual optimizations with limited scope. However, as shown in Fig. 7.2, such *separation of concerns* overlooks opportunities for joint optimization. Therefore, we propose a powerful transformation that is computationally expensive, but can be applied to sizable circuit *windows*.

Window sizes can be selected subject to runtime constraints imposed on the system. Our experimental results in Sect. 7.4, in fact, show that SPIRE can handle window sizes of thousands of gates by efficiently encoding the problem as an MILP with linearly many constraints in the size of the circuit.

Retiming methods based on [3] enforce timing constraints by requiring a register on every path whose delay exceeds a threshold. However, such methods require computationally-expensive path enumeration within the linear programming formulation. We avoid path enumeration by enforcing linearly many conditional STA-like constraints which determine optimal retiming and placement. Further, different choices for retiming, cloning and gate relocation perturb only a small set of local constraints directly (those affecting nearby edges). Aside from the system as a whole, we highlight the following contributions of this work:

- A method for retiming with an accurate embedded STA-like delay computation.
- A novel gate-cloning technique to create opportunities for retiming.
- A simultaneous retiming and re-placement technique.

The remainder of this chapter is organized as follows. Section 7.2 reviews background and notation. Section 7.3 presents our maximum-slack retiming formulation that incorporates STA, placement, and cloning. In Sect. 7.4, our methods are validated on a 45 nm high-performance microprocessor against leading-edge physical synthesis tools. Section 7.5 outlines additional optimizations that can further increase the scope of SPIRE. Conclusions are drawn in Sect. 7.6.

7.2 Background, Notation and Objectives

We now review background in static timing analysis and period-constrained retiming.

Static timing analysis with buffered wires. SPIRE depends on the ability to encode timing constraints efficiently, and in such a way that they can be easily adjusted to accommodate changes resulting from circuit optimizations. Static timing analysis relies on models to compute the delays of gates and nets. For example, it is common to use a look-up table to represent gate delays in terms of its inputs. In advanced CMOS technologies, buffering is utilized heavily during physical synthesis to reduce wire delay and improve timing. Therefore, it is important to estimate buffered wire delay in an interconnect delay model. In SPIRE we efficiently accommodate these considerations by using constant gate delays that are obtained from look-up-table-based delay models, and by using a linear interconnect-delay model introduced in Chap. 3. These assumptions allow the constraints represented in SPIRE to be in terms of a local neighborhood, and are thus only linear in number (assuming constant maximum edge and vertex degree).

To compute the initial conditions for SPIRE, the RAT and AAT of all fixed timing points are generated by an STA engine using very accurate delay models and a set of timing assertions created by designers [4, 5]. SPIRE considers the timing of register's

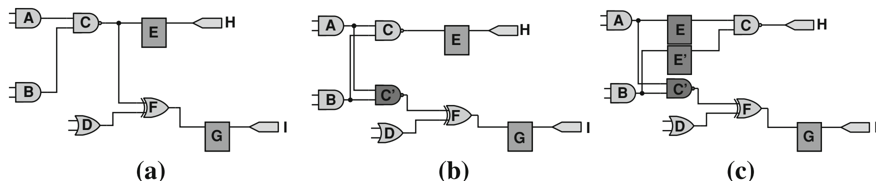


Fig. 7.2 Retiming and gate cloning to improve slack: **a** Register E cannot be moved past gate C because of fanout E–F. **b** If the NAND gate C is cloned, creating a new gate C' to drive its two sinks, it is possible to retime the top register without changing the logic function. **c** The final result with register E retimed

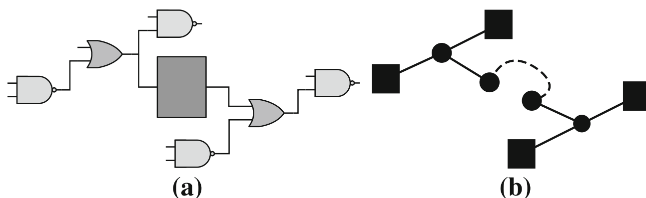


Fig. 7.3 A circuit (**a**) and its timing graph (**b**). The square objects have fixed AATs or RATs. STA is performed only on circular movable objects

input pin fixed and uses a static timing engine to determine its RAT value. Similarly, the AAT is fixed on output pin of a register. The timing analysis engine includes considerations of setup and hold time.
The timing metrics that we optimize include the minimum slack of all vertices ($\mathscr{M}$), the total negative slack in the circuit ($\mathscr{T}$), and the total slack below a threshold (Θ_T), computed as shown below. Note that $\mathscr{T} = \Theta_0$.

$$\mathscr{M} = \min_u \mathscr{S}(u) \tag{7.1}$$

$$\mathscr{T} = \sum_u \min(0, \mathscr{S}(u)) \tag{7.2}$$

$$\Theta_T = \sum_u \min(0, \mathscr{S}(u) - T) \tag{7.3}$$

In SPIRE, registers are allowed to move, while combinational gates remain *fixed* in place; this limitation is not inherent, as discussed in Sect. 7.5. After gate cloning (Sect. 7.3), the cloned gates can be physically relocated. For efficiency, we restrict our timing graph edges to those representing (1) each connection between the movable gates, and (2) each connection between a movable gate and a fixed gate. For the subcircuit in Fig. 7.3a, the resultant timing graph is shown in Fig. 7.3b.

Register retiming. The original linear programming formulations for minimum-period and minimum-area retiming were developed by Leiserson and Saxe [3].

Fig. 7.4 An LP for minimum-area retiming

```
Minimize
  Σ(u,v)∈E w(u,v) − r(u) + r(v)
subject to
  ∀(u,v) ∈ E, r(u) − r(v) ≤ w(u,v)
```

Fig. 7.5 An LP for min-area, period-constrained retiming

```
Minimize
  Σ(u,v)∈E w(u,v) − r(u) + r(v)
subject to
  ∀(u,v) ∈ E, r(u) − r(v) ≤ w(u,v)
  ∀(u,v) ∈ E|D(u,v) > P, r(u) − r(v) ≤ W(u,v) − 1
```

In their framework, a circuit is represented by a *retiming graph* $G(V, E)$, where each vertex $v \in V$ represents a combinational gate, and each edge $(u, v) \in E$ represents a connection between a driver u and sink v. An edge is labeled by a weight $w(u, v)$, indicating the number of registers (flip-flops) between u and v. The objective of minimum-area retiming is to determine labels $r(v)$ for each vertex v, denoting the number of registers that are moved from the outputs to the inputs of v, that minimize the total sum of edge weights. The weight of an edge after retiming is given by:

$$w_r(u, v) = w(u, v) - r(u) + r(v) \tag{7.4}$$

Therefore, the total number of registers in the retimed circuit can be minimized in terms of the following expression.

$$\sum_{(u,v)\in E} w(u, v) - r(u) + r(v) \tag{7.5}$$

Additionally, retiming labels have to meet *legality* constraints, $w(u, v) \geq r(u) - r(v)$ for each edge, to enforce the fact that edges cannot have negative weights. A linear program for the minimum-area retiming problem is given in Fig. 7.4. Leiserson and Saxe [3] observe that this problem is the dual of a min-cost network flow problem and can therefore be solved in polynomial time.

As shown in Fig. 7.5, the period can be constrained in this formulation by requiring weight ≥ 1 on every path between two vertices with delay exceeding target period P. However, this formulation requires $\Theta(|V|^2)$ constraints in the form of matrix D that stores the delay of the longest path between the vertices (u, v) in $D(u, v)$, and matrix W that stores the weight of that path. Then, a binary search is performed to determine the minimum achievable clock period. The feasibility of each period according to the legality constraints is checked using the Bellman–Ford algorithm [3].

Prior work in retiming also includes the ASTRA [6] algorithm, which is a faster approach. It relates the problem of clock skew optimization at each flip-flop to a retiming solution for min-period retiming, and uses the Bellman Ford algorithm to derive the longest path. Recently, the authors of [7] used program derivation to

automatically generate an algorithm for min-period retiming. Retiming was also explored for slack budgeting and power minimization for FPGAs [8].

Challenges in min-period retiming. Algorithms based on techniques from [3] enforce timing constraints by requiring registers on gate-to-gate paths that exceed a length threshold. This involves computationally expensive enumeration of such paths. Therefore, in our approach we avoid path enumeration by using slack, rather than period as a metric. Slack constraints are linear in the size of the circuit and all path delays are implicitly encoded through the AAT and RAT constraints.

Other retiming algorithms use network-flow based approaches which are difficult to extend to a multi-objective optimization [6]. Using interconnect delays instead of lengths has been a challenge, as wires can be dynamically re-buffered when their lengths change [9]. Unlike much of past literature, we use a buffered delay model to account for this.

Inherent limitations of retiming are associated with multi-fanout branches. To move a register backward through a gate, all fanout branches of the gate must include (or share) a register, and all these registers must be retimed at once. This constraint ensures that the number of registers on any PI-to-PO path stays constant during retiming. Therefore, fanouts can be a bottleneck for retiming. In order to alleviate this problem, we clone gates within the retiming formulation so as to provide additional backward-movement opportunities for registers (see Fig. 7.2).

7.3 Joint Optimization for Physical Synthesis

This section introduces the SPIRE system which combines several optimizations used individually in the past literature. As shown in Fig. 7.6, combining retiming and placement is better than applying them individually. In this example, only the combined approach closes timing. The main difficulty in combining placement, cloning and retiming is their inter-dependence—optimal locations and cloned configurations depend on the timing constraints which are altered by retiming.

Embedding the STA backplane into an ILP. In order to incorporate STA into SPIRE, we first encode the RAT and AAT variable computations into an MILP, with constraints corresponding to actual arrival time and required arrival time calculations, both of which are linear. Then, alternative constraints are introduced to analyze each timing arc, for the case where a register is between the source and sink of the arc. Figure 7.7 shows an LP simply for computing the worst-case slack. For circuit C with gates $G = \{u_1, u_2 \dots u_n\}$, and registers $R = \{l_1, l_2, \dots l_m\}$, the variables in this program are:

- AAT and RAT for each $u \in G$, denoted A_u, and R_u.
- $\mathscr{M}$ for the minimum slack.

In other words, for a gate u driven by $i_1, i_2, \dots i_S$ the constraints to enforce A_u are shown below. Here $1 \leq j \leq S$:

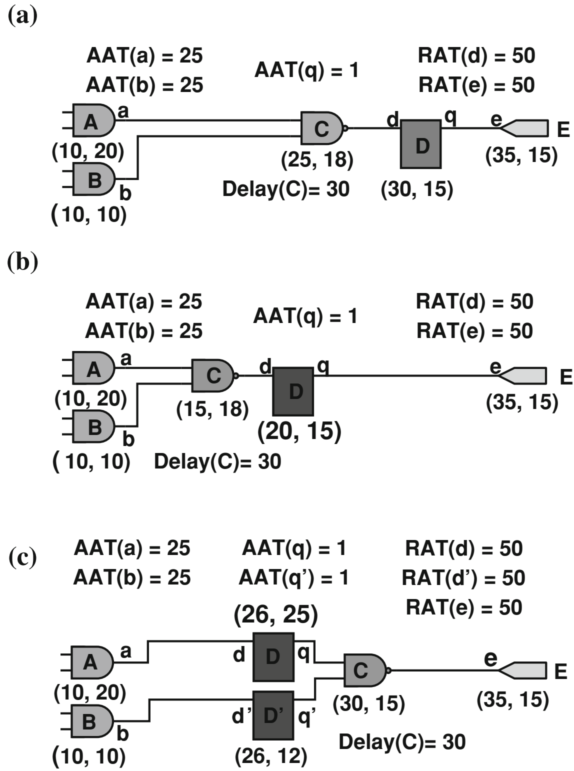


Fig. 7.6 Advantages of performance-driven retiming with simultaneous re-placement. Timing values of labeled pins are given, and physical locations of gates and ports are shown as (x, y) pairs. In the original circuit (**a**), the timing path feeding the input of the register has negative slack. Moving the gate and register in (**b**) improves the slack, but movement alone does not allow the path to meet timing constraints. Only by retiming and movement can all timing constraints be met in (**c**)

$$A_u \geq A_{i_j} + \tau * \text{HPWL}(i_j, u) + D_u \tag{7.6}$$

Since A_u must actually be equal to one of the values in Eq. 7.6, it is added to the objective function so that it can be minimized. The constraints guarantee that it will be greater than any path's delay. Adding it to the objective guarantees that it will be no more than the greatest path delay. Similarly for R_u, supposing that u drives gates $o_1, o_2, \ldots o_T$, then the corresponding constraints are of the form for $1 \leq k \leq T$:

$$R_u \leq R_{o_k} - \tau * \text{HPWL}(g, o_k) - D_u \tag{7.7}$$

We subtract RAT(u) from the objective function since this variable is maximized rather than minimized. The AAT and RAT of registers (and other end points like primary input and output pins) are simply set according to initial values obtained form the reference timing model. The term $-\mathscr{M}$ is added to the minimization objective. The total slack $\mathscr{T}$ can also easily be computed from the MILP and added as an objective. In practice, we minimize both. However, for brevity, we drop $\mathscr{T}$ from the MILP formulations for the remainder of the chapter. Note that the number of constraints in this formulation is proportional to the number of 2-pin arcs in the

Fig. 7.7 A linear program that calculates the minimum slack

Objective Minimize : $-\mathscr{M}$ $+\forall(\mathrm{u})(\mathrm{A_u}-\mathrm{R_u})$
`subject to` $\forall \mathrm{u} \;\; \mathscr{M} \leq \mathscr{S}(\mathrm{u})$ $\forall \mathrm{u} \forall(\mathrm{fanins\ f\ of\ u}) \mathrm{A_u} \geq \mathrm{A_f} + \tau * \mathrm{HPWL(f,u)} + \mathrm{D_u}$ $\forall \mathrm{u} \forall(\mathrm{fanouts\ f\ of\ u}) \mathrm{R_u} \leq \mathrm{R_f} - \tau * \mathrm{HPWL(u,f)} - \mathrm{D_u}$ $\forall \mathrm{register\ r}, \mathrm{R_r} \geq \mathrm{clock\ period}$ $\forall \mathrm{register\ r}, \mathrm{A_r} \leq 0$

circuit and not the number of paths. Further, *the number of constraints in which each gate and* 2*-pin connection appears is limited*, which is key to incorporating retiming, placement and cloning.

Max-slack retiming. Retiming is the most powerful optimization within SPIRE because it can effect drastic changes on the timing constraints. For instance, moving one register past a gate can allow *cycle stealing* on the order of gate delays along all paths that cross the register. In order to utilize the STA constraints described in the previous section, we develop a maximum slack formulation. The key idea in maximum-slack retiming is that there are two versions of the AAT and RAT computations on each vertex depending upon whether the vertex drives/is driven by a register. The constraints that are actually enforced are determined by the retiming. Therefore, the retiming program seeks a solution in which the values of retiming variables maximize worst-case slack.

Figure 7.8 shows the MILP that combines the STA constraints with retiming. During retiming, we only know the contents of the *retiming graph* (not the timing graph), because any edge in the retiming graph can include a newly retimed register. Therefore, STA constraints change depending on the retiming variable values. However, there are only two possibilities for each retiming arc: either the arc contains a register after retiming, or it does not (and combinations of arcs are implicitly considered). This situation is modeled through IF-THEN logic based on the retimed weight of the edge. If the weight is greater than zero, then the wirelengths involved in RAT and AAT computations change to incorporate the newly retimed register. For simplicity of presentation, we temporarily assume that the new register l will be placed at the *center of gravity (COG)* of the neighboring gates of l. Thus, the net connecting u to l has length HPWL(u, COG(l)) and the net connecting l to v has length HPWL(COG(l), v). In the next section, we eliminate this simplification and consider the static timing analysis of nearby gates when calculating slack-optimal register locations.

$$\begin{aligned}
&\mathbf{if}(\mathbf{w_r}(\mathbf{u},\mathbf{v}) == \mathbf{0})\\
&\quad R_u \leq R_v - \tau * \mathrm{HPWL}(u,v) - D_u\\
&\quad A_v \geq A_u + \tau * \mathrm{HPWL}(u,v) + D_v\\
&\mathbf{if}(\mathbf{w_r}(\mathbf{u},\mathbf{v}) \geq \mathbf{1})\\
&\quad R_u \leq R_l - \tau * \mathrm{HPWL}(u,\mathrm{COG}(l)) - D_u\\
&\quad A_v \geq A_l + \tau * \mathrm{HPWL}(\mathrm{COG}(l),v) + D_v
\end{aligned} \tag{7.8}$$

This IF-THEN logic is incorporated into a linear program using the *big-M* formulation. Under this formulation, a constraint $v < k$ takes the form $v < k + Mv_I$, where M is a large constant. If $v_I == 0$, the constraint reduces to the original, if $v_I \neq 0$ then the constraint simply becomes a bound on the variable v, i.e., $v < Mv_I$. Alternatively, IF-THEN logic can be modeled using *indicators*—binary variables that turn constraints on and off.[1] In our program, we define an indicator hasReg(u, v) as follows.

$$\begin{aligned} &\mathbf{if}(\mathbf{w_r}(\mathbf{u}, \mathbf{v}) > \mathbf{0})\text{hasReg}(u, v) = 1 \\ &\mathbf{if}(\mathbf{w_r}(\mathbf{u}, \mathbf{v}) \leq \mathbf{0})\text{hasReg}(u, v) = 0 \end{aligned} \tag{7.9}$$

This variable can be set in a variety of ways. One way is to use the constraint hasReg$(u, v) \leq w_r(u, v)$ and maximize it. If $w_r(u, v) == 0$ then hasReg$(u, v) = 0$. If $w_r(u, v) \geq 1$ then, since hasReg(u, v) is maximized, it is set to 1. However, maximization can sometimes conflict with the objective, therefore we use the following constraints instead:

$$\begin{aligned} &\text{hasReg}(u, v) \leq w_r(u, v) \\ &\text{if}(\text{hasReg}(u, v) == 0)\ w_r(u, v) = 0 \end{aligned} \tag{7.10}$$

The second constraint uses the *hasReg* variable as an indicator. Together, these two constraints require that hasReg $= 0$, *if and only if* $w_r(u, v) = 0$. For simplicity, we omit the setting of this variable from our formulations. As we will see in Section 7.3, maximization of real and integer variables can also fail when the objective has conflicting terms. Our formulation uses the constraints below to maximize general variables (without adding terms to the objective). We constrain the variable $C = \max(A, B)$ as follows.[2]

$$C \geq A,\ C \geq B,\ (C == A)||(C == B) \tag{7.11}$$

The min function is evaluated similarly. In Fig. 7.8, the slack, RAT, and AAT variables are real values while the retiming variables must be integer-valued. We utilize a constant weighting factor K to reconcile area with slack. The constant K can be adjusted based on the available area.

Note that the formulation in Fig. 7.8 does not require the derivation of the W or D matrices that were described in Sect. 7.2. Instead, timing calculations are performed within the MILP. Thus, the number of constraints is only $O(|E|)$ for a retiming graph with edge set E.

Register placement. Registers have special significance in a timing graph because their inputs are in a different clock cycle than their outputs. This facilitates *time borrowing*—the ability to shift delay from one timing path to another by decreasing the delay on inputs paths at the cost of increased delay on output paths, and vice

---

[1] Indicators are supported by the commercial MILP engine CPLEX 12.1.

[2] The Logic-OR can be implemented using intermediary variables δ_A, δ_B and indicator variables I_A, I_B with the following constraints: $\delta_A = C - A$, $\delta_B = C - B$, $I_A \leq \delta_A$, if$(I_A == 0)\ \delta_A = 0$, $I_b \leq \delta_B$, 6) if$(I_B == 0)\ \delta_B = 0$, $I_A + I_B \leq 1$.

Fig. 7.8 Max-slack retiming with STA embedded

Objective
$$\text{Minimize}: \; -\mathscr{M} + \sum_{(u,v)\in E}(K) \;\; w_r(u,v)$$

subject to

$$\forall(\mathrm{u,v}), \mathrm{r(u)} - \mathrm{r(v)} \le \mathrm{w(u,v)}$$
$$\forall(\mathrm{u,v}), \;\; \text{if}(!\text{hasReg}(\mathrm{u,v})) \;\; \mathrm{R_u} \le \mathrm{R_v} - \tau * \mathrm{HPWL(u,v)} - \mathrm{D_u}$$
$$\forall(\mathrm{u,v}), \;\; \text{if}(!\text{hasReg}(\mathrm{u,v})) \;\; \mathrm{A_v} \ge \mathrm{A_u} + \tau * \mathrm{HPWL(u,v)} + \mathrm{D_v}$$
$$\forall(\mathrm{u,v}), \;\; \text{if}(\text{hasReg}(\mathrm{u,v})) \;\; \mathrm{R_u} \le \mathrm{R_l} - \tau * \mathrm{HPWL(u,COG(l))} - \mathrm{D_u}$$
$$\forall(\mathrm{u,v}), \;\; \text{if}(\text{hasReg}(\mathrm{u,v})) \;\; \mathrm{A_v} \ge \mathrm{A_l} + \tau * \mathrm{HPWL(COG(l),v)} + \mathrm{D_v}$$
$$\forall \mathrm{u} \in \mathrm{V}, \;\; \mathscr{M} \le \mathscr{S}(\mathrm{u})$$

Fig. 7.9 Optimal register location relative to adjacent gates

Objective :
$$\text{Maximize} \;\; \mathscr{L}$$

subject to

$$\forall \mathrm{e} = (\mathrm{u,l}) \in \mathrm{E_l}, \;\; \mathrm{U_x^e} \ge \alpha_x^u, \;\; \mathrm{U_y^e} \ge \alpha_y^u$$
$$\forall \mathrm{e} = (\mathrm{u,l}) \in \mathrm{E_l}, \;\; \mathrm{L_x^e} \le \alpha_x^u, \;\; \mathrm{L_y^e} \le \alpha_y^u$$
$$\forall \mathrm{f} = (\mathrm{l,v}) \in \mathrm{E_l}, \;\; \mathrm{U_x^f} \ge \alpha_x^v, \;\; \mathrm{U_y^f} \ge \alpha_y^v$$
$$\forall \mathrm{f} = (\mathrm{l,v}) \in \mathrm{E_l}, \;\; \mathrm{L_x^f} \le \alpha_x^v, \;\; \mathrm{L_y^f} \le \alpha_y^v$$
$$\forall \mathrm{e} \in \mathrm{E_l}, \mathrm{L_x^e} \le \beta_x^l \le \mathrm{U_x^e}$$
$$\forall \mathrm{f} \in \mathrm{E_l}, \mathrm{L_y^f} \le \beta_y^l \le \mathrm{U_y^f}$$
$$\forall \mathrm{e} = (\mathrm{u,l}) \in \mathrm{E_l}, \mathrm{R_u} \le \mathrm{R_l} - \tau(\mathrm{U_x^e} - \mathrm{L_x^e} + \mathrm{U_y^e} - \mathrm{L_y^e}) - \mathrm{D_u}$$
$$\forall \mathrm{f} = (\mathrm{l,v}) \in \mathrm{E_l}, \mathrm{A_v} \ge \mathrm{A_l} + \tau(\mathrm{U_x^f} - \mathrm{L_x^f} + \mathrm{U_y^f} - \mathrm{L_y^f}) + \mathrm{D_v}$$
$$\forall \mathrm{e} = (\mathrm{u,l}) \in \mathrm{E_l}, \;\; \mathscr{L} \le \mathrm{R_u} - \mathrm{A_u} - \mathrm{D_u}$$
$$\forall \mathrm{f} = (\mathrm{l,v}) \in \mathrm{E_l}, \;\; \mathscr{L} \le \mathrm{R_v} - \mathrm{A_v} - \mathrm{D_v}$$

versa. By physically relocating registers, the interconnect delay around registers can be allocated to either the input or output paths.

In this section, we describe a formulation that integrates register placement with the retiming described in the previous section. Register locations alter STA constraints by changing interconnect length, and therefore, delays. On each edge with a register, SPIRE chooses the physical location that results in the best possible slack. The placement also interacts with retiming in that the retiming variables will optimize the STA constraints while considering register locations for each edge.

In order to perform this integration, we utilize the same type of case-logic as in the previous section. First we modify constraints so that AATs and RATs on edges with registers are calculated with respect to the placement. Register sharing along adjacent edges further complicates the formulation. However, we utilize the formulation from [10], to refine the placement of the shared register based on related timing. The retiming variables are, as in the previous section, optimized to activate the most favorable STA constraints. This interplay between retiming, placement, and STA is shown in Fig. 7.1.

Fig. 7.10 Max-slack retiming with relocation of registers

Objective
Minimize : $-\mathscr{M} + \sum_{(u,v)\in E}(K)\ \ w_r(u,v)$
subject to $\forall(u,v), r(u) - r(v) \le w(u,v)$ **if**(!hasReg(**u**, **v**)) : $\forall(u,v),\ \ R_u \le R_v - \tau * HPWL(u,v) - D_u$ $\forall(u,v),\ \ A_v \ge A_u + \tau * HPWL(u,v) + D_v$ Let **l** be register on (**u**, **v**) $\forall e = (u,l) \in E_l,\ \ \mathscr{L} \le R_u - A_u - D_u$ $\forall f = (l,v) \in E_l,\ \ \mathscr{L} \le R_v - A_v - D_v$ **if**(hasReg(**u**, **v**)) : $e = (u,l),\ \ U_x^e \ge \alpha_x^u,\ \ U_y^e \ge \alpha_y^u$ $e = (u,l),\ \ L_x^e \le \alpha_x^u,\ \ L_y^e \le \alpha_y^u$ $f = (l,v),\ \ U_x^f \ge \alpha_x^v,\ \ U_y^f \ge \alpha_y^v$ $f = (l,v),\ \ L_x^f \le \alpha_x^v,\ \ L_y^f \le \alpha_y^v$ $\forall e = (u,l), L_x^e \le \beta_x^l \le U_x^e$ $\forall f = (l,v), L_y^f \le \beta_y^l \le U_y^f$ $e = (u,l), L_x^e \le \beta_x^l \le U_x^e$ $e = (u,l), L_y^e \le \beta_y^l \le U_y^e$ $f = (l,v), L_x^f \le \beta_x^l \le U_x^f$ $f = (l,v), L_y^f \le \beta_y^l \le U_y^f$ $R_u \le R_l - \tau * (U_x^f - L_x^f + U_y^f - L_y^f) - D_u$ $A_v \ge A_l + \tau * (U_x^e - L_x^e + U_y^e - L_y^e) + D_v$

We first describe an LP formulation for local register relocation based on a simplified form of the LP in [10]. We then incorporate it into our retiming formulation.

Suppose register l can be incrementally placed to improve slack while leaving all other gates fixed. We define a timing graph $G_l = (V_l, E_l)$ that consists of vertices and edges that are adjacent to l. V_l contains the driver u, and sinks v, of l. The edge set E_l contains the timing arcs that a re adjacent to l. The LP formulation computes the variables β_x^l and β_y^l, the optimal x- and y-coordinates of l. The variables in this LP are as follows.

- α_x^v, α_y^v: fixed x- and y-coordinates of vertices $v \in V_l$.
- $U_x^e, U_y^e, L_x^e, L_y^e$: upper and lower bounds for the location of nets $e \in E_l$. These upper and lower bounds determine the HPWL of the particular net described by edge e as follows. HPWL(e) $= (U_x^e - L_x^e + U_y^e - L_y^e)$. As the location of the register changes, these net boundaries also change, and, in turn, change the HPWL.
- R_u, A_u: the AAT and RATs of nodes in V_l.
- $\mathscr{L}$: the local worst-case slack (of the worst pin in V_l).

The MILP to determine optimal register placement is shown in Fig. 7.9. This program sets the values of β_x^l and β_y^l such that $\mathscr{L}$ is maximized. Here, A_u of any vertex $u \in V_l$ that *drives* register l is fixed. Similarly R_v for any vertex v that is *driven by* l is also fixed. The only independent variables are β_x^l and β_y^l which determine the U and L variables. These, in turn, determine A_v, R_u for all nodes.

Fig. 7.11 Gate cloning in max-slack retiming

Objective

Minimize : $-\mathcal{M}+$
$\sum_{(u,v)\in E}(K)\text{RegCt}(u,v)+\sum(u)\text{IsCloned}(u)$

```
subject to
```

$\forall u, \forall$ fanins i of $u, \text{minPush}(u) \leq w(i,u) - r(i)$
$\forall u, \forall$ fanouts o of $u, \text{maxPull}(u) \geq w(u,o) + r(o)$
$\forall u$, $\text{if}(r(u) > 0)\text{maxPull}(u) \geq r(u)$
$\forall u$, $\text{if}(r(u) < 0)\text{minPush}(u) \leq -r(u)$
$\forall (u,v)$, $\text{if}(w_r(u,v) > 0)$
$\quad \text{RegCt}(u,v) = w_r(u,v), \text{CloneCt} = 0$
$\forall (u,v)$, $\text{if}(!\text{isClone}(u)\ \&\&\ !\text{hasReg}(u,v))$:
$\quad R_u \leq R_v - \tau * \text{HPWL}(u,v) - D_u$
$\forall (u,v)$, $\text{if}(!\text{isClone}(u)\ \&\&\ \text{hasReg}(u,v))$:
$\quad R_u \leq R_l - \tau * \text{HPWL}(u, \text{COG}(l)) - D_u$
$\forall (u,v)$, $\text{if}(\text{isClone}\ \&\&\ \text{hasClone}(u,v))$:
$\quad R_{\text{clone}(u)} \leq R_v - \tau * \text{HPWL}(\text{COG}(\text{clone}(u)), v) - D_u$
$\forall (u,v)$, $\text{if}(\text{isClone}(u)\ \&\&\ !\text{hasClone}(u,v))$:
$\quad R_u \leq R_v - \tau * \text{HPWL}(u,v) - D_u$
$\forall (u,v)$, $\text{if}(!\text{isClone}(v)\ \&\&\ !\text{hasReg}(u,v))$:
$\quad A_v \geq A_u + \tau * \text{HPWL}(u,v) + D_v$
$\forall (u,v)$, $\text{if}(!\text{IsClone}(v)\ \&\&\ \text{hasReg}(u,v))$:
$\quad A_v \geq A_l + \tau * \text{HPWL}(\text{COG}(l), v) + D_v$
$\forall (u,v)$, $\text{if}(\text{isClone}(v))$:
$\quad A_v \geq A_u + \tau * \text{HPWL}(u,v) + D_v$
$\quad A_{\text{clone}(v)} \geq A_v + \tau * \text{HPWL}(u,v) + D_v$
$\forall u$, $\text{if}(\text{isClone}(u))$
$\quad \mathcal{M} \leq R_{clone(u)} - A_{clone(u)} - D_{clone(u)}$
$\forall u, \mathcal{M} \leq R_u - A_u - D_u$

The program in Fig. 7.9 is modified in Fig. 7.10 to simultaneously incorporate retiming and placement, and no longer fixes the neighboring RAT and AAT variables. In this figure, each edge (u, v) on which a register appears constrains the placement of the register in question. It is assumed that all edges starting at u, i.e., of the form (u, v), such that $\text{hasReg}(u, v) = 1$ share the same registers. The register is placed in a location which minimizes the slack of neighboring gates. Since the slacks of neighboring gates in turn affect those of *their* neighboring gates, and so forth, a ripple effect ensues. Therefore, the register is actually placed in an optimal location with respect to the entire circuit. The key here is to enforce a small set of local constraints for each edge that interact with each other such that globally optimal solutions are chosen.

Cloning to increase the scope of retiming. A key insight in our work is that *opportunities for backward register movements are often limited by fanout branches in combinational circuits.* As illustrated in Fig. 7.2, retiming movements are blocked when fanouts of a gate do not share registers. We hope to increase these opportunities by cloning fanout branches such that registers can move beyond the cloned gate. We achieve this by relaxing legality constraints in specific ways that allow extra registers

to move backwards. In addition, the fanouts of any cloned vertex are divided such that the STA on some of the edges is computed with respect to the cloned, rather than original vertex.

The legality constraints in retiming ensure that no edge has negative weight. With cloning, edges can indeed have negative weight due to registers being retimed backwards through a cloned gate. However, forward retiming of registers still follows traditional legality rules.

Suppose node u has fanouts $O = \{o_1, o_2, \ldots o_T\}$ and fanins $I = \{i_1, i_2, \ldots i_m\}$. We represent this situation by imposing two constraints on the retiming variable r(u) for a node u: one which is enforced when $r(u)$ is positive, and one which is enforced when $r(u)$ is negative. If $r(u)$ is positive (i.e., the retiming is backward), then the maximum number of registers that are allowed to pass backwards is the greatest number of registers that appear on any fanout branch of u. If $r(u)$ is positive, then the constraint is the same as before:

$$\begin{aligned}
&\text{maxPull}(u) = \text{max}_{o \in O}(w(u, o) + r(o)) \\
&\text{minPush}(u) = \text{min}_{i \in I}(w(i, u) - r(i)) \\
&\textbf{if}(r(u) > 0) r(u) < \text{maxPull}(u) \\
&\textbf{if}(r(u) < 0) \text{minPush}(u) \geq -r(u)
\end{aligned} \tag{7.12}$$

Together, these two constraints can completely replace the general legality constraints. The presence of registers is indicated by a positive weight on an edge. Negative weights indicate that the driver of the edge was cloned. The original driver is connected to the retimed register on the (neighboring) edge(s) with non-negative weight, and the cloned driver drives the remaining sinks (as identified by edges with negative weight). We use the additional variable hasClone(u, v) which is set to 1 iff the register count on edge (u, v) is negative. These variables can be set in a similar way as hasReg. Recall that all constraints triggered under logical conditions can be incorporated into an MILP through indicator variables or big-M formulations.

The MILP incorporating cloning is shown in Fig. 7.11. For clarity, we illustrate cloning incorporated into the basic STA-based program with COG-based placements. In practice, we simultaneously place and clone registers and gates.

The slack is computed slightly differently in the presence of clones. New variables in Fig. 7.11 include indicator variables isCloned(u), $A_{clone(u)}$, $R_{clone(u)}$ for each vertex v. The variable isCloned$(u) = 1$ if hasClone$(u, v) = 1$ for one of the edges of the form (u, v). The computation of $A_{\text{clone}(u)}$, $R_{\text{clone}(u)}$ are:

$$\begin{aligned}
&\textbf{if}(\mathbf{w_r(i, u) - r(i) > 0}) \\
&\quad A_{\text{clone}(u)} \geq A_i + \tau * \text{HPWL}(i, \text{COG}(l)) + D_u \\
&\textbf{if}(\mathbf{w_r(u, i) - r(i) \leq 0}) \\
&\quad A_{\text{clone}(u)} \geq A_i + \tau * \text{HPWL}(i, u) + D_u \\
&\textbf{if}(\mathbf{w_r(u, i) - r(i) \leq 0}) \\
&\quad R_{\text{clone}(u)} \leq R_i - \tau * \text{HPWL}(\text{COG}(clone(i)), i) - D_u
\end{aligned}$$

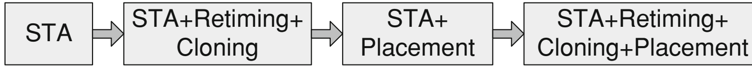


Fig. 7.12 Our SPIRE flow proceeds in phases. First the MILP that represents only static timing analysis is solved without design changes. The values of relevant variables are saved and passed to the next stage which runs an MILP that incorporates retiming and cloning. The retiming variables are saved and fixed in an MILP that allows latches to move. Finally, with known values for latch locations and retiming variables we run the complete linear program

For the new RAT variable, we assume that a node driven by a clone has no registers on the connecting edge. As illustrated in Fig. 7.11, the main differences in slack computation include 1) the additional edge $(u, clone(v))$ for every edge (u, v) where v is cloned, 2) the use of the clone's AAT, $A_{clone(u)}$, when computing the AAT of vertices v where (u, v) has a clone. We minimize the number of registers and clones in the retimed circuit using two variables isCloned and RegCt, which is computed as follows.

$$\mathbf{if}(\mathbf{w_r}(\mathbf{u}, \mathbf{v}) > \mathbf{0})\ \text{RegCt}(u, v) = w_r(u, v) \tag{7.13}$$

7.4 Empirical Validation

For very small circuits, a single mixed integer linear program implementing all of the optimizations in SPIRE can be solved in a reasonable amount of time. However, in order to push the boundaries of the largest circuits that SPIRE can solve, it is important to solve instances in several phases. Each of the components of SPIRE can be solved separately before being combined into a single mixed integer linear program. By saving the partial solutions and using them as a starting point for the next stage, we are able to achieve a significant speedup for large SPIRE instances without sacrificing optimality. Figure 7.12 shows the flow we use to improve the speed of SPIRE. It begins by running STA without any design changes allowed. The solution of this program is stored and used to seed the next stage, which adds retiming and cloning but fixes the locations of latches at the center-of-gravity of connected components. The solution of this program is used to add constraints to the next program, which allows latches to move, but not be retimed. Finally, the solution of that program is used to seed the combined program.

Experimental environment. We integrate our optimizations into an industrial physical synthesis flow. Our benchmarks are the largest functional units of a 45 nm high-performance microprocessor design. We operate on these benchmarks after timing-driven synthesis, timing-driven placement, electrical correction, and critical path optimization (through buffering and gate sizing) are completed [11]. We use an industrial timing analysis tool to obtain initial conditions for AATs and RATs throughout the circuit [12]. Our experiments were conducted on an 8-core system

Table 7.1 *Minimum slack* ($\mathscr{M}$) and *total negative slack* ($\mathscr{T}$) improvement during simultaneous retiming+placement on macros of a 45 nm microprocessor (see Eqs. 7.1–7.3). Maximal $\mathscr{T}$ improvement (100 %) is reached when design closes on timing. These cases are indicated in bold. $\%\mathscr{M}$ is computed as described in Eq. 7.14 with $\mathscr{P} = 174\ ps$

	#Std.	Initial			Retiming+Placement				Overhead	Improvements	
Design	cells	$\mathscr{M}$, ps	Regs	$\mathscr{T}$, ps	$\mathscr{M}$, ps	Regs	$\mathscr{T}$, ps	Time, s	% cells	% $\mathscr{M}$	% $\mathscr{T}$
azure1	536	3.42	41	0.00	10.14	49	0.00	1.19	0.00	3.87	0.00
azure2	1097	−2.53	79	−15.17	**2.95**	155	**0.00**	4.46	**6.93**	3.15	**100.00**
azure3	1032	−16.22	97	−212.69	−6.49	108	−37.95	0.4	1.07	5.59	82.16
azure4	1125	−2.30	79	−2.30	**3.82**	96	**0.00**	7.66	**1.51**	3.52	**100.00**
azure5	1140	−13.18	89	−114.54	**9.39**	161	**0.00**	40.71	**6.32**	12.97	**100.00**
azure6	1156	−10.49	83	−91.39	**7.14**	149	**0.00**	10.80	**5.71**	10.13	**100.00**
azure7	1198	−29.84	80	−3399.92	−17.02	145	−259.67	20.73	5.43	7.37	92.36
azure8	2578	−38.47	209	−391.03	−28.64	287	−265.68	24.87	3.03	5.65	32.06
azure9	2911	2.56	290	0.00	23.31	318	0.00	7.12	0.96	11.92	0.00
average									3.66	7.73	68.87

with 2.8 GHz AMD Opteron 854 CPUs and 80 GB of memory. Our MILPs were solved with ILOG CPLEX 12.1 configured to use up to 8 cores in parallel.

Table 7.1 shows a 7.7 % improvement (on average) in worst-case slack ($\mathscr{M}$) and a 69 % improvement in *total negative slack* ($\mathscr{T}$) when retiming with simultaneous placement. The slack improvements are reported in terms of the clock period $\mathscr{P} = 174\ ps$. $\mathscr{T}$ is computed as shown in Eq. 7.3 with threshold of $T = 0$. Percentage improvement in min-slack $\mathscr{M}$ is computed as follows.

$$\%\mathscr{M} = \frac{\mathscr{M}_{\text{new}} - \mathscr{M}_{\text{old}}}{\mathscr{P}} * 100\% \tag{7.14}$$

In addition, we note that the slack numbers are reported with respect to *buffered* wire delay. Past literature reports unbuffered wire delay, where slack may improve more dramatically, but such improvements may be misleading due to the need for subsequent buffering. In this experiment, the MILP for retiming with placement was given initial solution seeds from the max-slack MILP retiming shown in Fig. 7.8. This helped CPLEX to calculate MILP solutions quickly. The entire optimization sequence took < 41 s on each benchmark. Since our joint optimization was performed *after* several iterations of individual optimizations including placement, buffering, and gate sizing, and was able to significantly improve the slack, we can conclude that the individual optimizations were unable to find these solutions.

Table 7.2 evaluates the impact of cloning during retiming. In this experiment, we measure the *total thresholded slack* (Θ_T), as defined in Eq. 7.3, with the threshold $T = 100\ ps$. The threshold value represents the desired amount of guard-banding (protection) against process variations and NBTI, which can degrade timing. Empirical results indicate that cloning can improve the Θ_T of the circuit by up to 57 % over just retiming and placement. Thus, even when opportunities for cloning on the

Table 7.2 *Total thresholded slack* (Θ_T) improvement through simultaneous retiming, cloning and placement (see Eq. 7.3). Cloning also improved $\mathscr{M}$ on azure6 by 3.5%, while on remaining testcases the most-critical paths were not affected

Design	#std. cells	Initial Regs	Initial Θ_T, ps	Retiming+Placement Regs	Retiming+Placement Θ_T, ps
azure1	536	41	−4521.87	47	−2989.53
azure2	1097	79	−15597.31	153	−4537.57
azure3	1032	97	−15515.34	105	−14333.89
azure4	1125	79	−24206.70	81	−22226.57
azure5	1140	89	−35296.55	148	−18881.61
azure6	1156	83	−32183.65	148	−27566.43
azure7	1198	80	−46265.55	122	−33419.14
azure8	2578	209	−39253.82	296	−26272.53
azure9	2911	290	−13134.72	317	−9539.07

Design	Retiming+Cloning+Placement Regs	Retiming+Cloning+Placement Θ_T, ps	Retiming+Cloning+Placement Time, s	Overhead % cells	Improved % Θ_T
azure1	47	−2989.53	6.28	0.00	0.00
azure2	153	−4537.57	7201.14	0.00	0.00
azure3	110	−12739.10	2252.07	0.48	11.13
azure4	83	−21762.75	3727.78	0.18	2.09
azure5	537	−11333.49	7202.15	34.12	39.98
azure6	588	−11956.50	237.10	**38.06**	**56.63**
azure7	620	−17643.49	3741.82	41.57	47.21
azure8	657	−15117.06	7201.70	14.00	42.46
azure9	522	−4096.63	3905.28	**7.04**	**57.05**
average				15.05	28.51

critical path are limited, the remainder of the circuit can be improved for increased resilience.

Unlike previous localized transformations, SPIRE scales to design partitions with over 1000 cells as shown in the #std cells column in Table 7.1. SPIRE can process larger circuits by partitioning the design into windows of appropriate size, which can have overlaps.

7.5 Extensions

SPIRE's key advantage over existing physical synthesis transformations is the synergistic use of several types of optimizations. Our MILPs are more costly than existing transformations but also more powerful since they can be applied to larger windows than many of the localized transformations used in the industry today [10, 13]. This flexibility of SPIRE allows one to change size and scope of optimization and

offers rich trade-off opportunities between runtime and solution quality. However, increasing optimization strength will likely change the trade-off between runtime optimization-window size. Additional optimizations can be integrated into SPIRE as outlined below.

- To relocate combinational gates, create a variable for the x- and y-location for each gate and write the delay equations as in Sect. 7.3 in terms of those variables.
- To incorporate gate sizing in SPIRE, one must model nonlinear timing characteristics of individual gates or standard cells. This can be accomplished by precomputing the response to a set of discrete sizes (from the library) and selecting them using conditional constraints. If a particular gate size is selected, a corresponding gate delay will be used in the STA, as specified by a conditional constraint.
- Similarly, threshold voltage (V_{th}) assignment is modeled by selecting gate delays with Boolean variables. As lowering V_{th} improves speed at the cost of power, the number of low-V_{th} assignments must be upper-bounded.
- Common placement constraints including region constraints and obstacles can be represented in SPIRE. Region constraints are modeled with linear bounds on the x- and y-coordinates of each gate. To avoid obstacles, the placement region is divided into allowable regions that hug the obstacles. A disjunctive (OR-type) constraint is then added to require placement in one of the allowed regions. Routing congestion can also be represented as an obstacle using this mechanism to prevent any movable objects from being added in congested regions.

By integrating several optimizations and applying them to windows with thousands of objects, SPIRE offers a unique physical synthesis optimization that lies between local optimization of individual objects (which is typical of current tools) and global optimization of the entire design.

7.6 Conclusions

State-of-the-art physical synthesis methodologies tend to perform a series of local transformations to achieve a target clock period [11]. However, the persistent difficulty of timing closure in high-performance designs calls for *netlist transformations that can effect more powerful changes in the circuit*. To address these issues, we presented SPIRE, an MILP-based physical synthesis optimization in which dynamic netlist transformations including retiming, cloning, and placement, can be performed and co-optimized with respect to an embedded static timing analysis program. We demonstrated that isolated transformations, such as retiming, often run into obstacles that can only be resolved by other transformations, such as gate cloning. Empirical results show that SPIRE is able to significantly improve the worst-case and total slack in functional units of a 45 nm high-performance microprocessor after an industrial physical synthesis flow, consisting of several individual optimizations, is performed.

References

1. Alpert CJ, Chu C, Villarrubia PG (2007) The coming of age of physical synthesis. ICCAD 2007, pp 246–249
2. Trevillyan L et al (2004) An integrated environment for technology closure of deep-submicron IC designs. IEEE Des Test Comput 21(1):14–22
3. Leiserson CE, Saxe JB (1991) “Retiming synchronous circuitry,”. Algorithmica 6:5–35
4. Nair R, Berman C, Hauge P, Yoffa E (1989) Generation of performance constraints for Layout. TCAD 8(8):860–874
5. Sapatnekar S (2004) Timing. Springer, New York
6. Sapatnekar SS, Deokar RB (1996) “Utilizing the retiming skew equivalence in a practical algorithm for retiming large circuits,”. TCAD 15(10):1237–1248
7. Zhou H (2009) Deriving a new efficient algorithm for min-period retiming. ASP-DAC 2009, pp 990–993
8. Hu Y et al (2006) Simultaneous time slack budgeting and retiming for dual-vdd FPGA power reduction. DAC 2006, pp 478–483
9. Saxena P, Halpin B (2004) Modeling repeaters explicitly within analytical placement. DAC 2004, pp 699–704
10. Papa DA, Luo T, Moffitt MD, Sze CN, Li Z, Nam G-J, Alpert CJ, Markov IL (2008) RUMBLE: an incremental, timing-driven, physical-synthesis optimization algorithm. IEEE Trans on CAD 27(12):2156–2168
11. Alpert CJ et al (2007) Techniques for fast physical synthesis. In Proceedings of the IEEE, March 2007, Vol 95, No. 3, pp 573–599
12. Jess JAG et al (2006) Statistical timing for parametric yield prediction of digital integrated circuits. IEEE Trans on CAD 25(11):2376–2392
13. Moffitt MD et al (2008) Path smoothing via discrete optimization. DAC 2008, pp 724–727

Chapter 8
Broadening the Scope of Optimization Using Partitioning

Techniques covered in previous chapters have been developed primarily to operate in limited *optimization windows*, ranging from several gates (Chaps. 3, 5 and 6) to functional units of a CPU (Chap. 7). We extend their scope to a larger context—flat ASIC and SoC netlists—and facilitate greater parallelism during optimization. To accomplish this, the designs are divided by netlist partitioning tools into windows of manageable size, in which our earlier techniques can be applied. We evaluate window-partitioning in terms of runtime and solution quality as a method to extend the scope of physical synthesis optimization.

8.1 Introduction

Many important optimizations in physical synthesis are NP-hard, which motivates the use of high-performance heuristics to achieve timing closure. As outlined in Chap. 2, efficient (near-linear-time) heuristics, such as methods for large-scale standard-cell placement, are applied to entire netlists with millions of nets and standard cells. Alternatively, by limiting optimization to a very local scope, more CPU-intensive algorithms can be employed, including those that find optimal configurations of circuit elements. For tasks such as gate sizing, placement optimization within a single circuit row, and netlist partitioning, exponential-time exhaustive enumeration may be appropriate at scales of fewer than a dozen gates, with strong branch-and-bound implementations extending in scope to no more than 30–50 gates. *Our techniques range from applying to a dozen gates, as in interconnect-driven cloning, up to a few thousand gates in the case of SPIRE* (see Table 8.1). Scaling these optimizations to larger circuits will require applying them selectively within restricted windows of the design.

The controller/transformation approach to physical synthesis optimization introduced in Chap. 2 does not lend itself naturally to optimizations with large scope such as the ones proposed in previous chapters. This is because controllers choose *single*

D. A. Papa and I. L. Markov, *Multi-Objective Optimization in Physical Synthesis of Integrated Circuits*, Lecture Notes in Electrical Engineering 166,
DOI: 10.1007/978-1-4614-1356-1_8, © Springer Science+Business Media New York 2013

Table 8.1 Previously reported transformations and the maximum reported size of subcircuit to which they are applied

Transformation	Max reported subcircuit size (# standard cells)	Approximate runtime
RUMBLE (Chap. 3)	18	0.1 s
Ratchet (Chap. 5)	164	10 s
Interconnect-driven cloning (Chap. 6)	13	1 s
SPIRE (Chap. 7)	2911	10 s–2 h

```
OPTIMIZE-CLUSTER-WINDOWS
   ▷ Input: VLSI Circuit C, Target Window Size S,
            Controller D, Transformation T,
            Clustering Algorithm EXPAND
   ▷ Output: Optimized VLSI Circuit C′
1  while ( gate = D.next() )
2     window = gate
3     while ( window.size() < S )
4          EXPAND(window)
5     T.optimize(window)
```

Fig. 8.1 A generic iterative improvement physical synthesis algorithm that applies a transformation to a window based on bottom-up clustering. The performance of this algorithm can be tuned through the choice of clustering strategy, the selection of a controller and transformation pair, and through the runtime solution quality trade-off controlled by S. Chapter 3 explores using an n-hop clustering strategy and Chap. 5 was applied to windows selected in most-critical-first order

objects to optimize, and sequence such optimizations. However, our optimizations apply to larger numbers of objects and so there remains a problem of how to enumerate such subsections of the design on which to apply our techniques. In this chapter, we first describe how this was done for optimizations in Chaps. 3, 5 and 6, then we propose a strategy for selection of larger subcircuits for optimizations in Chap. 7 using top-down netlist partitioning (Fig. 8.1).

8.2 Background

The state of the art in physical synthesis relies on the controller/transformation model to select circuit elements to optimize, as introduced in Chap. 2. The most natural extension of the controller/transformation model to larger windows involves constructing a window around a given seed object that is designated by existing controllers. This method is appropriate in the case of a well-optimized design with relatively few problem areas. In this section, we review several methods to select windows by expanding a subcircuit around a given seed.

Breadth-first-search. In several important cases (gate sizing, buffer insertion, placement), the scope of simultaneous optimization among objects is determined by the connectivity and distance between the objects. Therefore, we aim to expand the window with objects that are directly connected to objects already in the window. If the goal is to optimize the seed, it is also more likely that something connected through a shorter path of nets and gates will influence the timing of the seed. Therefore, we consider the n-hop neighborhood as a good baseline strategy for expansion. The n-hop neighborhood is traversed efficiently in linear time by the breadth-first-search algorithm, as follows. Begin with a window containing only the seed s. Add all neighbors of s to a queue q. Dequeue a gate g from q and if it is not visited, add it to the window and mark it visited. Then add all of the neighbors of g to q. Repeat this procedure until the window reaches the desired size.

Most-critical-first. In cases where the goal is to fix a critical path, for example, using the techniques in Chap. 5, it may be advantageous to expand by adding the most-critical neighbor to the current window. This strategy begins with a window containing the seed s. Insert into a priority queue q the list of neighbors of s, sorted by their slack. Dequeue the most critical gate g from q and if it is not visited, add it to the window and mark it visited. Then add all unvisited neighbors of g to q. Repeat this procedure until the window is the desired size. Note that while the n-hop strategy radiates outward evenly around a gate, this strategy is very likely to expand along a single path and make a long, narrow window.

Slack-improvement order. It some cases an analytical model can be used to quickly estimate the amount of slack improvement that is possible due to the addition of the next gate. For example, a linear-delay model and coordinates can be used to estimate how much is the best-case improvement that can be provided by RUMBLE. Beginning from a window containing only the seed s. Insert into a priority queue q the list of neighbors of s sorted by slack improvement. Dequeue the gate g from q with highest slack improvement and if it is not visited, add it to the window and mark it visited. Then add all the unvisited neighbors of g to q (sorted by slack improvement). Repeat this procedure until the window is the desired size. This strategy requires a good slack improvement estimation technique and is therefore not always available. However, it provides an efficient trade-off between window size and solution quality.

The window selection strategies discussed in this section were found to work well in practice. Many other variants exist and, in general, the subcircuit selection strategy will depend strongly on the transformation it is used with. When coupling a subcircuit selection algorithm with a transformation, it is important to understand the effects of the transformation and what scope it needs to perform well.

8.3 Forming Subcircuits Using Top-Down Netlist Partitioning

In the previous section, methods to select subsections of a design based on a seed object were presented. Which method is appropriate for a particular transformation depends on its scope. For transformations that operate on a small neighborhood to

improve a target gate or net, bottom-up clustering allows one to easily select the set of nearby gates that are most likely to facilitate improvement to the target gate. Techniques of this type were used in Chaps. 3, 5 and 6 and successfully extended the scope of such physical synthesis optimizations as timing-driven gate movement, buffering, gate sizing and cloning. However, optimization windows remained relatively small in those cases, usually no more than around a dozen gates, but up to 164 in the case of Chap. 5. For transformations that apply to larger subsets there are too many combinations of gates for a comprehensive clustering algorithm to explore practically. In such cases, it is more appropriate to limit interactions with circuit elements outside of the subcircuit, and therefore partitioning is a good choice.

Netlist partitioning is an essential technique to moderate complexity in physical design systems. It enables algorithms and methodologies based on the divide-and-conquer paradigm. The goal of a partitioning algorithm is to divide a netlist into two or more groups of gates such that every gate is in exactly one group, and some cost function, such as net-cut, is optimized. Given a hypergraph representation G, of a netlist, the k-way hypergraph partitioning problem seeks k *disjoint partitions* of G. In this work we map the problem of finding subcircuits of a netlist to the k-way partitioning problem.

The Multilevel Fiduccia-Mattheyses (MLFM) framework is a well-studied approach to hypergraph partitioning and is presently the dominant technique for large-scale netlist partitioning [1]. It begins with a coarsening phase during which vertices of the hypergraph are merged to form a *clustered* hypergraph which has fewer vertices, e.g., half as many. The hypergraph is clustered repeatedly until a *top-level* hypergraph with 50–200 vertices is found. Then a *top-level solution* is constructed by means of a specialized solver designed for problems this size. For example, the Randomized Engineer's Method places vertices into partitions in largest-first order and tries to maintain balance as it proceeds. Following top-level solution construction, a *refinement* phase begins, wherein the hypergraph is unclustered, and the partitioning of the clustered hypergraph is projected onto the unclustered hypergraph. From this projected solution, an iterative improvement algorithm is applied, with the Fiduccia-Mattheyses (FM) algorithm being the most competitive today. Unclustering and iterative improvement are repeated until a partitioning of the *bottom-level* hypergraph (i.e., the input hypergraph) is obtained. Additional passes consisting of alternations of coarsening and refinement phases can be applied in so-called V-cycles to further improve results. One popular software implementation of MLFM, hMETIS, can be obtained from [2].

In order to produce subcircuits of a target size P of a netlist with hypergraph $G = (\mathbf{V}, \mathbf{E})$, we employ balanced k-way partitioning with $k = \frac{|\mathbf{V}|}{P}$. We then optimize each of the k windows individually. Each technique will have a runtime solution quality trade-off determined by the value of P. Table 8.1 shows a table of techniques reported in previous chapters and the size of subcircuits they can be applied to.

8.4 Trade-Offs in Window Selection

In addition to the scope of a given transformation as discussed above, several other considerations affect the choice of window selection technique, such as the interactions between the windows. Important factors include:

1. How subcircuit optimization is made relevant to the optimization of entire circuits
2. How overlaps between optimization windows affect solution quality and runtime
3. Whether all circuit elements are included in some window
4. The relative sizes of different windows.

We discuss trade-offs in window selection techniques in detail below.

Interactions between transformations and window selection methods. When the objective of a particular transformation is to minimize area, to fix local constraints or to repair design rule violations, optimizing subcircuits directly improves the entire circuit. However, when dealing with *non-local* timing constraints, relevant optimization objectives for a subcircuit must be carefully formulated. For example, when moving sequential elements in RUMBLE, combinational timing paths that leave the subcircuit but renter at a different point can strongly affect results. In Chap. 3 we refer to these types of configurations as *pseudomovable feedback paths*, and they must be carefully included into a subcircuit to account for their timing impact on the solution. More generally, windowing optimizations consider timing values on the boundaries *fixed*, while this may not be true in practice. Each transformation must carefully manage this assumption and include everything into the subcircuit that can change due to the effects of the transformation. As such, having a smaller boundary reduces the possibility of changes impacting the quality of optimization. This aspect of windowing is equally applicable to partitioning and clustering techniques. Trade-offs between these two window selection techniques are summarized in Table 8.2 and described next.

Window selection through clustering. Clustering techniques *per se* do not track overlap between windows, but leave several possibilities. One possibility is to construct optimization windows one by one, optimize the subcircuit in a given window, and then go on to the next window. Without sufficient care, such a technique is likely to create significantly overlapping windows, and some circuit elements may not be covered by any window. Overlaps occur when nearby circuit elements are used as seeds and expanding windows around them include similar sets of gates. This increases overall optimization effort by repeating transformations on the same circuit elements multiple times, but may sometimes improve solution quality by considering multiple contexts for each circuit element and iterating improvement algorithms on them. *Overlapping optimization windows cannot, in general, be processed in parallel—a serious drawback when a large number of networked workstations are available*. Circuit elements omitted from optimization windows may represent lost opportunities for optimization, but sometimes one can rule out such opportunities, e.g., for elements with high slack, low area or electrical parameters that satisfy relevant constraints (Fig. 8.2).

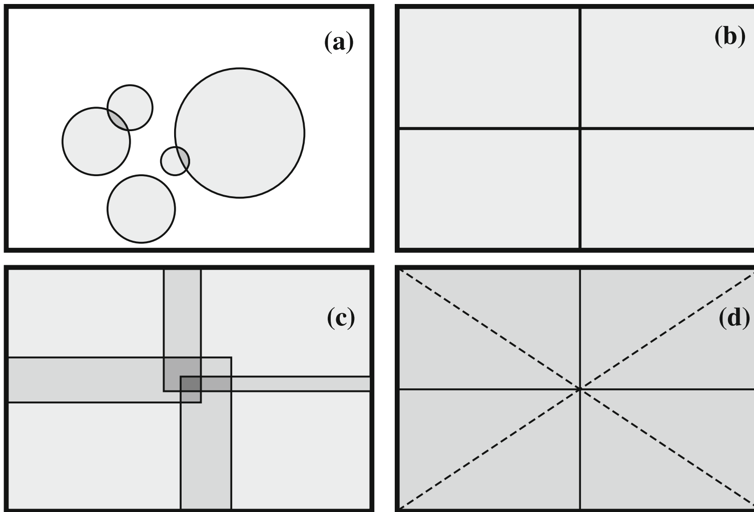


Fig. 8.2 Venn diagrams illustrating different window selection techniques. The outer rectangle in each image represents the entire design while shaded regions inside represent clusters or partitions. **a** Clustering grows windows around a seed object and typically creates overlapping windows that do not cover the circuit. **b** Partitioning divides the entire circuit into windows of approximately equal size that do not overlap. **c** The windows formed by partitioning can be expanded to deliberately create overlaps between adjacent partitions. **d** Partitioning can be performed multiple times to find orthogonal partitioning solutions. In (**d**) two independent 4-way partitioning solutions are overlaid, the solution from (**b**) is augmented by an additional one with dashed cutlines

Table 8.2 A comparison between window selection techniques

Property	Clustering	Partitioning
Window isolation	**Mediocre** (optimized indirectly by greedy algorithms)	**Substantial** (captured by the objective function and optimized by high-performance algorithms)
Window overlaps	**Substantial** (nearby seeds can cause overlapping windows)	**None** (but can be created through window inflation or repartitioning)
Circuit coverage	**Incomplete** (requires additional steps to revisit skipped nodes)	**Complete** (by construction)
Balanced windows	**Poor** (can be widely varying depending on adjacent net degree)	**Good** (balanced partitioning seeks similarly sized partitions)
Amenability to parallism	**Mediocre** (overlapping clusters cannot be solved simultaneously)	**Strong** (All partitions can be solved simultaneously)

A second possibility, relevant when overlaps should be limited in order to conserve runtime, is to mark each circuit element included in some window as *visited*, so as to prevent its inclusion in another window; a variant technique does not *mark boundary* elements of each window. Thus, it is possible to create (nearly) non-overlapping windows by clustering. However, in some cases this may leave cells with no unvisited

neighbors, and such windows may represent lost opportunities for optimization. If it is important to ensure that optimization windows cover the entire circuit, one can perform iterations where a new window is started for each circuit element not covered by earlier windows.

From a solution quality perspective, it is typically advantageous to construct windows of the largest size that can be efficiently processed by a given optimization (e.g., see Table 8.1). In such cases, therefore, it is advantageous for windows to be of similar size. However, if efficiency concerns dictate that windows cannot overlap, some windows may have to be smaller. Also, some windows may represent well-formed clusters of logic (e.g., multipliers or decoders) that are only loosely connected to the remaining circuit. Such windows can also be smaller than maximal reasonable size.

Window selection through balanced partitioning. Balanced partitioning addresses concerns about interacting windows effectively. Multilevel Fiduccia-Mattheyses (MLFM) partitioning exhibits near-linear runtime complexity in the size of netlists and runs efficiently on the largest VLSI netlists [1]. The most common objective function of MLFM partitioning is to minimize the number of nets that cross between two partitions. Therefore, MLFM partitioning minimizes the sizes of boundaries, and maximizes the *isolation* of each window. Such isolation helps to ensure that optimizations found locally will be preserved when taken in the context of the entire circuit. Partitioning also reduces the total overlap between windows by construction and is guaranteed to cover all elements in the circuit. Because of balance constraints in the partitioning formulation, all windows will have similar sizes and minimizing net cut ensures the logic within each window will be well-connected on average. These properties suggest that balanced partitioning is better-suited to identifying minimally-overlapping windows for non-local optimizations.

In cases when some overlap between partitions is desired to improve solution quality, clustering techniques seem to hold an advantage over partitioning techniques.[1] In particular, clustering techniques are better equipped to combine pairs of connected circuit elements (e.g., gates) together in at least one common window. Strategies employing partitioning techniques can address this limitation by performing several partitioning starts to obtain multiple solutions (increasing the likelihood that two given connected circuit elements will appear in at least one common optimization window).

8.5 Empirical Validation

For experiments reported in this section, we used the same computational facilities and EDA infrastructure as in Sect. 7.4, but added a larger design `azure10` with 4144 standard cells. For a given design, we partition the netlist into k partitions of approximately equal size using the hMETIS partitioner [2], for values of

[1] One hybrid technique begins by partitioning windows to smaller than the desired size then expands each using clustering to both cover the circuit and create overlap.

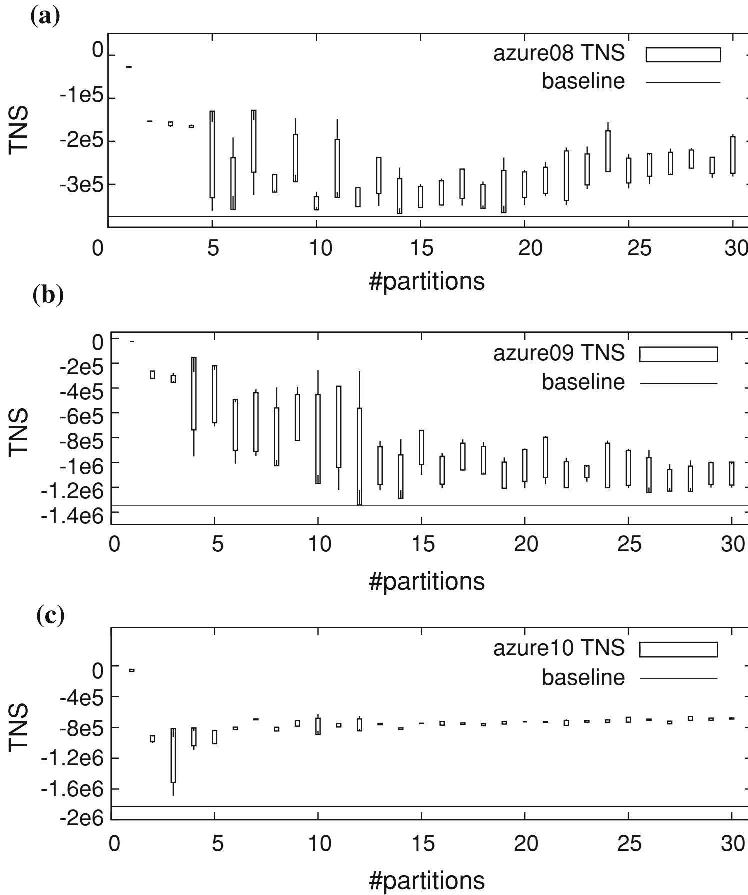


Fig. 8.3 An illustration of SPIRE's effect on $\mathscr{T}$ (TNS) versus the number of approximately equal-size partitions of three industrial microprocessor design blocks generated by the hMETIS partitioner [2]. **a** azure08; **b** azure09; **c** azure10. The horizontal axis indicates the number of partitions k. The vertical bars extend to $+/-$ one standard deviation from the mean value of $\mathscr{T}$. The wicks of candlesticks extend from the min to the max value of $\mathscr{T}$. The baseline indicates the value of $\mathscr{T}$ without changes to the circuit

$k = 1, \ldots, 30$. For each value of k, we solved k separate SPIRE MILP instances, and combined the solutions into a single solution for the testcase. We measured circuit-performance parameters after such optimization for each value of k and study the impact of the size of each partition on the performance of the circuit.

The techniques in Table 8.1 all improve solution quality at the cost of runtime when called on larger instances. This runtime solution-quality trade-off determines the best size for subcircuits in practice. In this section we demonstrate a trade-off between runtime and solution quality by partitioning large netlists and applying SPIRE (see Chap. 7).

Figure 8.3 shows an experiment incorporating the hMETIS partitioning software into SPIRE [2]. Each design was divided into $1 \leq k \leq 30$ partitions using 5 separate starts of the hMETIS partitioner [2]. SPIRE was invoked on every partition, and statistics of the resulting values of $\mathscr{T}$ are plotted. From this experiment, we observe:

- The best solution quality is obtained when the largest circuits are optimized.
- Using smaller windows sacrifices some solution quality, but it quickly converges in two of the three cases.
- Additional partitioning produces smaller, faster instances.
- In some cases smaller windows can provide greater improvement. This can be explained by our use of a time-out. Smaller windows are more completely explored within the time-out [3].
- Netlist partitioning is fast enough to apply to the largest ASICs and SoCs.
- In some cases the bars indicating $+/-$ one standard deviation can extend beyond the min or max value of $\mathscr{T}$. This occurs when the distribution of solutions is highly skewed toward its minimum or maximum.
- Solution quality can be significantly improved by applying several rounds of partitioning and selecting the best seen results. Such additional rounds can be performed in parallel. Because the smallest (fastest) windows often provide greater improvement than mid-size (slower) windows, one good strategy begins by solving small windows first, then proceeding to larger windows. A time-out or the runtime solution quality trade-off can be used to determine stopping criteria.

Partitioning and clustering allow one to apply each of the transformations in this book efficiently to the largest available designs. However, balanced, non-overlapping partitions are more amenable to parallelism. To this end, we partitioned a design with 102,063 standard cells into 1,000 partitions and ran SPIRE on each of them. SPIRE was able to find improvement in 119 of the partitions totaling 1.31e6 ns of TNS improvement. We plotted the amount of improvement in a histogram in Fig. 8.4. This experiment has been performed on a pool of compute servers because all of the partitions can be solved in parallel. In addition, each partition is solved using ILOG CPLEX 12.1 configured to use up to 8 processors in parallel.

8.6 Conclusions

In this chapter we have described a method to scale physical-synthesis optimizations to the largest commercial ASICs and SoCs. Working with such designs, we have applied our transformations after commonly used local transformations including buffer insertion, gate sizing, and detailed placement as follows. We first divide the entire netlist into windows of appropriate sizes for a particular large-scope optimization. We then apply that optimization within each window, leveraging inherent parallelism of disjoint windows. We then combine the solutions into a single optimized result. This method runs in near-linear time in terms of the number of windows and

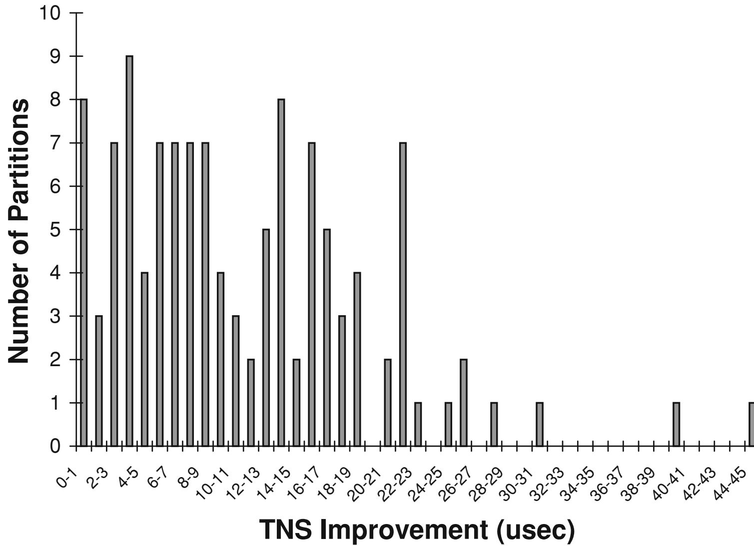


Fig. 8.4 A histogram of TNS improvement in partitions of a large ASIC

thus scales to a large number of windows. As long as each window is sized appropriately, algorithms with high runtime complexity can be applied while retaining affordable runtime on large designs. In addition we have identified three sources of parallelism compatible with our techniques—non-overlapping partitions, using a multi-core MILP solver, and multiple independent partitioning configurations.

We have shown that while increasing the scope of optimization provides improved solution quality, a divide-and-conquer framework allows EDA software to broaden the scope of heavy-weight physical synthesis optimizations and exploit parallelism. By controlling window size, we provide a trade-off between runtime and solution quality that can be tuned to make our large-scope transformations practical on the largest available designs.

References

1. Fiduccia CM, Mattheyses RM (1982) A linear-time heuristic for improving network partitions. In: Proceedings of the design automation conference, pp 175–181
2. hMETIS. http://www-users.cs.umn.edu/karypis/metis/hmetis/
3. Fišer P, Schmidt J (2010) It is better to run iterative resynthesis on parts of the circuit. In: IWLS, pp 17–24

Chapter 9
Co-Optimization of Latches and Clock Networks

Optimizations developed in earlier chapters affect many aspects of physical synthesis, but often target sequential elements, which particularly impact circuit performance. In order to obtain synergies between these optimizations, we explore the infrastructure for physical synthesis used by IBM for large commercial microprocessor designs. We focus our attention on a very challenging high-performance design style called *large-block synthesis* (LBS). In such designs latch placement is critical to the performance of the clock network, which in turn affects chip timing and power. Our research uncovers deficiencies in the state-of-the-art physical synthesis flow vis-à-vis latch placement that result in timing disruptions and hamper design closure. We introduce a next-generation EDA methodology that improves timing closure through careful latch placement and clock-network routing to (i) avoid timing degradation where possible, and (ii) immediately recover from unavoidable timing disruptions. When evaluated on large CPU designs recently developed at IBM, our methodology leads to double-digit improvements in key circuit parameters, compared to IBMs prior state-of-the-art methodologies.

9.1 Introduction

Design-complexity growth has consistently outpaced improvements in design automation in the last 30 years. The shortfall is called the *design productivity gap* and tends to increase the number of designers per project over chip generations [1]. However, the economics of the semiconductor industry limits the size of design teams, and the shortfall must be alleviated through increased design automation.

Modern CPU Design Styles. High-performance microprocessors demand very labor-intensive IC design styles. In order to cope with the high frequencies of these designs (3–6 GHz), engineers have traditionally partitioned them into hierarchies, with bottom-level blocks containing fewer than 10,000 standard cells. This methodology requires significant manpower for several reasons (i) the partitioning task is

D. A. Papa and I. L. Markov, *Multi-Objective Optimization in Physical Synthesis of Integrated Circuits*, Lecture Notes in Electrical Engineering 166,
DOI: 10.1007/978-1-4614-1356-1_9, © Springer Science+Business Media New York 2013

performed manually and it requires an experienced design architect, (ii) each designer can handle only around a dozen blocks; the use of smaller blocks increases their numbers and necessitates more designers, and (iii) *integrating* blocks into higher levels of the design hierarchy requires a dedicated designer for each *unit-level* assembly that combines multiple bottom-level macros.

Large-Block Synthesis. In order to improve the automation of synthesized blocks in high-performance microprocessor designs, a new design style is being pursued. Functional units are being *flattened* and all macros inside are merged into a single large, flat, high-performance block. The resulting entities are called *large-block synthesis* (LBS) blocks. The typical LBS testcase will have more than 25,000 thousand cells and possibly as many as 500,000 cells. The high-performance nature of such designs makes physical synthesis quite challenging. In particular, existing tools target high-performance designs (4 GHz or more) with small blocks under 10,000 cells, or low-power designs (400–800 MHz) with blocks having millions of standard cells. To improve the performance of the LBS methodology, current tools and techniques must be revised and extended.

Latch and Clock Network Co-Design Challenges. The large-block synthesis design style creates several conditions that stress existing physical synthesis flows in new ways. Like in high-performance small blocks, latches in large blocks must be placed in clusters near a *local clock buffer* (LCB) to limit clock skew and power [2]. However, the placement region of a large block leaves significant room for latch to be displaced by a greater distance. **The first major challenge** in physical synthesis of large blocks is limiting the displacement of latches when moving them close to LCBs. In addition, clock skew at every latch affects timing constraints for combinational logic. Therefore, critical path optimization—the focus of preceding chapters—must account for clock skew, but this information is not known until clock networks are designed. The latter step is commonly referred to as *clock insertion*. If clocks are inserted before the latches are properly placed, the timing picture will be overly pessimistic. Waiting to consider skew until too late in the flow may result in suboptimal circuit characteristics. **The second major challenge** is the fundamental issue of optimizing timing in the presence of clock skew, which requires careful ordering of latch placement and clock network synthesis operations. Traditional approaches to these problems suffer from significant timing degradations during sudden changes, e.g., moving a latch far across the chip to the location of an LCB. **The third major challenge** is avoiding severe timing degradations that harm convergence while managing latch placement constraints and optimization considering clock skew.

Our contributions. In this chapter we develop specific techniques to address the challenges above. In particular we note the following contributions.

- A graceful design flow to achieve timing closure by avoiding disruptive changes through careful reordering of steps. In some cases disruptions could not be avoided, and in these cases we either revise the offending optimization or mitigate the amount of disruption immediately after the disruption is detected.
- An algorithm to reduce the maximum latch displacement due to clock skew constraints by strategically inserting additional LCBs.

- A technique to reduce the displacement of combinational logic in response to moving latches to obey clock skew constraints. Compared to fixing the latches and rerunning global placement, our technique reduces combinational logic displacement significantly.
- A novel optimization for control signals that drive LCBs following a timing degradation caused by latch clustering.

The remainder of this chapter is organized as follows. Section 9.2 outlines a prior physical synthesis methodology for high-performance CPU design and the first major steps we took to cope with the large-block synthesis design style. Remaining specific problems in the flow that cause timing degradations are described in Sect. 9.3. Our new graceful physical synthesis flow is detailed in Sect. 9.4. We demonstrate the empirical improvements in our flow in Sect. 9.5. Conclusions are drawn in Sect. 9.6.

9.2 Background

In order to cope with the concerns of LBS designs, we adapt the typical microprocessor flow with several extensions designed for large ASICs. This section describes existing physical synthesis techniques for multi-million gate designs, and how they can be applied successfully to high-performance CPU designs.

Force-directed Placement. The current physical-synthesis methodology used at IBM relies on a quadrisection-based quadratic placement algorithm for high-performance microprocessor designs [3]. This algorithm works by first solving the quadratic program that is typical in analytic placement algorithms, then divides the cells into four groups by drawing cutlines to satisfy a density constraint. Next, it solves the quadratic program on those regions individually and repeats the process in a nested fashion until the cells can be placed by an end-case solver. The cut-based nature of this algorithm can cause small changes in the netlist to translate into large changes between two successive physical synthesis runs. This behavior exhibited by a placement algorithm is called *instability*. To avoid such disruptions, our next-generation flow incorporates a more stable *force-directed* approach that generally also results in better wirelength. The force-directed approach proceeds by an even spreading of cells after each quadratic solve, and this is the source of the improvement in stability. Figure 9.1 illustrates the progress of force-directed placement. A more stable placement process is important to ensure a steady path toward convergence despite disruptive changes during physical synthesis.

Force-directed placement algorithms are typically geared toward optimizing wirelength, and do not take clock network synthesis into account. As a result, latches are likely to be placed far from each other, spread throughout the placeable area. In turn, clock power and skew budgets can be exceeded when a high-performance clock network is synthesized using such post-placement latch locations. Therefore, metrics beyond wirelength must be employed during placement to satisfy chip performance requirements and minimize adverse impact on the clock network [4]. The following

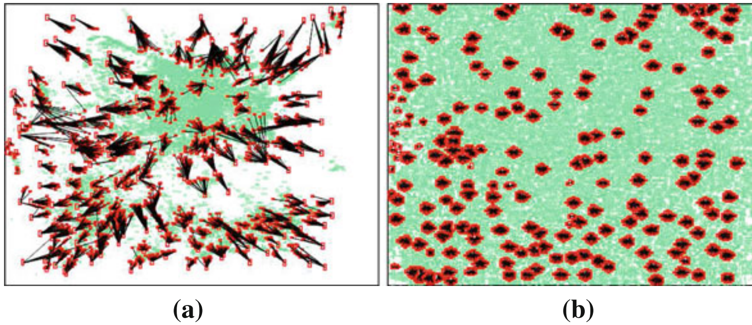


Fig. 9.1 The locations of cells during force-directed placement at the clockopt placement stage. **a** After one iteration of quadratic programming followed by cell spreading, a graceful spreading of cells can be observed. **b** The final placement resulting from repeating these iterations to convergence, followed by detailed placement and legalization

post-placement optimization problem is designed to mitigate timing degradations by minimizing latch displacement while creating tight latch clusters that enable reduced clock network power and skew.

Latch and Clock Co-Design. Latch locations are critically important to chip timing and dynamic power. We formalize the problem of optimizing latch locations for timing and power as follows.

Definition 1 (*The Latch and Clock Co-Design Problem*) Given a placed and optimized circuit layout G with l latches, a local clock buffer standard-cell LCB, a maximum number C of latches that can be driven by a local clock buffer, and a maximum distance $D > 0$ between any latch and the local clock buffer that drives it, satisfy all of the following constraints and minimize the following objective. Insert $\lceil \frac{l}{C} \rceil$ copies of LCB into the design, so as to drive at most C latches with each LCB, and place them to minimize latch displacement. Move any gates necessary so that latches are located within the required distance $D > 0$ from the local clock buffer that drives them. Minimize the sum of displacements of gate locations in the new circuit layout H as compared to G, $\sum_{g \in G}$ distance(location(G, g), location(H, g)).

Mercury is a state-of-the-art physical synthesis flow developed and used at IBM that is optimized for ASIC designs with over a million standard cells. It achieves a fourfold speed-up over previous approaches on designs that size. However, the Mercury flow was not designed for high-performance blocks, and is still not used on small blocks. Instead, the default flow for small blocks is referred to as the *Perseus* flow. Because LBS designs are high-performance, we first tried adapting the Perseus flow, which was designed for high-performance blocks. However, the runtime scaling implied that the largest LBS designs would require over one day of runtime in physical synthesis alone, while the required turn-around time for the entire flow is only 12 h. In order to achieve a speed-up on LBS testcases, we applied the Mercury

flow and enhanced it to deliver acceptable quality of results. The Mercury flow is also inherently more graceful than the Perseus flow, because directly after global placement, it quickly fixes all electrical violations and returns the timing environment to a meaningful state. Originally, the Perseus approach to electrical correction alternated timing-driven buffer insertion and gate sizing. This flow experienced convergence problems and was ineffective in fixing electrical violations. Placement causes degradations by creating long wires with electrical violations, therefore we conclude that a next-generation flow must include a post-placement clean-up step that specifically targets electrical violations, to ensure graceful convergence.

9.3 Disruptive Changes in Physical Synthesis

Recall that physical synthesis begins with a gate-level netlist that is produced by logic synthesis, then derives an optimized netlist and produces a chip layout A number of significant changes to the state of the design must occur while it is being processed. For example, when physical synthesis begins, gate locations are unknown, and a global placement algorithm must be invoked to find locations for all of the gates in the design. This is a disruptive change that will create vital new information as well as invalidate previously held assumptions. Whereas logic synthesis relies on crude timing models that abstract way interconnect, accurate interconnect delay models used after placement are likely to increase estimates of path delay. Whereas logic synthesis relies on crude timing models that abstract way interconnect, accurate interconnect delay models used after placement are likely to increase estimates of path delay. How a physical synthesis tool reacts to disruptive changes alters quality of results significantly. In this section, we discuss several sources of disruptions during physical synthesis and specific disruptive changes.

Changes in the Accuracy of Interconnect-delay Models. RTL-to-GDSII design methodologies begin with running logic synthesis on a rough RTL netlist. Then, a designer inspects the output of the logic synthesis tool vis-à-vis meeting timing constraints under a *zero wire-load model*. This sanity check ensures that physical synthesis is not invoked on a design where gate delays alone violate timing constraints. Subsequently, the netlist must be placed to facilitate interconnect delay estimation, e.g., using *Elmore-delay* formulas. The availability of physical information and the emergence of interconnect delays introduces a large disruption in timing estimates.

Uncertainties in Global Placement. In the example above, we pointed out that the input to physical synthesis is unplaced, and thus a global placement algorithm must be run before physical optimization can begin. From a physical synthesis perspective, the primary shortfall in state-of-the-art global placement algorithms is that they do not fully comprehend timing or electrical characteristics of gates and wires. Instead, they model optimization of these circuit characteristics using wirelength, on the assumption that good wirelength correlates with other objectives but is easier to optimize. As a result, timing and electrical characteristics are often undermined by

global placement, even on an optimized netlist.[1] Without improving multi-objective placement itself, avoiding this disruption in a physical synthesis flow is difficult.

Relocation of Latches Toward Local Clock Buffers. During synthesis, each clock domain is given a single *local clock buffer* to drive all the latches in that domain. However, each LCB is limited by a maximum capacitance that it can drive, and so later in the flow LCBs must be cloned in order to limit their fanout. This is not done during synthesis, since latch locations have not yet been determined. We would like to minimize the total length of clock interconnect between the LCB and the latches it drives, and this requires placement information. In order to limit the load driven by the LCB, and also reduce clock skew, we place the latches very close to the LCBs. During LCB cloning, the latches are grouped together into *latch clusters* and moved adjacent to the LCB that drives them. Such latch movement is disruptive in several ways, especially for the placement and timing of critical paths. It is not uncommon to see the worst-slack path degrade from around −50 ps to below −1 ns in response to this step. Minimizing latch movement is a key contribution of this chapter.

Early Timing Estimates based on Ideal Clocks. Since there is no placement information available directly after logic synthesis, a clock network cannot yet be routed. As such, detailed analysis of clock skew is impossible, and we therefore calculate nominal clock skews at latch pins with *idealized clocks*. Different methodologies synthesize clock networks at different stages. However, in high-performance methodologies at IBM, we consider the skew caused by the last level of the clock network after latches are placed and LCBs are inserted. Nets with high load lead to high clock skew, which can cause a serious disruption in timing, so the placement of latches and LCBs is critical. However, realistic clock networks are necessary to optimize the latch-to-latch paths while accounting for clock skew. Therefore, our work seeks to minimize the unavoidable timing disruption from realistic clock networks.

Simplified Slew Propagation. Static timing analysis is one of the largest consumers of runtime during physical synthesis, taking about 40 % of a typical physical synthesis run. One of the techniques used to mitigate this expense is called *pin-slew propagation*. In pin-slew propagation, instead of slew rates being propagated along paths, the slew rate used at a particular point is computed using a *default slew rate* asserted on its fanin gates, and propagated through one level of logic. This allows changes to timing to propagate only locally, which is considerably faster than *path-slew propagation*. However, this is an approximation that results in a loss of accuracy. In order to compensate, we switch to path-slew propagation during a late high-accuracy optimization mode. At the switch to path-slew, signal paths can experience major timing disruptions and become severely critical. We therefore develop a technique to improve the accuracy of default slew rate to mitigate this disruption.

[1] In state-of-the-art flows, placement can be invoked several times following optimization.

9.4 A Graceful Physical-Synthesis Flow

In this section, we develop a next-generation physical-synthesis flow that reduces or eliminates many of the disruptions and timing degradations outlined in the previous section. Our research strategy is based on observing and analyzing specific timing degradations. After understanding the emergence of such degradations, we first try to rearrange relevant steps of the design flow and revise individual flow steps so as to avoid degradations. When avoidance is impossible, we attempt to resolve the degradations immediately after observing them, using specialized design transformations. In the remainder of this section, we describe the improvements that implement our general strategy.

Gradual Evolution of Clock Networks is paramount to our next-generation physical synthesis flow and compliments techniques for latch placement proposed in previous chapters. To this end, we observe that in order to improve clock skew in high-performance design blocks, it is important to place latches reasonably close to driving local clock buffers. This step is performed during a stage of physical synthesis called *clock optimization*, during which realistic clock-network models are generated, LCBs are cloned, and latches are placed close to LCBs. As described in the previous section, all of these changes are disruptive for timing closure, and significant care must be taken during this stage to ensure a graceful flow.

The preexisting flow for this stage began by exposing the last level of the clock network, then performed LCB cloning and latch clustering, calculated net weights, and finally performed a global placement step called *clockopt placement*. This version of the flow is more disruptive than necessary due to the ordering of optimizations. The main problems are (i) the clocks are unhidden before the LCBs are cloned and latches are moved close by, and thus the clock skews are very large[2] and (ii) the net weights used for the global placement are based on inaccurate timing estimates that result from unhiding the clocks before optimization. This flow is illustrated in Fig. 9.2.

We solve these problems through a careful reordering of optimizations that takes into account which information is used by which step. In our new flow, which is shown in Fig. 9.3, the first step is to perform a new kind of LCB cloning and latch clustering, which is described below under *Length-Constrained Latch Clustering*. At this point, we have changed the clock network significantly and this requires timing assertions to be reread to get meaningful timing information. After that, in keeping with the philosophy that whenever we cause disruption we should repair it immediately, we introduce a new step following LCB cloning and latch clustering called LCB fanin optimization. This new step is designed to repair the damage caused by LCB cloning, and is described below under *Local Clock Buffer Fanin Optimization.* The timing should be completely recovered to its previous state following LCB fanin optimization because the LCB control signals are not high-performance signals. At this point net weighting is performed on a much more appropriately optimized netlist,

[2] Before LCB cloning, all latches on the chip are driven by a single LCB with very high fanout, resulting in very different latencies between different corners of the chip.

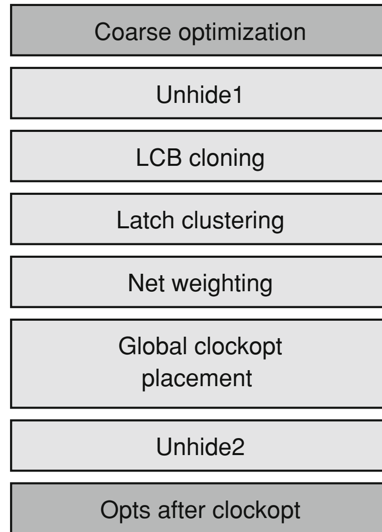


Fig. 9.2 The preexisting clock optimization flow exhibits several disruptive features. During Unhide1, the last level of the clock network is exposed to timing analysis, but the latches are not yet optimized. LCB cloning creates additional LCBs to limit the fanout of each LCB and latch clustering determines which LCB will drive each latch. Global clockopt placement ignores existing locations when determining a new location for each gate. Timing is reasserted after placement in Unhide2. Finally, additional coarse optimization is performed based on new timing conditions

and a novel placement step called incremental clockopt placement is performed as described below under *Incremental Clockopt Placement.* Following this placement step, LCBs are inserted and latches are placed near the LCBs, so as to minimize the disruption caused by unhiding the clocks (Fig. 9.4).

Idea 1: Length-Constrained Latch Clustering. At the beginning of the clockopt stage, latches are placed without any clocking-related constraints using the techniques in Chap. 3. We consider these locations to be the ideal latch locations from a signal timing perspective, and try our best to preserve these locations through the clockopt stage. However the LCBs must be cloned to limit the capacitance they drive, and latches must be placed close to the LCBs to reduce the clock skew. Therefore, we employ a geometric clustering algorithm called k-means which finds groups of closely-placed latches to be driven by the same LCB [5]. Pseudocode for our algorithm is given in Fig. 9.5. To reduce the disruption caused by moving latches close to LCBs, we define a new parameter *maximum latch displacement* and relax the constraint on the number of LCBs until no latch is more than this distance from an its LCB. The result is a tunable trade-off between timing disruption caused by latch displacement, and additional clock buffers which consume power and area (see Fig. 9.6). We have found empirically that, at the 32 nm node, latch displacement can be reduced to $<$500 routing tracks at the cost of a 25 % increase in LCB count.

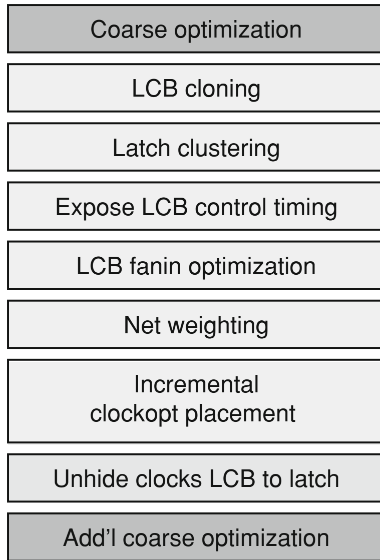


Fig. 9.3 Our next-generation clock optimization flow uses careful ordering of steps to avoid the largest degradations. LCB cloning creates additional LCBs to limit the fanout of each LCB and latch clustering determines which LCB will drive each latch, this is now done before clock timing is exposed. After many new LCBs are inserted, the control signals that drive them are traveling over an unoptimized high-fanout net. We optimize these control signals paths in LCB fanin opt. Incremental clockopt placement moves gates as little as possible when ensuring that latches are placed close to LCBs. Clocks timing is only exposed after the LCB to latch load is reduced to acceptable levels. Finally, coarse optimization based on mercury is performed

Idea 2: Local Clock-Buffer Fanin Optimization. LCBs typically support an *enable* signal or other control signals that are used for clock gating. After LCBs are cloned, all of the new LCBs are connected to the same control signal that was driving the original LCB. Immediately after LCB cloning, this net often experiences a severe timing violation caused by the heavy load of the high fanout. In trying to ensure a graceful design flow, we attempt to fix this unavoidable degradation immediately after it is created. To this end, we have created a novel LCB fanin optimization step and apply it immediately after LCB cloning. This step includes: (i) timing-driven gate placement for any logic in the control of LCBs, (ii) timing-driven buffer insertion to optimize long nets that may be created and (iii) timing-driven gate sizing to optimize the power levels of gates in the control logic. We have found empirically that these three steps are sufficient to restore timing to the level observed before LCBs were cloned.

Idea 3: Incremental Clockopt Placement. In the process of timing closure introduced in Chap. 2, physical synthesis is composed of iterations of (i) global placement, (ii) timing optimization, and (iii) per-iteration net weighting guided by timing analysis. However, running a complete global placement algorithm, albeit with new net weights influenced by the previous optimization, is a powerful disruption to timing

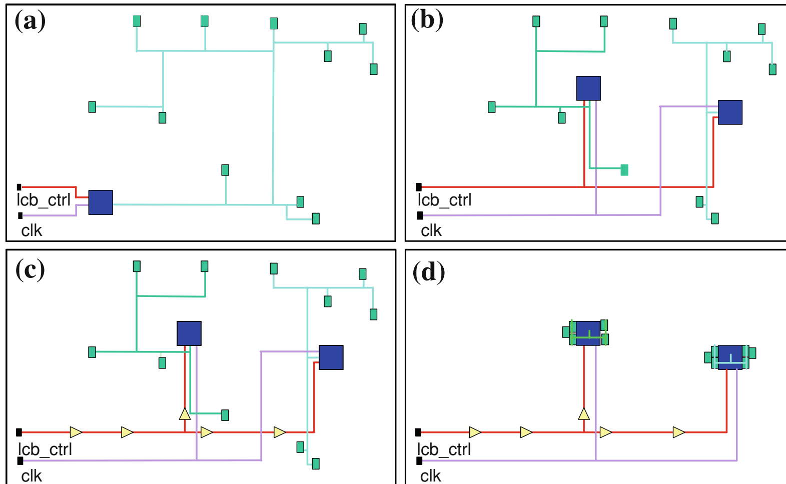


Fig. 9.4 An illustration of the flow in Fig. 9.3. At the beginning of clock optimization in **a** the clock is still idealized and latches are placed around the chip. In **b** local clock buffers (LCBs) are cloned and used to drive several latches each. To accommodate the timing impact of all the new LCBs, LCB control signals are optimized in (**c**). Global placement then moves latches close to LCBs in (**d**). Finally, leaf-level clock networks are inserted and clocks are unidealized

closure. In the IBM Physical synthesis flow, the first iteration employs very coarse models and constraints, e.g., relaxing the legality constraint for placement into looser, grid-based bin-area constraints. The second iteration uses more realistic models and requires a legal placement. At the end of the second iteration, a new constraint is added, the tool must then clone LCBs and move the latches near an LCB. In order to accomplish this with minimal design disturbance, we temporarily add two-pin nets with high weights to connect each latch to its driving LCB before global placement. Then, as global placement seeks to minimize weighted net length, the fake nets cause it to move each latch closer to the connected LCB, so as to shorten the fake nets. After placement, the fake nets are removed. The latches must be moved next to an LCB even if this displacement is very large, however, the bulk of remaining logic does not need to move far. Therefore, in order to minimize timing disruptions, we develop a new placement technique called *incremental clockopt placement*, which begins with a set of locations and leverages a technique for spreading and detailed placement called *iterative local refinement* on it [6–8]. This technique begins with a placement solution and overlays a gridded tile structure throughout the layout area. Gates located in a particular tile of this grid can be moved to one of eight neighboring tiles so as to improve wirelength while maintaining gate density. Crucially, we add a *maximum movement threshold* beyond which any displacement causes a high penalty to be imposed in the wirelength cost function. This allows the placer to bring the latches close to the LCBs, and allows the rest of the logic to adjust to the new locations of the latches, but prevents any large displacements in logic that will harm

```
CLUSTER-LATCHES
      ▷ Input: VLSI Circuit C, Maximum Number of Latches per LCB M
      ▷ Input: Number of LCBs K
      ▷ Output: Sets of Latch Clusters S, Maximum Latch Displacement L
 1    centers.ADD(center of gravity of all latches)
 2    foreach( 0 < i < K )
 3       new center = LOCATION(latch that is the furthest from any point in centers)
 4       centers.ADD(new center)
 5    latch list = list of latches
 6    sort latch list by distance to any point in centers
 7    while ( !latch list.EMPTY() )
 8       closest center = CLOSEST-CENTER(centers, latch list.FRONT())
 9       S[closest center].ADD(latch list.FRONT())
10       latch list.POP()
11       if (S[closest center].SIZE() ≥ M) ▷ cluster is full
12             centers.REMOVE(closest center)
13             sort latch list by distance to any point in centers
14    L = compute the maximum latch displacement for the clusters in S
```

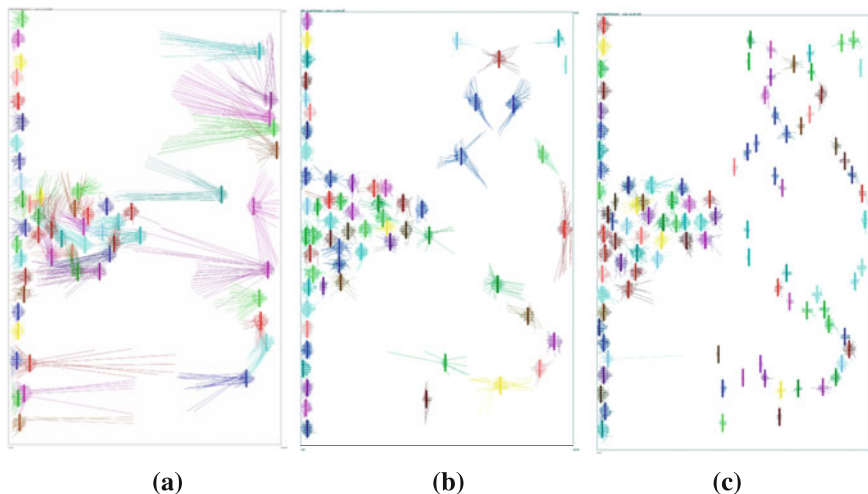
```
CLUSTER-LATCHES-LENGTH-CONSTRAINT
     ▷ Input: VLSI Circuit C, Maximum Number of Latches per LCB M
     ▷ Input: Maximum Number of LCBs N, Latch-Displacement Target D
     ▷ Output: Sets of Latch Clusters S
1    k = ceil(number of latches / M)
2    while ( k < N and maximum latch displacement > D )
3       k = k + 1
4       (S, L) = CLUSTER-LATCHES(C, M, k)
```

Fig. 9.5 An algorithm for length-constrained latch clustering

timing unnecessarily. The result is a significant reduction in total cell movement, which ensures a more graceful transition to tight latch clusters (Fig. 9.7).

9.5 Empirical Validation

In order to validate our proposed methodology we ran the physical synthesis tool PDS (commonly used at IBM) in various configurations to isolate individual flow improvements presented in the previous section. We used LBS microprocessor designs being developed at IBM for 32 and 22 nm technology nodes. Table 9.1 shows that our benchmarks range in size from 17,322 to 206,369 standard cells before optimization. Physical synthesis then inserts between 16.52 and 66.14 % more cells during optimization, with a median value of 27.33 %. The increase is mostly due to buffers and inverters, but specific numbers depend on local resynthesis and technology mapping. The performance requirements of these blocks are also an important characteristic, with target clock frequencies ranging from 1 to 4.35 GHz.



(a) (b) (c)

Fig. 9.6 Adding LCBs (shown by *vertical bars*) reduces the maximum latch displacement (*thin lines*). This behavior is controlled by two parameters (i) maximum increase in the number of LCBs, as a percentage of the minimum number (ii) maximum latch displacement, with (i) taking precedence over (ii). **a** The minimum number of LCBs is 56 and the maximum latch displacement is high. **b** By limiting parameter (i) to 12.5 % we get a maximum of 63 LCBs, and this noticeably reduces the maximum latch displacement. **c** We limit the maximum latch displacement to a tight limit using parameter (ii) but relax parameter (i) to get low latch displacement and 100 LCBs

Table 9.1 Large-block synthesis benchmark characteristics

Design	Technology node (nm)	Initial # gates	Final # gates	Cycle time (ps)	Dimensions (μm)
LBS1	22	206369	251021–255495	1000	1000×900
LBS2	32	190777	234912–248370	328	1498×1930
LBS3	32	51159	64909–74525	230	378×499
LBS4	32	88835	103514–122659	390	1000×800
LBS5	32	22837	28238–29184	230	449×225
LBS6	32	17322	26613–28779	460	180×397

The Final # gates column shows the range of possible gate counts using data from experiments presented in Tables 9.2 and 9.3

Implementation Insight: Default Slew Percentile. The common practice in computing a default slew rate is to sample the slew rates of the top critical pins. For example, one might calculate slew rates of the 800 most critical paths and use the average as the new default slew rate. Because we observe a degradation when switching to path-slew mode, we note that this must be an optimistic slew rate for those paths that are harmed, and we seek to make this estimate more pessimistic. Taking a larger set of pins to sample from is likely to increase optimism because we are examining them in most-critical-first order. Reducing the sample set will likely increase pessimism, but increase sensitivity and uncertainty that will make the

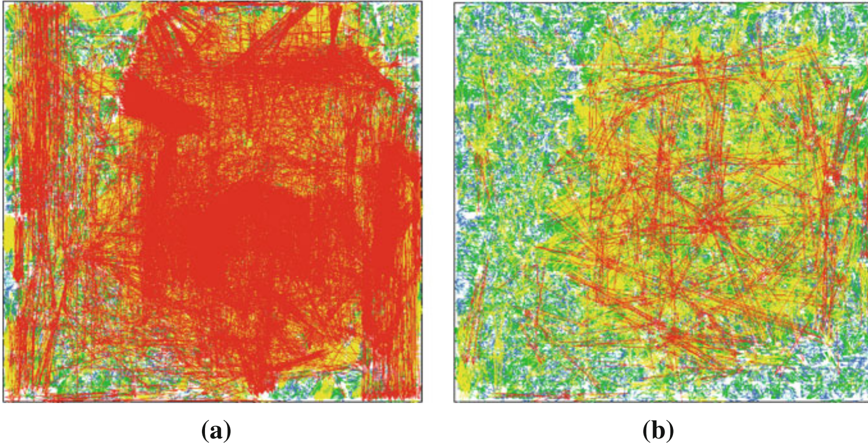


Fig. 9.7 Using incremental clockopt placement significantly reduces the disruption of the clockopt placement step. In each plot, a vector indicates the movement of a cell during the clockopt phase. *Red* vectors indicate displacements by over 500 tracks. *Yellow*, *green* and *blue* indicate 200, 100 and 50 tracks respectively. **a** Displacement vectors for all cells resulting from traditional force-directed placement. **b** Incremental placement reduces the number of red vectors drastically. Nearly all of the red vectors in this plot are due to latches which must be moved far to get to the nearest LCB

result unstable. Instead, we propose to automatically set the threshold for slew rate averaging as a certain percentile of pin slew rates (this threshold can be computed in linear time using the *nth-element* algorithm available in the C++ Standard Template Library). For example, if default slew rate percentile is set to 10 %, and we sample 500 pins, we will take the 50th worst slew rate from the sample set. After studying this parameter, we have found that 35 % is the best value to eliminate degradations when switching to path-slew mode. However, this pessimism must not be too great, for this would cause unexpected timing improvement at the switch to path-slew mode. This situation is problematic because earlier optimizations work hard to solve timing problems that disappear upon more accurate analysis, which wastes runtime, area and power.

Empirical Results. In Table 9.2 we compare the Perseus baseline to the following additions (i) only Mercury, (ii) only force-directed placement (FDP), and (iii) only gradual evolution of clock networks. In these designs we compare circuit performance metrics including the worst slack path in the design, and Φ, the sum of slacks below a threshold, which is computed as follows considering every timing endpoint i:

$$\Phi = \sum_i \min(0, \text{worst_slack}(i) - \text{slack_target}) \tag{9.1}$$

where slack-target is an input parameter to physical synthesis.

From the data in Table 9.2 we observe the following:

Table 9.2 The impact of individual components in the graceful flow. Time is the runtime of physical synthesis in seconds

Design	Mode	Time	Worstslack	Φ	WL	Area
LBS3	Baseline	40844	−76.641	−4203	1.43e7	0.5345
	Mercury	26578	−25.680	−384	1.45e7	0.4965
	FDP	37929	−55.015	−1519	1.32e7	0.5076
	Clockopt	37785	−7.536	−1214	1.41e7	0.5693
LBS4	Baseline	54442	−158.345	−81110	2.57e7	0.9942
	Mercury	41726	−189.391	−58881	2.47e7	0.9420
	FDP	52939	−167.016	−67799	2.41e7	1.0091
	Clockopt	56396	−148.050	−53442	2.04e7	0.8838
LBS5	Baseline	15274	−97.544	−6078	6.93e6	0.2382
	Mercury	9449	−98.374	−6293	6.98e6	0.2423
	FDP	16196	−82.391	−6288	6.87e6	0.2380
	Clockopt	13498	−87.287	−6265	6.80e6	0.2373
LBS6	Baseline	18476	−103.142	−16335	5.40e6	0.2218
	Mercury	13265	−89.288	−15300	5.74e6	0.2213
	FDP	18325	−88.207	−11755	5.14e6	0.2143
	Clockopt	19182	−103.682	−13958	5.05e6	0.2167
Average improvement	Mercury	**1.46X**	**−14.87%**	**−30.27%**	1.18%	**−2.72%**
	FDP	1.03X	**−13.19%**	**−26.21%**	**−4.93%**	**−1.75%**
	Clockopt	1.04X	**−26.67%**	**−29.18%**	**−7.59%**	**−1.82%**

WorstSlack is slack of the worst path in the circuit in picoseconds. Φ is calculated as in Eq. 9.1 and is expressed in picoseconds. WL is the sum of half-perimeter wirelengths and is expressed in routing tracks

- Mercury accounts for nearly all of the speed-up of this flow. It does not achieve the fourfold speed-up observed for million-gate designs, but a 1.42 × speed-up is significant for these designs which can take half a day.
- The use of force-directed placement adds stability to the flow and contributes a significant improvement in wirelength.
- Gradual clockopt results in a significant wirelength reduction as a result of calculating net weights based on a netlist optimized for timing after LCB cloning. This good timing result is a direct consequence of avoiding degradation in our graceful flow.
- Each component provides a significant overall improvement in terms of timing and area metrics.

In our next experiment we compare the baseline Perseus flow with our entire methodology combining all of the features presented in this chapter. The results are shown in Table 9.3. We observe the following

- Every testcase demonstrates an improvement in worst slack.
- Both worst slack and Φ average improvements are large, validating that the graceful methodology is an effective method to improve a timing closure flow.

Table 9.3 The impact of our graceful flow on key design parameters

Design	Mode	Time	WorstSlack	Φ	WL	Area
LBS1	Baseline	54105	−76.943	−1635	7.10e7	1.52
	Gradual	45753	−72.229	−367	6.73e7	1.55
LBS2	Baseline	41106	−128.605	−2004	8.97e7	1.84
	Gradual	42959	−56.667	−2276	9.65e7	1.96
LBS3	Baseline	25906	−28.102	−862	1.39e7	0.52
	Gradual	12846	−3.362	−66	1.39e7	0.44
LBS4	Baseline	30691	−153.924	−51674	2.55e7	0.99
	Gradual	22281	−70.667	−20025	2.05e7	0.73
Average improvement		**1.22X**	**−51.04%**	**−54.39%**	**−12.34%**	**−11.49%**

Time is the runtime of physical synthesis in seconds. WorstSlack is slack of the worst path in the circuit in picoseconds. Φ is calculated as in Eq. 9.1 and is expressed in picoseconds. WL is the sum of half-perimeter wirelengths and is expressed in routing tracks

- Area gains are inconsistent, but reductions of at least 10 % in cell area typically lead to reduced power, lower routing congestion and the potential for more aggressive floorplanning in future designs.
- Significant wirelength reductions alleviate demand for routing resources, resulting in improved routing congestion and improved downstream design closure.
- All metrics show strong improvement as a result of our methodology.

These experiments demonstrate the impact of each component in our methodology, and show that they ultimately translate into strong improvements in primary metrics of circuit performance and cost.

9.6 Conclusions

In this chapter we have introduced a new strategy to mitigate and eliminate disruptive changes in a physical synthesis flow. In implementing this strategy, we have identified key timing degradations that occur when new design parameters are introduced during physical synthesis. We then carefully revised relevant steps of the flow, made changes to the ordering of steps, and developed new optimization algorithms that were subsequently integrated into the overall flow. Our contributions are evaluated in the context of an industrial physical synthesis flow at IBM and several recent, large commercial IC designs that defied previous-generation physical synthesis tools. On the most challenging design type available to us, large-block synthesis designs, our flow achieves double-digit (percentage) improvements in all major circuit metrics considered.

References

1. International technology roadmap for semiconductors, 2009 edition. [Online]. http://www.itrs.net/
2. Cheon Y, Ho P-H, Kahng AB, Reda S, Wang Q (2005) Power-aware placement. In: Proceedings of DAC, pp 795–800
3. Vygen J (1997) Algorithms for large-scale flat placement. In: Proceedings of DAC, pp 746–751
4. Roy JA, Papa DA, Markov IL (2008) Fine control of local whitespace in placement, VLSI design, 2008:10, Article 517919. doi:10.1155/2008/517919
5. Selim SZ, Ismail MA (1984) K-Means-Type algorithms: a generalized convergence theorem and characterization of local optimality. PAMI 6(1):81–87
6. Viswanathan N, Chu C (2005) FastPlace: efficient analytical placement using cell shifting, iterative local refinement and a hybrid net model. IEEE Trans. on CAD 24(5):722–733
7. Viswanathan N, Pan M, Chu C (2006) FastPlace 2.0: an efficient analytical placer for mixed-mode designs. In: Proceedings of ASP-DAC , pp 195–200
8. Viswanathan N, Pan M, Chu C (2007) FastPlace 3.0: a fast multilevel quadratic placement algorithm with placement congestion control. In: Proceedings of ASP-DAC, pp 135–140

Chapter 10
Conclusions

A physical synthesis flow reads a mapped netlist produced by logic synthesis, then computes physical locations for gates and improves the performance of the circuit, until timing constraints are met. We observe that state-of-the-art flows consist of a series of optimizations that operate at two distinct scales, near-linear time algorithms that apply to the whole netlist, and more expensive transformations that typically operate on a handful of gates or interconnections. Such a limited view of the solution space of circuit optimization leaves many transformations vulnerable to becoming trapped in local minima. We observe this phenomenon on large, high-performance designs and improve upon the state of the art by integrating optimizations that are traditionally applied separately. Our novel transformations achieve broad opportunities for increased circuit performance and can handle larger design subsections than existing physical-synthesis transformations, thereby extending the scope of optimization. Given that the placement of sequential elements is a critical factor to the success of timing closure, we develop a next-generation timing-closure flow that improves the placement of sequential elements and facilitates the synthesis of high-performance clock networks.

10.1 Summary of Results

In this book, we describe several contributions that advance the strength and capabilities of modern software tools for IC physical synthesis, with the ultimate goal to improve the quality of leading-edge semiconductor products (see Fig. 10.1). Starting with

narrowly-focused optimizations, we identified obstacles to further improvements in circuit performance and addressed these obstacles with more powerful integrated transformations that outperform chained individual optimizations. Scalability was achieved in this approach by mapping circuit transformations to formal mathematical optimizations and through the use of efficient analytical delay models. To further improve the scope and efficiency of such hybrid transformations we developed

D. A. Papa and I. L. Markov, *Multi-Objective Optimization in Physical Synthesis of Integrated Circuits*, Lecture Notes in Electrical Engineering 166,
DOI: 10.1007/978-1-4614-1356-1_10, © Springer Science+Business Media New York 2013

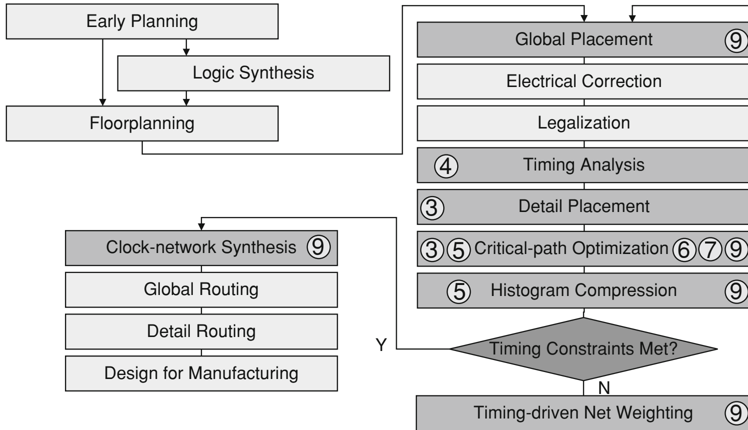


Fig. 10.1 The optimizations in this book improve nearly every stage of a state-of-the-art physical synthesis flow. For example, we illustrate that Chap. 4 deals with Timing Analysis by a adding a *circled 4* to that step in the flow

robust computational infrastructure and powerful circuit-analysis tools. Despite these enhancements, hybrid transformations remain somewhat expensive, motivating the development of divide-and-conquer frameworks that can handle large IC designs. When integrating our new transformations into the physical-synthesis infrastructure at IBM, we realized that these optimizations with increased scope tend to introduce disruptions into the design flow, and these disruptions adversely affect end results. We therefore developed a next-generation physical-synthesis flow that ensures a graceful improvement of key design parameters. Specific contributions are itemized below.

Simultaneous Placement and Buffering

At advanced technology nodes, multiple cycles are required for signals to cross the chip, making latch placement critical to timing closure. The problem is intertwined with buffer insertion because the placement of such latches depends on the location of buffers on adjacent interconnect. In Chap. 3 we detail our linear-programming-based algorithm to compute the optimal location of pipeline latches under a linear interconnect delay model [1, 2]. We then extend our algorithm to move nearby combinational logic gates to improve the effectiveness and applicability of this approach to simultaneous placement and buffering. Experimental results validate our transformation—our techniques improve slack by 41.3 % of cycle time on average for a large commercial ASIC design.

Bounded Transactional Timing Analysis

As local circuit optimizations become increasingly multi-objective in modern physical synthesis flows, a tighter interaction between optimization algorithms and timing analysis is necessary. Such optimizations must employ heuristics to search for good implementations of subcircuits, but timing analysis offers no support for retracting circuit modifications [3, 4]. In Chap. 4 we describe our extension to traditional static timing analysis that records a history of incremental network delay computations in a stack-based data structure, so that the timing can be returned to a previously-known state upon retraction of a circuit modification. It also explicitly *bounds* the scope of propagation to a local window in anticipation of retraction. These extensions greatly improve the performance of static timing analysis for local circuit modifications in the presence of retraction. For the classical variant of STA, our experimental results demonstrate an improvement of up to 246×, while a factor of up to 5.2× is achieved as compared to common lazy evaluation techniques.

Simultaneous Placement and Gate Sizing in a Discrete Domain

Gate locations that optimize timing depend on boundary timing conditions in the local subcircuit. Similarly, the optimal drive strength of a gate depends on the input slew rate and output capacitance. But these two problems are related because the output capacitance of a gate depends upon the length of interconnect it drives. Given a set of discrete candidate locations and power levels, we formulate the *path smoothing problem* in terms of a *disjunctive timing graph*, and develop a computation of optimal locations by incorporating a generalization of static timing analysis into an efficient branch-and-bound framework [5]. Empirically, our approach consistently improves solution quality in a large-scale modern industrial benchmark. Experimental results in Chap. 5 indicate that the techniques used in this chapter are accurate enough to improve the critical path optimization and slack-histogram compression stages of physical synthesis.

Timing-Driven Gate Cloning for Interconnect Optimization

In a complete physical synthesis flow, optimization transformations that can improve the timing on critical paths that are already well-optimized by a series of powerful transformations (timing driven placement, buffering and gate sizing) are invaluable. We develop an innovative gate cloning technique to improve interconnect delay on critical paths during physical synthesis [6]. Using the buffer-aware interconnect timing model introduced in Chap. 3, new polynomial-time optimal algorithms are presented for timing-driven cloning, including finding both optimal sink partitions (identifying the fanouts) for the original and the duplicated gates, as well as physical locations for both gates. In particular, for a gate g with m fanouts, Chap. 6 describes in detail two polynomial-time algorithms. For the case when g is fixed, we present

an $O(m)$-time optimal algorithm to maximize the worse slack of g. For the case when the g is movable, and one for the case when g is movable. If g is fixed, our $O(m \log m)$-time algorithm maximizes the worst-slack of g. For one hundred test-cases at the 45 nm technology node, we demonstrate significant timing improvement due to our cloning techniques as compared to other existing timing-optimization transformations. Extensions to handle other optimizations and constraints, such as wirelength, total negative slack and placement obstacles are further discussed.

Performance-Driven Retiming, Placement, Buffering and Logic Cloning

One of the most common situations in which the latch placement techniques of Chap. 3 are insufficient is a critical path wherein moving a gate immediately next to its most-critical input is the optimal solution but does not meet timing constraints. For example, when relocating the latch adjacent to its only input still violates a setup time constraint. We develop SPIRE, a new physical synthesis transformation that simultaneously incorporates retiming, gate relocation, gate duplication, and buffer insertion to improve this situation [7]. The need for SPIRE is demonstrated by example, motivating the integration of all considered techniques to meet timing constraints. SPIRE improves the performance of partitions in a high-performance microprocessor design. Empirical results on 45 nm microprocessor designs show 8 % improvement in worst-case slack and 69 % improvement in total negative slack *after* an industrial physical synthesis flow was already completed.

Broadening the Scope of Physical-Synthesis Optimization using Partitioning

The optimizations described in this book extend physical-synthesis transformations beyond a handful of gates. Unfortunately, the computational complexity of such optimizations makes them too inefficient to apply to entire netlists of large ASIC and SoC designs. Therefore, we develop a technique to identify appropriately-sized subsets of large designs on which our transformations can be applied efficiently. Our method utilizes existing hypergraph partitioning algorithms to divide the circuit in a top-down fashion until the subsets are the desired size. Empirical results demonstrate that this technique can work in practice and illustrate a run-time solution quality trade-off for SPIRE, the transformation described in this book that can optimize subcircuits with thousands of standard cells.

Co-Optimization of Latches and Clock Networks in Large-Block Physical Synthesis

Optimizations described in this book affect nearly every stage of a typical industrial state-of-the-art physical-synthesis flow. In order to obtain synergies between them, we explore the infrastructure for physical synthesis used by IBM for large

commercial microprocessor designs. We focus our attention on a very challenging high-performance design style called large block synthesis (LBS). In such designs latch placement is critical to the performance of the clock network, which in turn affects chip timing and power. Our research uncovers deficiencies in state-of-the-art physical synthesis flows vis-à-vis latch placement that result in timing disruptions and hamper design closure. We introduce a next-generation EDA methodology that seeks a more graceful timing-closure process. This is accomplished through careful latch placement and clock-network routing to (i) avoid timing degradation where possible, and (ii) immediately recover from unavoidable timing disruptions. Our methodology leads to double-digit improvements in key circuit parameters of large CPU designs developed at IBM.

10.2 Opportunities for Further Optimizations

The transformations developed in our work, along with prerequisite circuit analysis techniques, have significantly improved the quality of modern very large-scale integrated circuits developed at IBM. Much of this improvement is due to careful integration into a graceful physical-synthesis flow described in Chap. 9. Further work can address the following challenges.

Dealing with Modern Interconnect

With the explosion in the number of design rules, metal layers, and different routing pitches at advanced CMOS technology nodes, routing congestion is an increasing design challenge and layer assignment significantly affects delay estimation. The use of RUMBLE (Chap. 3) must take into account preexisting layer assignments. Areas with high wiring congestion may necessitate detours of critical interconnects, impacting circuit performance and jeopardizing timing closure. Therefore, routing demand and layer assignment must be analyzed early in physical synthesis and tracked through the physical synthesis flow in response to certain types of circuit transformations. We see an opportunity to formalize the handling ofbreak routing congestion in timing closure and develop effective benchmarks and algorithmic solutions [8]. As a first step, gate-placement techniques from Chaps. 3 and 7 can be extended to avoid congested areas. More sophisticated methods may be required in the methodology of Chap. 9 especially when dealing with clock trees and latch clusters.

Optimizing Power

Observe that in high-performance microprocessor designs, clock distribution is responsible for a large fraction of power consumption. We believe that our tech-

niques described in Chap. 9 improve not only circuit performance, but also power consumption. Configuring an environment for rigorous evaluation of power characteristics is an important direction for future work.

Global Placement to Improve Sequential Slack

Our transformations described in Chap. 7 make heavy use of physical retiming to improve combinational slack of circuits in question. This optimization was combined with placement, buffering and logic cloning. A further opportunity is to perform global placement so as to increase the *potential for such improvements*. This potential is expressed by the metric known as *sequential slack* [9]. Optimizing sequential slack during placement can provide improved opportunities for clock skew scheduling and retiming, and thus further broadens the scope of physical synthesis optimization. We expect that new global placement algorithms that optimize sequential slack can increase the applicability and effectiveness of retiming transformations developed in Chap. 4.

Handling of Large Macros and Intellectual Property (IP) Blocks

With billions of transistors integrated into a single chip, design complexity becomes a major challenge, as it defies the efforts of the best engineers and the capabilities of most recent software tools. One method to limit that complexity is to reuse design components in the form of IP blocks, but placement of such blocks is still largely done manually today. Such blocks typically incorporate latches immediately before and after primary outputs and inputs. Therefore, one bottleneck in circuit performance is the slowest sequential path between two such blocks. Incorporating this information into floorplanning and global placement algorithms is a significant opportunity to improve the design automation and performance of complex SoC designs [10–12].

Parallel Processing

Parallel processing is currently pursued by most developers of EDA software tools. Techniques presented in this book lend themselves naturally to such extensions. In particular, Chap. 4 outlines parallel extensions for bounded transactional timing analysis. Chapter 7 solves MILPs using the CPLEX tool in multi-core mode. Chapter 8 develops divide-and-conquer techniques for physical synthesis that partition the netlist and can spawn parallel computing tasks. Further incorporating our new transforms into physical synthesis tools and exploiting their inherent parallelism will improve the speed of next generation hardware as well as the physical synthesis tools used to design them.

Dealing with Process Variability

To account for the impact of variations in the manufacturing process, IBM has developed a robust statistical timing environment called EinStat [13]. However, statistical timing analysis is currently only used for sign-off timing, whereas optimization relies on the more conventional static timing analysis tool EinsTimer. Extending statistical timing analysis with features from Chap. 4 and incorporating it into physical-synthesis transformations (e.g., from Chaps. 5 and 6) will likely reduce pessimism in early design stages, accelerate timing closure, increase chip yield and reduce manufacturing cost.

References

1. Papa DA et al (2008) RUMBLE: an incremental, timing-driven, physical-synthesis optimization algorithm. In: Proceeding of ISPD 2008, pp 2–9
2. Papa DA, Luo T, Moffitt MD, Sze CN, Li Z, Nam G-J, Alpert CJ, Markov IL (2008) RUMBLE: An Incremental, Timing-Driven, Physical-Synthesis Optimization Algorithm. IEEE Trans. on CAD 27(12):2156–2168
3. Papa DA, Moffitt MD, Alpert CJ, Markov IL (2010) Bounded transactional timing analysis. Tau 2010
4. Papa DA, Moffitt MD, Alpert CJ, Markov IL (2010) Speeding up physical synthesis with transactional timing analysis. IEEE Design and Test of Computers, Sept 2010
5. Moffitt MD et al (2008) Path smoothing via discrete optimization. DAC 2008, pp 724–727
6. Li Z, Papa DA, Alpert CJ, Hu S, Shi W, Sze CN, Zhou Y (2010) Ultra-fast interconnect driven cell cloning for minimizing critical path delay. ISPD 2010, pp 75–82
7. Papa DA, Krishnaswamy S, Markov IL (2010) SPIRE: a retiming-based physical-synthesis transformation system. ICCAD 2010, pp 373–380
8. Papa DA, Adya SN, Markov IL (2004) Constructive benchmarking for placement. GLSVLSI 2004, pp 113–118
9. Hurst A, Chong P, Kuehlmann A (2004) Physical placement driven by sequential timing analysis. ICCAD 2004, pp 379–386
10. Adya SN, Chaturvedi S, Roy JA, Papa DA, Markov IL (2004) Unification of partitioning, floorplanning and placement. ICCAD 2004, pp 550–557
11. Roy JA, Adya SN, Papa DA, Markov IL (2006) Min-cut Floorplacement. IEEE Trans. on CAD 25(7):1313–1326
12. Roy JA, Papa DA, Adya SN, Chan HH, Lu JF, Ng AN, Markov IL (2005) Capo: robust and scalable open-source min-cut floorplacer. ISPD 2005, pp 224–227
13. Visweswariah C, Ravindran K, Kalafala K, Walker SG, Narayan S (2004) First-order incremental block-based statistical timing analysis. DAC 2004, pp 331–336