NORTHWESTERN UNIVERSITY

System–Level Optimizations for High Performance DSM Circuits

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Electrical and Computer Engineering

By

Jia Wang

EVANSTON, ILLINOIS

June 2008

ABSTRACT

System–Level Optimizations for High Performance DSM Circuits

Jia Wang

Process scaling has enabled the production of integrated circuits with millions of transistors. System-on-a-Chip becomes feasible as more functionalities can be packed into a single chip. As the human brain power is limited, the design process of such sophisticated systems should be automated in order to meet the stringent design specifications and short time-to-market. On one hand, ever-increasing system sizes require scalable algorithms for efficient system design space exploration. On the other hand, shrinking VLSI feature sizes require previous ignored physical effects to be considered for the reliability and manufacturability of the system. In this dissertation, we will present a few essential advances in design automation of high performance DSM circuits for the above challenges.

We investigate floorplanning techniques in order to address the methodology shift from logic centric to interconnect centric. We propose the processing rate as an unified performance measure and develop a floorplanning approach to optimize it directly through efficient minimum cycle ratio algorithms. Then, we present two graph-based floorplan approaches – one of them models the adjacency relations between the non-overlapping

rectangular blocks, and the other models the non-overlapping constraints between the blocks, while the sizes of both graphs are linear in terms of the number of the blocks.

Moreover, we investigate sequential system optimization techniques for system optimizations under performance bounds. We propose to combine sizing and clock skew optimization in a convex-programming-based framework and design an algorithm to solve the problem based on the method of feasible directions and min-cost network flow. We then present an optimal minimum area retiming algorithm that incrementally relocates flip-flops in a sequential circuit without changing its functionality subject to a clock period bound. Compared to the conventional algorithms that solve the same problem, our algorithm only requires linear storage and is practically much more efficient.

Further more, we investigate design for manufacturability techniques to consider issues in chip fabrication at the design time. We propose to perform risk-aversion min-period retiming in order to overcome process variations through retiming and present a heuristic incremental retiming algorithm. We study the antenna effect during fabrication process that damages gates and design an optimal algorithm in routing stage to insert jumpers under a bound of antenna ratio such that the damages of antenna effects will be limited.

Acknowledgements

I would like to express my gratitude to my advisor Professor Hai Zhou for his bringing me to VLSI design automation and for his supportive help and encouragement throughout my graduate study. Without his knowledge and invaluable guidance, it is not possible for me to understand design automation, to learn the art of algorithm design, and to become an independent researcher.

I would like to thank my dissertation committee, Professors Robert Dick, Yehea Ismail, Russ Joseph, and John Lillis, for their interests and suggestions on my work.

I would like to thank my high school teacher Mr. Zhanwang Qian for the training in formal mathematical reasoning skills and for the self-confidence achieved though hard working and competition.

Many thanks go to the members of the NuCAD Research Group, where I spent the past few years, for their discussions and collaborations, Dr. Ruiming Chen, Debasish Das, Bach Ha, Dr. Chuan Lin, Nikos Liveris, and Dr. Debjit Sinha. I should also include Dr. Zhenyu Gu in this list.

I am highly indebted to my family. I thank my parents for their support. I thank Danna, my wife, for her accompanying me at this distant land and most importantly, her love.

JIA WANG

EVANSTON, APRIL 2008

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

With the shrinking down of feature sizes in Deep Sub-Micron (DSM) VLSI technology, more functions are integrated into one chip and the performance of the circuits increases tremendously. The great promise of "system–on–a–chip" (SoC) becomes realistic and the VLSI systems become pervasive in our modern society. As it is impossible to design a single algorithm to realize and optimize such complicated systems, the system design process is separated into individual stages such that the design goal can be achieved through solving solvable design problems in each stage.

With the process scaling, new design challenges raise from two intrinsic complexities of the VLSI systems. The first complexity is known as the *system complexity*. As the transistor density increases, more functionalities are integrated in a single chip. A designer must explore a design space of tremendous number of system realizations to choose a correct one satisfying all the stringent design specifications in a short time-to-market. The second complexity is known as the *silicon complexity*. As VLSI feature sizes become smaller, previous ignored physical effects have to be considered for the reliability and manufacturability of the system. These challenges require the optimization algorithms to be scalable to ensure optimization efficiency for VLSI systems with ever-increasing sizes, and require the design methodology shift to address the limitations in the current design flow by proposing new design optimizations across previously separated design stages such that the design goal can be achieved with reduced complexity.

In this dissertation, we will present our advances in a few key problems of high performance DSM circuit design automation. These key problems include floorplanning techniques to address the methodology shift from logic centric to interconnect centric, sequential system optimizations to explore a larger design space for sequential circuits, and the design problems considering chip manufacturability.

1.1. Floorplanning Techniques

Interconnects become the dominant issue for performance and power consumptions for high-performance DSM circuits with the process scaling. A general trend is to bring physical information up into early design stages. Floorplanning, which addresses the issues of placing blocks as well as interconnect planning, is an enabling technology for such methodology shift.

The traditional objective for floorplanning is the total weighted summation of half perimeter wire length (HPWL). This objective is easy to calculate and other objectives are usually approximated through weighted HPWL. However, if the system performance should be optimized, such approximation is no longer applicable. In Chapter 2, we propose the processing rate as a unified performance measure and present the sequential system floorplanning approach that utilizes efficient minimum cycle ratio algorithms to optimize it.

The most fundamental issue in floorplanning is to represent non-overlapping rectangular blocks. On one hand, maintaining adjacency relationship between the blocks would benefit interconnect centric optimizations since block adjacency means shorter global interconnect. On the other hand, maintaining non-overlapping constraints would allow to

explore more physical floorplans in a mathematical programming formulation. In Chapter 3, we present *Constrained Adjacency Graph* (CAG) as an adjacency graph representation to maintain the adjacency relationship. We develop a linear complexity algorithm for constructing floorplans from CAGs and an iterative packing heuristic to improve the CAGs for better area without changing the adjacency relations dramatically. In Chapter 4, we present *Linear Constraint Graph* (LCG) as a constraint graph representation for general floorplans. It is the first constraint-graph-based floorplanning approaches where the numbers of the vertices and the edges are linear in terms of the number of the blocks.

1.2. Sequential System Optimizations

A large system is usually consisted of multiple components and each component has multiple implementations with different characteristics, e.g., performance, power consumption, and area. To synthesis the system from those implementations in order to meet the performance and cost constraints is a challenging problem because of the tremendous number of combinations. In the VLSI optimizations, such implementations could be different gate sizes, wire widths, supply voltages, and even buffers on interconnects. In Chapter 5, we generalize the continuous sizing techniques by presenting a convex-programming-based framework. In this framework, the system cost, e.g. power and area, is optimized subject to the performance constraint. We propose to combine sizing and clock skew optimization in this framework and develop an algorithm based on the method of feasible directions and the min-cost network flow techniques. The algorithm improves upon the conventional subgradient optimization approach that usually performs poorly without fine tuning.

Another method for performance optimization is to apply retiming, which optimizes a sequential system by relocating flip-flops while preserving the circuit functionality. As minimum clock period retiming would minimize the clock period but might incur overhead in flip-flop area and power consumption, minimum area retiming would optimize flip-flop area and power consumption but is of higher complexity. However, the conventional minimum area retiming algorithms were not scalable to large systems because of the quadratic storage requirement. In Chapter 6, we present an incremental minimum area retiming algorithm that only requires linear storage for better scalability. This is achieved by dynamically generating critical timing constraints only when they are needed and maintaining them as a forest. Our algorithm guarantees the optimality and is practically much more efficient than the conventional ones.

1.3. Design for Manufacturability

Shrinking geometries make the circuits more vulnerable to the variations and the damages during the fabrication process. The manufacturing yield and the product reliability can be greatly affected. Considering such physical effects in an early stage of system design would become inevitable.

In Chapter 7, we propose to characterize the stochastic output of the chip manufacturing process by a risk-aversion measure of the system clock period. We formulate the *risk-aversion min-period retiming* problem based on this measure in order to overcome process variations through retiming and present a heuristic incremental retiming algorithm to improve the yield by solving the problem.

In Chapter 8, we investigate the antenna effect in VLSI fabrication. It is a phenomenon where current caused by plasma process flows through gate oxides and damages them and thus reduces both the yield and the reliability. As combining jumper insertion to routing is known to be an effective method to reduce the damage, we present an dynamic programming algorithm to solve the jumper insertion for antenna avoidance problem under ratio upper-bound optimally.

CHAPTER 2

Processing Rate Optimization by Sequential System Floorplanning

The performance of a sequential system is usually measured by its frequency, or equivalently, its clock period. Sequential optimizations such as minimal period retiming [**18**] and clock skew scheduling [**19**] can be used to optimize the clock period by balancing the combinational path delay between consecutive flip-flops. When interconnect delays begin to dominate the performance because of the aggressive scaling down of geometries in Deep Sub-Micron (DSM) VLSI technology, things become much more complicated. Unlike gate delays, interconnect delays are only available until very late in the current design flow and always after floorplanning and placement. Different floorplans or placements give different minimal clock periods after sequential optimizations. Ignoring such optimization possibilities in the design flow may result in sub-optimal solutions. The work [**20**] addressed this problem by considering the optimization potential in physical placement. In their work, the maximum ratio of the delay over the flip-flop number along any cycle in the circuit is known to bound the minimal clock period that can be achieved through sequential optimizations. To optimize the ratio in the placement, they identified the cycle with the maximum ratio in the current placement and used it as a constraint to find a new placement with a smaller maximum ratio.

When the operating frequency of the sequential system is given, it is possible that one clock period is too small to propagate a signal from one end of a long wire to the other end in one clock period. In this case, wire pipelining is vital to allow multi-cycle communication over a long wire. As suggested in [**21**], retiming can be used to pipeline the long wires. However, if the frequency is higher than the best one that could be achieved by retiming, more wire-pipelining units like flip-flops must be introduced on those long wires to pipeline them. The side effect is that these additional units may change the latency of some parts of the circuit so that the functionality is different from the original one. Both the latency insensitive design (LIS) [**22, 23**] and the wire-pipelining correcting method [**24**] addressed this problem. In these two approaches, the throughput, which is defined as the amount of the processed inputs per clock cycle, is traded for higher frequency and is no longer one as in retiming or clock skew scheduling. In both cases, the throughput is bounded by the minimum ratio of the flip-flop number to the number of the required wire-pipelining unit along any cycle. The work [**17**] optimized the throughput bound in LIS with floorplanning. They used a heuristic throughput evaluation method in the simulated annealing (SA) floorplanner Parquet [**25**] based on an assumption that an exact throughput evaluation is too time-consuming in SA. The heuristic method estimates the throughout by assigning different weights to different wires according to their contributions to the throughput.

We find that by looking at the bound of the processing rate, we can unify the situations where the throughput is fixed and the frequency is optimized and where the frequency is fixed and the throughput is optimized. In any case, the processing rate is bounded by the minimum ratio of the flip-flop number over the delay along any cycle. Since this bound is

independent of the operating frequencies and the afterward optimization/wire pipelining methodologies, it is more general than the clock period bound or the throughput bound.

We optimize the processing rate bound directly in a SA based floorplanner. Unlike the previous approaches where the bounds were handled indirectly, Howard's algorithm [**26, 27**] is applied to compute the bound exactly inside the inner loop. The resulted floorplans are evaluated under different frequencies to obtain the throughputs and the processing rates. To show that our approach achieves better solution quality and running time, we apply our algorithm to GSRC benchmarks as in [**17**] and compare our results with theirs. We need to point out that the results in [**17**] are optimized under different frequencies separately so our one-floorplan-fit-all approach is more universal and less time consuming.

The *Adjacent Constraint Graph* (ACG) [**10, 2**] is used as our floorplan representation. It preserves the geometrical adjacency information and its operations map to local perturbations in physical space. To exploit those local perturbations, we apply Howard's algorithm incrementally, which speeds up the floorplanner by 29% on average. In addition, we show that addressing the fixed-outline constraint explicitly in the cost function is possible and effective in the ACG representation.

The rest of this chapter is organized as follows. In Section 2.1, we show how the minimal cycle ratio bounds the processing rate of a sequential system and formulate the *Floorplanning for Processing Rate* (FPR) problem. The SA based floorplanner incorporating processing rate optimization is presented in Section 2.2. Experimental results are shown in Section 2.3. Section 2.4 concludes the chapter.

2.1. Processing Rate and Floorplan Problem

2.1.1. Processing Rate Bound

We define the processing rate of a sequential system as follows.

Definition 2.1 (Processing Rate)**.** *In a sequential system, the processing rate is defined as the length of processed input sequence per unit time.*

Considering the frequency and the throughput in a synchronous system, we have the following observation.

Observation 2.1. *For a synchronous system, the processing rate is equal to the product of the frequency and the throughput.*

A synchronous system is modeled by a directed graph $G = (V, E)$. In the simplest case, each vertex is a combinational gate and each edge is an interconnect wire with signal direction. When the system contains modules that could not be treated as gates, we will follow the method in [**21**], where vertices are pins of modules and edges represent either the interconnects or the fan-in fan-out timing constraints inside modules. Let the number of the flip-flops along a wire $e \in E$ be $w(e)$ and the interconnect delay along the wire be $d(e)$. By optimal buffer insertion [**28**], the interconnect delay is linear to its wire length. For any cycle C in the graph, we define $w(C) = \sum_{e \in C} w(e)$ and $d(C) = \sum_{e \in C} d(e)$.

Minimal period retiming optimizes a system by relocating the flip-flops so that the clock period of the system, which is equal to the longest combinational path delay between two consecutive flip-flops, is minimized. On the other hand, clock skew scheduling assigns non-zero skews to the clocks driving the flip-flops. The flip-flops act like that they are

moved around to balance the combinational path delay so that the minimal clock period can be achieved. If there is a cycle C in the system, when applying minimal period retiming or clock skew scheduling, the flip-flops along C divide C into $w(C)$ combinational parts. So there must be a combinational path between two consecutive flip-flops C whose delay is at least $\frac{d(C)}{w(C)}$. Therefore, the minimal clock period is bounded by

$$\lambda_G = \max_{cycle\ C\ in\ G} \frac{d(C)}{w(C)} \tag{2.1}$$

As the throughput is always one here, the processing rate is the same as the frequency. So it is upper bounded by $\frac{1}{\lambda_G}$.

When the operating frequency is fixed, either latency insensitive design (LIS) [**22**] or wire-pipelining with correction [**24**] can be applied to pipeline the long wires.

LIS makes the system functionally insensitive to the latency of the long wires by a latency insensitive communication protocol and additional control logics around computational blocks in the original system. In LIS, signals propagating in the system are divided into two categories: informative events corresponding to the signals in the original systems and stalling events asking the receiving entities, called pearls, to stall a cycle waiting for the informative ones. To pipeline long wires, wire pipelining units, called relay stations, are placed on those wires. Relay stations act like flip-flops but can handle the communication protocol. In addition to the area overhead introduced by the additional control logics, the throughput is affected by the stalling events introduced to the system. As discussed in [**23**], the throughput is bounded by the minimum cycle mean among all cycles in the system where the cycle mean is defined as the pearl number plus the relay station number divided by the pearl number along a cycle.

According to [**22, 23, 17**], we use G to model a system such that every vertex corresponds to a computational block which could be treated as gates and every edge corresponds to a communication channel. $w(e)$s are all 1 now because initially there is no flip-flops on the communication channels and every computational block latches its outputs in flip-flops every clock cycle. Let the clock period be ϕ under the target frequency. We need at least $w^*(e) = \lceil \frac{d(e)}{\phi} \rceil$ wire-pipelining units to pipeline the wire e. Among them, one is the original flip-flop latching the output and all the others are relay stations. Therefore, the throughput is bounded by

$$\min_{cycle\ C\ in\ G} \frac{w(C)}{\sum_{e \in C} w^*(e)} \leq \min_{cycle\ C\ in\ G} \phi \frac{w(C)}{d(C)} = \frac{\phi}{\lambda_G}$$

So processing rate is bounded by $\frac{1}{\lambda_G}$.

Another approach [**24**] applies the wire-pipelining correcting method after inserting extra flip-flops to pipeline the long wires. We still use the ϕ to represent the clock period. The minimal number of flip-flops needed to pipeline a wire is $w_p(e) = \lceil \frac{d(e)}{\phi} \rceil$. We cannot feed a new input every clock cycle without affecting the functionality if there is the additional latency caused by extra flip-flops in a cycle. With ρ-slow transformation [**24**], we feed a new input every ρ cycles where

$$\rho = \max_{cycle\ C\ in\ G} \lceil \frac{\sum_{e \in C} w_p(e)}{w(C)} \rceil$$

The throughput of the system is $\frac{1}{\rho}$. The processing rate is bounded by $\frac{1}{\lambda_G}$ again since

$$\begin{aligned}\frac{1}{\rho\phi} &\leq \min_{cycle\ C\ in\ G} \frac{w(C)}{\sum_{e\in C} w_p(e)\phi} \\ &\leq \min_{cycle\ C\ in\ G} \frac{w(C)}{\sum_{e\in C} d(e)} = \frac{1}{\lambda_G}\end{aligned}$$

Therefore, we have the following theorem.

Theorem 2.1. $\frac{1}{\lambda_G}$ *is the upper bound of the processing rate of a synchronous system no matter what technique is used for wire pipelining.*

Proof. The above discussions imply that the claim holds. □

This bound is independent of the operating frequency and afterward optimization methodologies and affected only by the interconnect configurations. Intuitively, designs with larger bounds are superior to the ones with smaller bounds since the afterward optimization methodologies could possibly achieve those bounds.

2.1.2. Problem Definition

We formulate the *Floorplanning for Processing Rate* (FPR) problem as follows.

Problem 2.1 (Floorplanning for Processing Rate)**.** *In a directed graph $G = (V, E)$, every vertex represents a pin in a module with given width and height; every edge e represents a wire connecting two pins. Two weights are assigned to each wire e: $w(e)$ is the number of the flip-flops on e; $d(e)$ is the delay of the wire e. It is asked to find a floorplan*

to maximize the processing rate bound,

$$\frac{1}{\lambda_G} = \min_{cycle\ C\ in\ G} \frac{\sum_{e \in C} w(e)}{\sum_{e \in C} d(e)} \tag{2.2}$$

2.2. Floorplanning for Processing Rate Optimization

2.2.1. ACG Floorplanning

Adjacent Constraint Graph (ACG) [**10**] is a representation for general floorplans. It is a constraint graph containing horizontal and vertical constraint edges. ACG simplifies the classical horizontal and vertical constraint graph by removing redundancies through three conditions: first, there is exactly one relation between any pair of modules; second, no transitive edge is allowed; third, there is no cross which is an edge configuration as shown in Figure 2.1. Allowing crosses in the graph may introduce quadratic number of edges as shown in Figure 2.2.

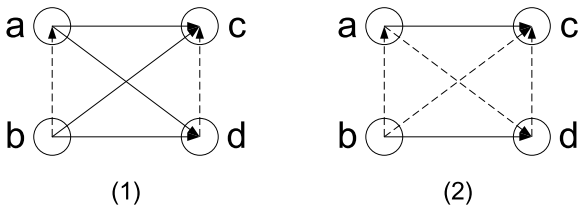

Figure 2.1. (1) Horizontal cross; (2) vertical cross.

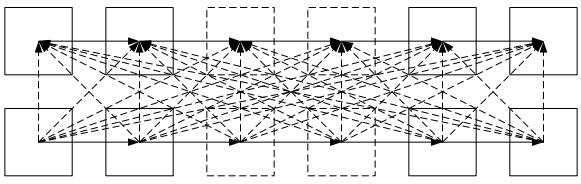

Figure 2.2. A constraint graph with quadratic number of edges.

To maintain those conditions, *Reduced ACG* is proposed in [**2**] according to the following property of ACG. As shown in Figure 2.3, the edges starting from a vertex are divided into four classes: class 1, edge to the adjacent vertex; class 2, edges in the same group (horizontal or vertical) as the class 1 one to the following vertices; class 3, the first edge in the group different from the class 1 one; class 4, the remaining edges, where every edge must be in the group different from the previous one. Reduced ACG is obtained by removing all the class 4 edges from ACG and there is an one-to-one mapping between ACG and Reduced ACG. Operations on Reduced ACG make only local changes to the graph and map to local perturbations in physical space. Figure 2.4 shows an example for the floorplan, ACG, and Reduced ACG.

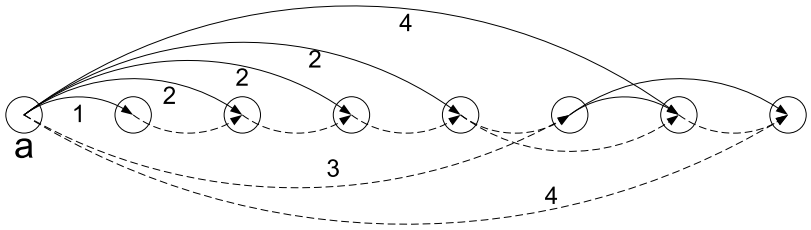

Figure 2.3. Edges starting from a are divided into 4 classes in ACG.

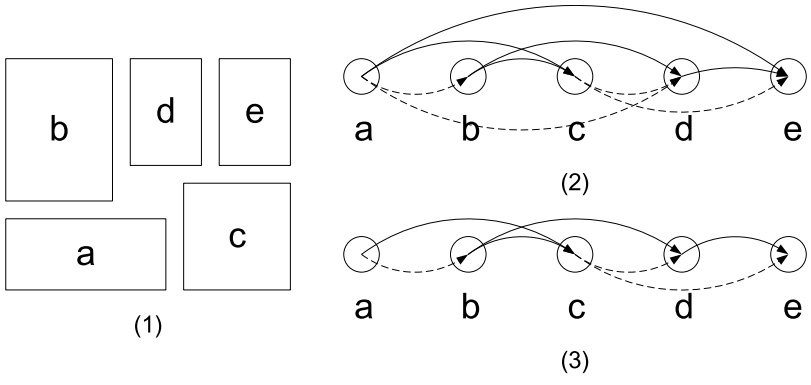

Figure 2.4. An example shows (1) floorplan; (2) ACG; (3) Reduced ACG.

The SA based floorplanner used in [**2**] is extended to optimize the processing rate by combining the processing rate bound into the cost function. When a floorplan is obtained during SA, the physical locations of modules are computed and the delays of the interconnects are calculated as the Manhattan distance between modules. The bound $\frac{1}{\lambda_G}$ is computed directly as described in the following sections. We also develop a method to address the fixed-outline constraint in the cost function.

2.2.2. Direct Bound Evaluation

Computing $\frac{1}{\lambda_G}$ is actually solving the minimum cycle ratio problem.

Given a strongly connected directed graph $G = (V, E)$ with two edge weight $w_1(e)$ and $w_2(e) > 0$ for each $e \in E$, the minimum cycle ratio problem is to compute the following minimum cycle ratio

$$\phi_{min} = \min_{cycle\ C\ in\ G} \frac{\sum_{e \in C} w_1(e)}{\sum_{e \in C} w_2(e)} \tag{2.3}$$

When $w_2(e) = 1$ for any edge e, it becomes the minimum cycle mean problem. According to [**27**], Howard's algorithm is the fastest one in practice to solve the minimum cycle mean problem. The version presented in their work is a simplified one from [**26**]. Based on the discussions in [**26**], we modify the implementation of Howard's algorithm in Section 2.5 of [**27**] to solve the minimum cycle ratio problem.

The intuition behind Howard's algorithm is to maintain a policy graph G_π through the computation. A policy graph is a sub-graph of G where there is exactly one edge starting from any vertex. The cycles in a policy graph can be enumerated since there is exactly one cycle in each weakly connected component of G_π. The minimum cycle ratio ϕ of G_π is obtained then, which is obviously an upper bound of ϕ_{min}. It can be asserted that

$\phi = \phi_{min}$ if there is no negative cycle in G regarding to the edge weights $w_1(e) - w_2(e)\phi$. If there are negative cycles, one of them is identified in a newly constructed policy graph; this cycle has a cycle ratio less than ϕ. A vertex labeling is maintained to interleave the above two steps, i.e., to check for the negative cycles and to construct a new G_π.

In order to evaluate $\frac{1}{\lambda_G}$ of a system G given its floorplan, the graph G is first decomposed into strongly connected components (SCC). Only one such computation is needed per benchmark and the *Strongly–Connected–Components* algorithm in [**29**] works well here. We assign $d(e)$ to be the $w_2(e)$ and $w(e)$ to be the $w_1(e)$. By applying Howard's algorithm for minimum cycle ratio in each SCC and picking up the minimum among them, we obtain $\frac{1}{\lambda_G}$.

2.2.3. Incremental Bound Evaluation

In Howard's algorithm, different initial G_π's affect the number of the iterations and thus the running time. Intuitively, an initial G_π with a smaller ϕ tends to converge quicker than the one with a larger ϕ.

In our floorplanners, the floorplan does not change too much between successive steps in SA because of those local Reduced ACG perturbations. The final G_π's of the previous floorplan obtained by Howard's algorithm give near-optimal ϕ's. We reuse those final G_π's as our initial policy graphs for each SCC. For the first floorplan when those G_π's are not available, we follow [**27**] to construct it by choosing the edge with the minimum w_1 weight starting from each vertex.

As shown in Section 2.3, this incremental technique speeds up the floorplanner by 29% on average.

2.2.4. Handle the Fixed-outline Constraint

Following [**25**], the fixed-outline constraint is modeled with two constants. One is the maximum white-space fraction γ and the other is the aspect ratio $\alpha \geq 1$. Suppose the total area of all the modules are A. The desired width and height of the fixed-outline floorplan are computed as

$$H_* = \sqrt{(1+\gamma)A\alpha}, W_* = \sqrt{(1+\gamma)A/\alpha} \tag{2.4}$$

In [**25**], several approaches were proposed to handle the fixed-outline constraint. Addressing the constraint directly in the cost function was shown to be not successful for a classical SA based sequence-pair floorplanner. Adding slack-based moves was proposed to solve this problem. These moves change the aspect ratio of the floorplan toward the desired one. By applying these moves when the aspect ratio is not the desired one, a floorplan satisfying the fixed-outline constraint may be obtained during SA.

Instead of designing a new move and mixing it to the existing ones, we propose a way to handle the fixed-outline constraint directly in the cost function. For a floorplan with width W and height H where $H \geq W$, the cost associated with the fixed-outline constraint is defined as

$$outline\_cost = e^{\max(\frac{W}{W_*},1)+\max(\frac{H}{H_*},1)-2} \tag{2.5}$$

This cost is larger than 1 if the constraint is not met; when the constraint is satisfied, the cost becomes 1. The total cost of a floorplan is calculated as the product of the *outline_cost* and the other cost. The intuition is that the exponential function is very

sensitive when the outline difference is large but less sensitive when the outline difference is small. So the floorplanner will push the floorplan hard toward the desired outline when there is enough whitespace. When the outline constraint is almost or totally satisfied, optimizing other cost becomes the priority.

It is possible during SA some floorplans meet the fixed-outline constraint but the final floorplan do not meet the constraint. So the best floorplan meeting the fixed-outline constraint is recorded during SA and is reported at the end.

2.3. Experimental Results

2.3.1. Experimental Setup

The experiments are conducted on a Windows machine with 1.4GHz Pentium M processor and 512M memory where the floorplanner is implemented using C++ and compiled with GCC 3.4.2.

Following the experimental setup in [**17**], we pick up GSRC benchmarks including n10, n30, n50, n100, n200, and n300. The signal directions and flip-flops are derived as follows according to [**17**]. The last pin of a net is treated as the source of the net and then the net is broken into two-pin nets. All the pins belong to one module are treated as one pin located at the center of the module so that the modules could be treated as gates. There is exactly one flip-flop on each edge to represent the situation where each module latches its outputs.

Classical MCNC benchmarks were used in [**17**] as well. However, as the sources of the nets were determined randomly in their work, we cannot generate the same configuration and thus we only experiment on GSRC benchmarks.

2.3.2. Results for Floorplanning for Processing Rate

We first test the floorplanner without fixed-outline constraint. The cost function used is $4\sqrt{area} + \lambda_G$ and the incremental bound evaluation method is employed. We run each benchmark for ten times and the best one is reported in Table 2.1 where the "ws (%)" column shows the white space in percentage.

For each floorplan, we record the random seed used and use it to perform another SA floorplanning without the incremental bound evaluation. The resulting floorplan is the same but the running time is different. The experimental results are reported in Table 2.2. For the incremental one and the non-incremental one (under "non-incre."), we report the running time as well as the total number of iterations (in "#iter.") of Howard's algorithm. Improvements in running time are reported in "impr." column and the average improvement is 29%.

circuit	area	ws (%)	λ_G	time (sec)
n10	240.6K	7.9	292.8	47.9
n30	220.9K	5.6	296.0	46.5
n50	217.1K	8.5	239.5	45.7
n100	196.0K	8.4	185.0	56.2
n200	195.6K	10.2	373.9	475.9
n300	312.7K	12.7	497.3	540.4

Table 2.1. Results of floorplanning for processing rate (FPR).

To compare our results with those in [**17**], the throughput for our final floorplans under different frequencies are computed. The frequencies are modeled by the *critical length*, which is the distance that a signal travels in one clock cycle. The corresponding critical lengths are 30%, 50%, 70%, and 100% of the square root of the total area of all the

circuit	incremental		non-incre.		impr. (%)
	time	#iter.	time	#iter.	
n10	47.9	8.9M	56.9	9.6M	15.8
n30	46.5	4.1M	65.0	5.3M	28.5
n50	45.7	3.8M	69.2	4.9M	34.0
n100	56.2	4.4M	75.4	5.5M	25.5
n200	475.9	5.1M	740.6	7.0M	35.7
n300	540.4	4.5M	827.6	6.7M	34.7
average improvement					29.0

Table 2.2. Incremental vs. non-incremental bound evaluation.

circuit	method	100%	70%	50%	30%	time (sec)
n10	FPR	0.200/7.9	0.200/7.9	0.333/7.9	0.625/7.9	47.9
	[**17**]	0.166/8.9	0.250/7.2	0.500/4.0	0.636/6.2	60.0
n30	FPR	**0.167/5.6**	**0.375/5.6**	**0.444/5.6**	0.650/5.6	46.5
	[**17**]	0.200/6.9	0.375/8.3	0.500/6.8	0.636/8.5	60.0
n50	FPR	0.167/8.5	0.250/8.5	0.400/8.5	0.588/8.5	45.7
	[**17**]	**0.133/7.2**	0.375/6.9	0.473/6.7	0.636/7.2	60.0
n100	FPR	**0.000/8.4**	**0.188/8.4**	**0.333/8.4**	**0.500/8.4**	56.2
	[**17**]	0.000/9.1	0.200/9.1	0.375/8.6	0.500/9.7	60.0
n200	FPR	0.364/10.2	0.514/10.2	0.600/10.2	0.722/10.2	475.9
	[**17**]	-	-	-	-	-
n300	FPR	0.400/12.7	0.500/12.7	0.620/12.7	0.750/12.7	540.4
	[**17**]	-	-	-	-	-

Table 2.3. Throughput, white space, and running time: FPR (PM 1.4GHz) vs. [**17**] (PIII 1.4GHz).

circuit	u.b.	100%	70%	50%	30%
n10	3.42 *	1.70	2.43	2.83	2.65
n30	3.38	1.82	1.95	2.43	2.55
n50	4.18	1.87	2.40	2.69	3.08
n100	5.41	2.36	2.74	3.15	3.93
n200	2.67	1.52	1.66	1.91	2.21
n300	2.01	1.15	1.37	1.45	1.59

* All the numbers shown are in the unit $\times 10^{-3}$.

Table 2.4. Processing rate vs. upper bound.

modules. According to Section 2.1, we calculate $w^*(e)$ first. Then by applying Howard's algorithm the same way as computing $\frac{1}{\lambda_G}$, the throughput is obtained.

In [**17**], the experiments were conducted on a 1.4GHz Pentium III machine. For each circuit among n10, n30, n50, n100 and each of those critical lengths, they ran the experiment for 60 seconds ten times and we copy the best ones. They did not report the results for n200 and n300 but we report our results for these two benchmarks. The results are compared in Table 2.3 with the format $1-throughput/white\ space$ (%), which means that the smaller the number, the better. The dominating solutions in throughput and white space are highlighted. It can be seen that our results dominate theirs for about half of the cases and are not dominated by theirs for the other half. Moreover, since the machine configurations are comparable, our running times are much better considering [**17**] spent $60 \times 4 = 240$ seconds for each benchmark.

In Table 2.4, we report the processing rates for each critical lengths as well as the bounds $\frac{1}{\lambda_G}$ (in "u.b."). Although the bounds tend to be looser for larger critical lengths, we believe that the fidelity of the bound to the throughput makes our approach effective.

2.3.3. Results for Fixed-outline Floorplanning for Processing Rate

The fixed-outline constraint is handled explicitly by the cost function. The cost function used is $outline\_cost \times (w\sqrt{area} + \lambda_G)$. The area weight w is set to 0.5 for n10, n30, n50, and n100 and 2 for n200 and n300. Small w gives better processing rate bound but if meeting the fixed-outline constraint is a problem, w is increased to emphasize better area. We run our floorplanner with the maximum white-space of $\gamma = 15\%$ and the aspect ratio

of $\alpha = 1$, $\alpha = 1.5$, and $\alpha = 2$ respectively. The best results from ten runnings are reported in Table 2.5. The numbers are in the format $\lambda_G/running\ time\ (sec)$.

circuit	$\alpha = 1.0$	$\alpha = 1.5$	$\alpha = 2.0$
n10	282.0/51.9	282.5/51.2	303.8/65.5
n30	248.3/60.5	254.4/55.0	269.0/55.0
n50	209.0/62.7	210.7/59.2	221.1/60.1
n100	144.2/121.9	138.0/119.4	140.7/123.1
n200	358.2/637.8	378.8/620.6	382.5/863.9
n300	494.0/1091	548.4/840.2	539.8/830.9

Table 2.5. Results of fixed-outline floorplanning for processing rate.

2.4. Summary

In this chapter, we showed that optimizing the processing rate bound, which is the minimum ratio of the flip-flop number to the delay in any cycle, is more general than either optimizing the clock period or the throughput for a sequential system because the bound is independent of the operating frequencies or the available design methodologies. We built a SA based floorplanner optimizing for the processing rate bound by evaluating it directly in the inner loop of SA without introducing much overhead in running time. Moreover, exploiting the incremental structure of the evaluating algorithm sped up the evaluating process. Experimental results con- firmed the effectiveness of our approach.

CHAPTER 3

Exploring Adjacency in Floorplanning

Many years of research have been done on floorplanning. Generally speaking, in a floorplan, rectangular *block*s are required to be arranged in a rectangular bounding box. A block may be hard, which means the shape is fixed, or soft, which mean only the area is fixed. Floorplanning, in its original meaning [**30**], focuses on realizing adjacency relations for soft blocks. Grason graphs [**31**] and rectangular dualization techniques [**32, 33, 34**] were developed to address the requirement for adjacency. The floorplanning flow commonly started with a graph that describes the connectivities between blocks, which was also known as the structure graph [**34**]. The edges in the graph were the desired adjacency relations. The floorplan that realizes those adjacency relations does not exist most of the time. Thus, the graph was planarized and properly triangulated such that the resulting graph has a rectangular dual. There were algorithms with linear complexity to identify the graphs that have rectangular duals and to construct the rectangular duals [**33**]. Several perturbations were designed in the work [**34**] such that the adjacency graph could be used in iterative algorithms like simulated annealing. However, those algorithms were still complicated and not widely used today.

Then, floorplan representations emphasizing more on placement than floorplanning [**30, 35**] were developed. Compared to floorplanning, placement focuses on placing hard blocks without overlap, possibly with a pre-defined bounding box. Most of the time,

simulated annealing is used to improve the floorplan according to a cost function via randomized perturbations. Those approaches were believed to be packing centric instead of connectivity centric [**35**] such that interconnects were only addressed by the cost function. The drawback was that although searching for a floorplan with the least white-space was efficient by using area as the cost function, including the interconnects in the cost function may result in large time overheads in evaluating the cost function. The overheads were worsen with the increasing of the problem sizes. Accuracy of interconnect estimation could also be affected since a less accurate method would be used in order to shorten the total running time. These overheads were confirmed by a recent work [**36**]: even when the most simple half-perimeter wire length (HPWL) model was used in interconnect estimation, the HPWL evaluation itself spent 90% of the total running time in the simulated annealing process for problems with 100 to 300 blocks.

So, floorplanning with adjacency relations is still preferable for connectivity-centric methodologies. Returning to the rectangular dual approaches, despite the complexity involved, several issues must be addressed before their applications to current floorplanning problems. The rectangular dual consists of rectangular *room*s that contain blocks. In some situations, the requirement of keeping adjacency relations may result in rooms that are significantly larger than the blocks contained by them, which in turn enlarges the whitespace in the floorplan. One such case is that when a small block has several large neighbors, the room containing the small block should be large enough to realize the adjacency relations. Another issue is that although blocks were thought to be soft such that they could be fit into rooms, hard blocks are common in today's floorplanning problems with the introduction of hard Intellectual Property (IP) cores. We propose to

allow more freedom in *NOT* keeping the adjacency relations to achieve small whitespace. The intuition behind this is that when the whitespace is small, two blocks are close to each other if there are only a few blocks separating them in comparison to the situation that adjacency relations are kept but the whitespace is large. The freedom allows us to design perturbations targeting at separating area and interconnect optimizations: first, the area optimization will not change the adjacency relations dramatically such that good relative positions for interconnects are preserved; second, the overheads of interconnect estimations are only added to the interconnect optimization.

Besides previous works on adjacency graphs [**31, 32, 33, 34**], Adjacent Constraint Graph [**10**] is recently proposed to help exploring the adjacency in floorplans. ACG tries to approximate the adjacency graph by removing redundancies in the constraint graph. However, since it is still a constraint graph, there are edges not between adjacent blocks. In this work, we focus on developing a representation based on the adjacency graph but less complicated for maintaining and optimizing compared to previous adjacency graph approaches. Our contributions include: first, *Constrained Adjacency Graph* (CAG) is proposed as an adjacency graph representation for floorplanning problems; second, sufficient and necessary conditions of CAG are presented as well as a linear complexity algorithm for constructing *dissected floorplan*s (defined in Section 3.1) from CAGs; third, a "tree-weaving" algorithm and an iterative packing heuristic are developed to improve a CAG in area without changing the adjacency relations dramatically. The practical usages of CAG are confirmed by the experiments on floorplanning problems with 100 to 300 blocks. The experiments start with a CAG generated from quadratic programming using the interconnects. A randomized greedy improvement heuristic that uses a cost function

of the weighted sum of the area and the HPWL is applied to improve the CAG. The results show that better floorplans are found with much less running time compared to an up-to-date simulated annealing floorplanner based on sequence pairs [**25**].

The rest of this chapter is organized as follows. In Section 3.1, CAG is defined and the sufficient and necessary conditions are presented. Then the algorithm to construct dissected floorplans from CAGs are described in Section 3.2. Packing based whitespace reduction techniques including the "tree-weaving" algorithm and the iterative packing heuristic are developed in Section 3.3. Experimental results are given in Section 3.4, Section 3.5 concludes the chapter.

3.1. Constrained Adjacency Graph

3.1.1. Definitions

Given a bounding box, we can dissect it into rectangular *room*s by horizontal and vertical segments. There should be no overlap of the rooms and no empty space outside the rooms. We call this dissection a *dissected floorplan* if every room accommodates exactly one block and there is no degenerated topology where four rooms share a common point. An adjacency graph whose vertices represents the rooms can be constructed corresponding to the dissection: there is one edge connecting two vertices iff the two rooms are adjacent. The adjacency graph is a planar graph and when the dissection is a dissected floorplan, all the faces in the adjacency graph are triangles.

Constrained Adjacency Graph (CAG), as indicated by its name, extends the adjacency graph corresponding to a dissected floorplan by adding constraints to its edges. More formally,

Definition 3.1 (Constraitned Adjacency Graph)**.** *Suppose $G = (V, E)$ is a directed graph with vertices representing rooms and the edges representing adjacencies. There are two types of edges: a vertical edge from b to t means the room t should be touched from bottom by the room b; a horizontal edge from l to r means the room r should be touched from left by the room l.*

The graph G is a Constrained Adjacency Graph *(CAG) iff the is a dissected floorplan such that there is an edge connecting two vertices iff the two rooms are adjacent and the edge describes the adjacency relationship between them.*

An example of a dissected floorplan along with its CAG is shown in Figure 3.1.

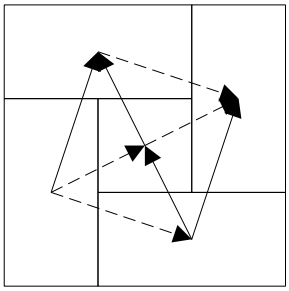

Figure 3.1. A dissected floorplan with 5 rooms along with its CAG. Solid arrows are vertical adjacency relations and dashed arrows are horizontal ones.

In comparison to grason graphs [**31**], CAG does not need to handle the degenerated topology; comparing to rectangular dualization techniques [**32, 33, 34**], CAG explicitly adds constraint to the adjacency relations. These two merits make maintaining and optimizing CAGs less complicated and details will be given in the following sections.

3.1.2. Sufficient and Necessary Conditions

The necessary conditions for a graph G to be a CAG can be obtained by inspecting a dissected floorplan. Straightforwardly, a CAG should be a directed acyclic graph (DAG) and every vertex in a CAG is reachable from the vertex representing the room at the bottom-left corner. The other conditions apply to every single vertex as illustrated in Figure 3.2. The detail follows.

Let E_V be the set of the vertical edges and E_H be the set of the horizontal edges. The vertical and horizontal subgraphs of G are defined as $G_V = (V, E_V)$ and $G_H = (V, E_H)$ respectively. Obviously both G_V and G_H are directed acyclic graphs (DAG). A *horizontal/vertical path* is a path whose edges are all horizontal/vertical ones. Since G is a DAG, every horizontal/vertical path is simple, i.e., it never passes one vertex more than once.

For an arbitrary vertex v, its *top edge*s are all the vertical edges leaving it and its *top neighbor*s are the vertices at the end of its top edges. From dissected floorplans, it is true that if v has at least one top neighbor, there is a unique horizontal path, denoted by $P_{top}(v)$, exactly containing all the top neighbors. Therefore, the *left-most top neighbor* and the *right-most top neighbor* can be defined as the starting vertex and the ending vertex of $P_{top}(v)$ respectively. The above definitions can be easily extend to the other three boundaries. The following lemmas hold for v in a CAG.

Lemma 3.1 (Neighbor Condition for v)**.** *Suppose $P_{top}(v) = (v_1, v_2, \ldots, v_m)$. Then v_i is the bottom-most left neighbor of v_{i+1} and v_{i+1} is the bottom-most right neighbor of v_i for all $1 \leq i < m$. For $P_{bottom}(v)$, $P_{left}(v)$, and $P_{right}(v)$, there are similar results.*

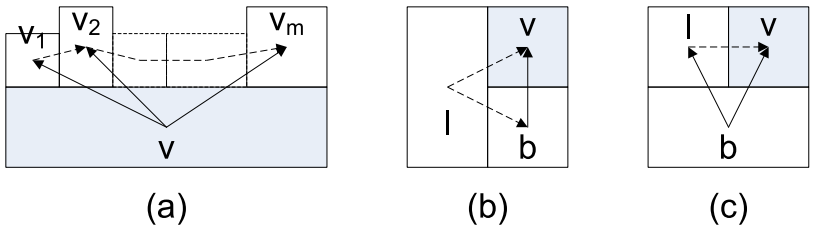

Figure 3.2. (a) Neighbor condition for v; (b) (c) Corner condition for v.

Proof. This condition must hold as illustrated in Figure 3.2 (a). □

Lemma 3.2 (Corner Condition for v)**.** *The room v's top-left corner is not on the boundary of the dissected floorplan iff it has both top and left neighbors. Suppose the left-most top and the top-most left neighbor of v are t and l respectively. Then either l is the bottom-most left neighbor of t or t is the right-most top neighbor of l. There are similar results for the other three corners.*

Proof. This condition must hold as illustrated in Figure 3.2 (b) and (c). □

These conditions are also sufficient for a graph to be a CAG, as stated in the following theorem.

Theorem 3.1. *A directed graph $G = (V, E)$, whose edges E are divided into vertical edges E_V and horizontal edges E_H, is a CAG iff*

(1) *G is a DAG and there is a vertex such that every vertex is reachable from it.*

(2) *The neighbor condition holds for every vertex.*

(3) *The corner condition holds for every vertex.*

Proof. According to Lemma 3.1 and 3.2, the above conditions are necessary. The proof for the sufficient part will be given Section 3.2 by constructing dissected floorplans from the directed graphs satisfying those conditions. □

3.2. Dissected Floorplan from CAG

When the block sizes, which are also the minimal sizes of the rooms, are given along with the CAG, we design an $O(n)$ algorithm to construct a dissected floorplan satisfying all the adjacency constraints with minimum area as presented below. Since only the conditions in Theorem 3.1 are used to derive the algorithm, the algorithm proves the conditions in Theorem 3.1 are sufficient.

For a vertex v, suppose that the width of the block to be accommodated in the room v is $v.width$ and the height is $v.height$. Define $(v.left, v.bottom)$ to be the coordinates of the bottom-left corner of the room v and $(v.right, v.top)$ to be the top-right corner. Assume $(0, 0)$ is the bottom-left corner of the floorplan and (W, H) is the top-right corner. The algorithm will determine the coordinates in the horizontal direction and the ones in the vertical direction separately. Because of the geometric symmetry, it is enough to focus on the algorithm *H-CAG-Place* that determines the coordinates in the horizontal direction only, i.e., $v.left$ and $v.right$ for all v as well as W. The algorithm is shown in Figure 3.3 and the details follow.

First of all, a dummy vertex $Head_V$ and corresponding edges are added such that $Head_V$ is the bottom neighbor of all the vertices in G that do not have a bottom neighbor yet. The following lemma can be proved.

Algorithm H-CAG-Place

Inputs

A CAG G.

Outputs

$v.left$, $v.right$, and W.

```
1  Add Head_V.
2  Head_V.left ← 0.
3  Order the vertices by their discovery times in a DFS of G_V from Head_V.
4  For each vertex v except Head_V following the order in 3:
5    Compute v.left according to the cases as shown in Figure 3.5.
6  Compute W using Equation (3.4).
7  Compute all the v.rights using Equation (3.5).
```

Figure 3.3. The H-CAG-Place algorithm.

Lemma 3.3. *The neighbor condition for $Head_V$ holds and every vertex in G is reachable from $Head_V$ through only vertical edges.*

Proof. Consider a dummy block below the current floorplan and whose width is the same as the current floorplan. The neighbor condition for $Head_V$ holds since it can represent the dummy block. As there is no degenerated topology where four rooms share a common point, any block can be reached from the dummy block through vertical adjacency relations, i.e., every vertex is reachable from $Head_V$ through only vertical edges. □

Now by performing a depth-first-search (DFS) in G_V starting from $Head_V$ and visiting top neighbors from left to right for each vertex, the vertices can be sorted in the order of their discovery times, i.e., the times they are first discovered in the DFS. An example of a CAG with the dissected floorplan as well as the depth-first tree with the ordering are shown in Figure 3.4 This order is defined as the *V-CAG order*. (Similarly the *H-CAG order* can be defined.)

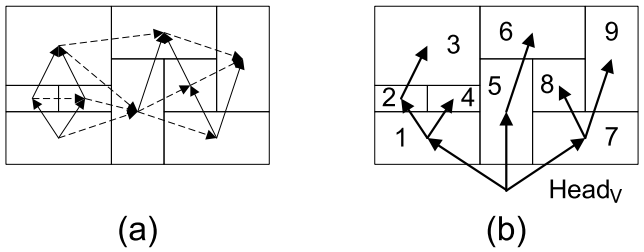

Figure 3.4. (a) A CAG with the dissected floorplan; (b) The depth-first tree with vertices labeled according to the V-CAG order.

Then, the $v.left$s for each v is determined by dynamic programming following the V-CAG order. The $Head_V.left$ is set to 0 at the beginning. Since every vertex v except $Head_V$ has a bottom neighbor, there are three cases for calculating each $v.left$ as shown in Figure 3.5. In the first case, the vertex v does not have a left neighbor. So the room v should be on the left boundary of the floorplan, i.e.

$$v.left = 0 \tag{3.1}$$

In the second case, the vertex v have both left and bottom neighbors where the bottom-most left neighbor l of v is the top-most left neighbor of the left-most bottom neighbor b of v. Now the vertex v should have the same left boundary as b, i.e.,

$$v.left = b.left \tag{3.2}$$

In the third case, b should be the right-most bottom neighbor of l because of the corner condition. A series of vertices $l_1, l_2, \ldots, l_k$ can be find such that $l_1 = l$, l_{i+1} is the right-most top neighbor of l_i and l_i is the right-most bottom neighbor of l_{i+1} for all $1 \leq i < k$, and either l_k does not have a top neighbor or its right-most top neighbor t does not have l_k as its right-most bottom neighbor. When l_k does not have a top neighbor, a dummy

vertex t with $t.left = 0$ can be added temporarily to simplify the following procedure. Now, $v.left$ should be no less than $l_i.left + l_i.width$ for all $1 \leq i \leq k$ since they are on the two sides of a vertical segment. Fro the vertex b and t, $v.left$ should be no less than $b.left$ and $t.left$ to satisfy the T-junctions formed by the vertical segment and the top side of b and the bottom side of t. So, the $v.left$ in an area optimal dissected floorplan should be

$$v.left = \max\{b.left, \max_{1 \leq i \leq m}(l_i.left + l_i.width), t.left\} \tag{3.3}$$

The following lemmas can be proved for the validity and the complexity of the dynamic programming.

Lemma 3.4. *In the V-CAG order, for each Equation (3.2) and (3.3), the vertices appearing at the right side come before the vertex appearing at the left side.*

Proof. The lemma holds since the V-CAG order is obtained by performing DFS starting from $Head_V$ and visiting top neighbors from left to right for each vertex. □

Lemma 3.5. *Each vertex v appears at most $m_{top} + 2$ times in Equation (3.1), (3.2), and (3.3) where m_{top} is the number of the top neighbors of v.*

Proof. As shown in Figure 3.5, each vertex appears once at the left side, once as t, and m_{top} times as b and l_i in Equation (3.1), (3.2), and (3.3). □

The next step is to calculate W. This is straightforward since

$$W = \max_{v \in V}(v.left + v.width) \tag{3.4}$$

must be true for an area optimal floorplan.

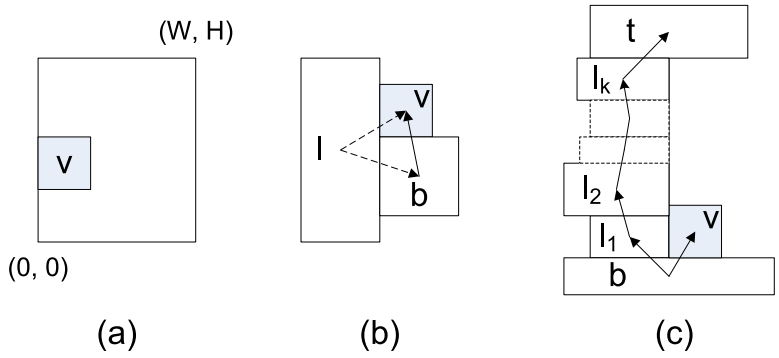

Figure 3.5. Three cases for calculating $v.left$.

Finally, all the right boundaries are determined as follows.

$$\forall v \in V, v.right = \begin{cases} r.left, & \text{if } v \text{ has a right neighbor } r, \\ W, & \text{if not.} \end{cases} \tag{3.5}$$

The algorithm called *V-CAG-Place* that determines the coordinates in the vertical direction can be derived similarly. In summary, the corresponding minimal area dissected floorplan is constructed by the H-CAG-Place algorithm and the V-CAG-Place algorithm, as stated in the following theorem.

Theorem 3.2. *Once the CAG and the minimal room sizes are given, applying the H-CAG-Place and the V-CAG-Place algorithm constructs a dissected floorplan with minimal area and satisfying the adjacency constraints. The algorithms consume $O(n)$ time and space where n is the number of rooms.*

Proof. The correctness of the algorithm is implied by Lemma 3.4. The time and space complexities of the algorithm are implied by Lemma 3.5. □

3.3. Whitespace Reduction via Packing

Rooms could be much larger than the contained blocks because of the requirements on adjacencies. In this sections, we will present techniques that reduce whitespace without change the adjacency relations dramatically.

3.3.1. Packing of Dissected Floorplans

As shown in Figure 3.6, some blocks in a dissected floorplan can be pushed downward since there are vertical vacancies in the rooms below them. The resulting floorplan is no longer a dissected floorplan. We call it the *V-packing* of a dissected floorplan. Similarly, the *H-packing* of a dissected floorplan is obtained by pushing all the blocks leftward.

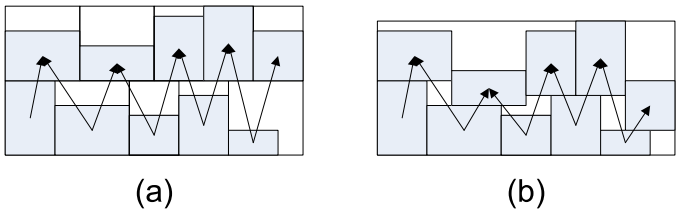

Figure 3.6. (a) A dissected floorplan with the vertical subgraph G_V; (b) The V-packing of the dissected floorplan. Both rooms and blocks (gray ones) are shown here.

Since G_V describes the vertical adjacency relations, the above pushing downward is formulated as the *V-CAG-Pack* algorithm as shown in Figure 3.7. There is a *H-CAG-Pack* algorithm as well.

The V-CAG-Pack algorithm will not generate overlap because in a dissected floorplan, if there are two rooms whose projections to the $x - axis$ are overlapped for a segment longer than 0, there is a vertical path from one of them to the other. This is stated as the

Algorithm V-CAG-Pack

Inputs

A CAG G.

Outputs

$v.bottom$, $v.top$, and H.

```
1  Add Head_V.
2  Assign v.height as the edge weight for every vertical edge (u, v).
3  For every vertex v except Head_V:
4    v.bottom ← the length of the longest path from Head_V to v in G_V.
5    v.top ← v.bottom + v.height.
6  H ← max_{v∈V, v≠Head_V} v.top.
```

Figure 3.7. The V-CAG-Pack algorithm.

following lemma which includes the horizontal direction for the H-CAG-Pack algorithm as well.

Lemma 3.6. *For two vertices u and v, if $u.left < v.left < u.right$ or $v.left < u.left < v.right$, there is a vertical path from u to v or one from v to u; if $u.bottom < v.bottom < u.top$ or $v.bottom < u.bottom < v.top$, there is a horizontal path from u to v or one from v to u.*

Proof. Consider the dissected floorplan corresponding to the CAG. Assume $u.left < v.left < u.right$. Then since u and v should not overlap, either $u.bottom \geq v.top$ or $v.bottom \geq u.top$. In both cases, we can reach one of them from the other following vertical adjacency relations, i.e, a vertical path from u to v or one from v to u. Similarly, the lemma holds for all the other cases. □

In summary, given a CAG G, the V-packing of the dissected floorplan, written as $VP(G)$, is obtained by applying the V-CAG-Pack and the H-CAG-Place algorithm; the

H-packing, written as $HP(G)$, is obtained by applying the H-CAG-Pack and the V-CAG-Place algorithm.

3.3.2. Packed Dissected Floorplans

In the V-packing of a dissected floorplan, since the blocks are not packed along the horizontal direction, there are still possibilities for large whitespace. Similarly, in the H-packing of a dissected floorplan, large whitespace may appear along the vertical direction. It is not easy to perform packing on both directions simultaneously. Intuitively, if a dissected floorplan can be constructed such that it is "packed" along the horizontal direction, the whitespace of the V-packing would not be significant. The idea is formalized as follows.

Definition 3.2 (Packed Dissected Floorplans)**.**

A dissected floorplan is H-packed if the H-CAG-Pack algorithm gives the same $v.left$ for every vertex v as the H-CAG-place algorithm. A dissected floorplan is V-packed if the V-CAG-Pack algorithm gives the same $v.bottom$ for every vertex v as the V-CAG-place algorithm.

In the V-CAG-Pack algorithm, since $v.bottom$ is calculated as the length of the longest path in G_V, a longest-path tree rooted at $Head_V$ can be identified, in which a vertex u is the parent of a vertex v only if (u, v) is the last edge on the longest path from $Head_V$ to v. We call this tree the *V-LP tree* of the CAG. Similarly the *H-LP tree* rooted at $Head_H$ is defined after the H-CAG-Pack algorithm.

Given a tree T rooted at $Head_H$ containing all the vertices, the *H-Tree-Weaving* algorithm as shown in Figure 3.8 creates a new CAG whose H-LP tree is T and the corresponding dissected floorplan is H-packed. The details follow.

Algorithm H-Tree-Weaving

Inputs

A tree T rooted at $Head_H$.

Outputs

A CAG G^* whose H-LP tree is T.

```
1  Head_H.left ← 0; Head_H.width ← 0.
2  Order the vertices by their discovery times in a DFS of T from Head_H.
3  For every vertex v except Head_H following the order in 2:
4    Find if the left-most bottom neighbor u of v exists and all the left
     neighbors of v according to Figure 3.10.
5    If u exists:
6      Find the bottom neighbors of v starting from u according to
7      Figure 3.11.
8      If (d) happens:
9        u ← y. Go to 6.
10 Finalize the G*.
```

Figure 3.8. The H-Tree-Weaving algorithm.

The new CAG G^* is created by adding vertices one by one following the order of their discovery times computed by a DFS on the tree T where the children of a vertex are visited from the bottom to the top. Keep G^* as a CAG during the algorithm is over-strict. Instead of that, G^* is kept as a *H-stepwise* CAG. As shown in Figure 3.9, the H-stepwise CAG relaxes the constraints along the right boundary of the bounding box: instead of one vertical path from the bottom to the top along the right boundary, multiple vertical paths joined by horizontal paths are allowed, e.g., vertical paths from f to e, from d to c, and from b to a are joined by horizontal paths from d to e and from b to c. In addition, blocks sizes are taken into consideration such that the rooms represented by the end point of those vertical paths can be extended to the top boundary, e.g., $d.left + d.width$ is no less than $v.left + v.width$ for every v on the path from d to c and $b.left + b.width$ is no less than $u.left + u.width$ for every u on the path from b to a. When all the vertices are

added to G^*, it is *finalized* into a CAG. This finalization process can be divided into three steps. First, a dummy vertex assumed to have $Head_H$ as its parent with infinite width is added to G^* just like adding any other vertex. It is actually placed on top of all the original vertices. Second, this dummy vertex along with any edges connected to it are removed. Finally, the rooms along the right boundary are extended horizontally to touch the boundary.

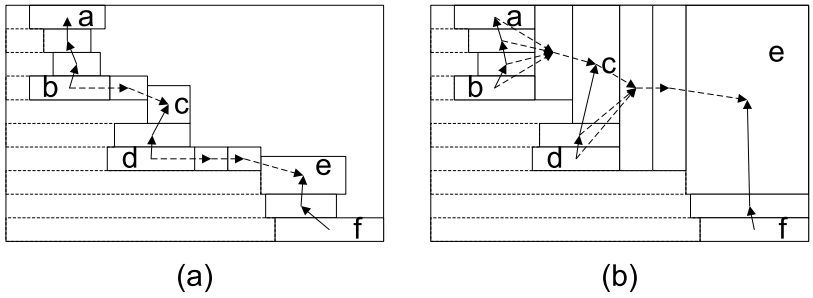

Figure 3.9. (a) A H-stepwise CAG; (b) Finalize the H-Stepwise CAG into a CAG.

The method to add a vertex v to G^* is as follows. Assume $Head_H.left = Head_H.width = 0$. Suppose v's parent is p. The $v.left$ is set to $p.left + p.width$. Then, the left-most bottom neighbor u of v is determined: if v has a bottom sibling, u is that bottom sibling; if v does not has a bottom sibling, u is found by following the right-most bottom neighbor starting from p such that $u.left + u.width > v.left$; if no such u exists, v won't have bottom neighbor. The left neighbors of v are found at the same time: for the first case, only p is the left neighbor; for the two latter cases, every vertex reached except u is a left neighbor. These three cases are shown in Figure 3.10.

After the left-most bottom neighbor u of v is found, all the bottom neighbor of v are found from left to right. There are four cases as shown in Figure 3.11. They all starts by identifying a horizontal path from u following the top-most right neighbors. In Figure 3.11

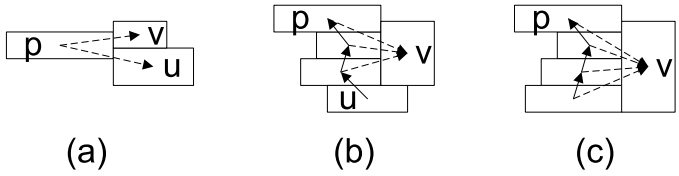

Figure 3.10. Find the left-most bottom neighbor u of v: (a) v has a bottom sibling u; (b) v does not have a bottom sibling but u can be found; (c) v has no bottom neighbor. Rooms are extended accordingly for clarity.

(a), the path ends at a vertex w satisfying that:

$$w.left < v.left + v.width < w.left + w.width$$

In Figure 3.11 (b) and (c), the path ends at a vertex w without a right neighbor satisfying that:

$$w.left + w.width \leq v.left + v.width$$

Then a vertical path to w is identified by following the right-most bottom neighbors. For (b), every vertex x on the path does not have a right neighbor and satisfying that:

$$x.left + x.width \leq v.left + v.width$$

For (c), there is a vertex x on the path such that:

$$v.left + v.width < x.left + x.width$$

In any of the above three cases, all the vertices on the horizontal path from u to w are added as v's bottom neighbor and G^* is kept as a H-stepwise CAG. For the fourth case in Figure 3.11 (d.1), there is a vertex x on the vertical path to w with a right neighbor

satisfying that:

$$x.left + x.width \leq v.left + v.width$$

Suppose the top-most right neighbor of x is y. As shown in Figure 3.11 (d.2), y can be extended vertically to touch v from the bottom. All the vertices on the vertical path from x to w except x should be added as the left neighbor of y on top of x. All the vertices on the horizontal path from u to w should be added as the bottom neighbor of v. Then, y is treated the same way as u and this whole process is repeated again until one of the first three cases is reached.

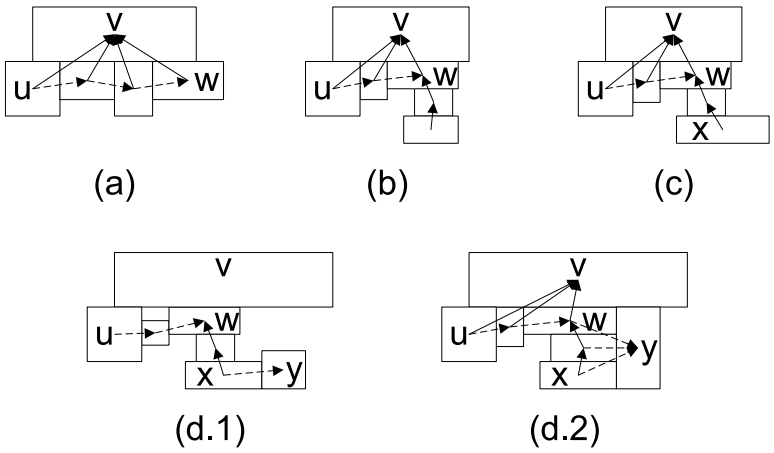

Figure 3.11. Four cases for finding all the bottom neighbors of v.

Similarly, the *V-Tree-Weaving* algorithm can be derived. The following theorems state the correctness of the algorithms.

Theorem 3.3. *The H-Tree-Weaving algorithm creates a H-packed dissected floorplan with the given H-LP tree. The V-Tree-Weaving algorithm creates a V-packed dissected floorplan with the given V-LP tree.*

Proof. The H-Tree-Weaving algorithm first generates the H-stepwise CAG. It can be verified that both the neighbor and the corner conditions hold for all the vertices except the ones along the right boundary of the bounding box. The finalization process will then ensure that the conditions hold for all the vertices along the right boundary of the bounding box. Similarly, we can prove the correctness of the V-Tree-Weaving algorithm. □

3.3.3. Iterative Packing

Intuitively, when the H-LP tree is generated from a dissected floorplan whose H-packing contains little whitespace, the adjacency relations belonging to the H-LP tree are preserved and other adjacency relations will not be changed dramatically during the H-Tree-Weaving algorithms. On the other hand, the H-Tree-Weaving algorithm not only creates a H-packed dissected floorplan with a given H-LP tree: it relaxes some vertical adjacency requirements such that blocks are allowed to be pushed downward further in the V-packing. This also true for the V-Tree-Weaving algorithm and an example of applying them alternatively is shown in Figure 3.12.

Actually applying the two algorithm alternatively will reach an admissible placement as proposed along with the O-Tree representation [**37**]. In admissible placements, blocks cannot be pushed either leftward or downward overlapping-freely without moving other blocks. This motivates us to design the iterative packing heuristic as shown in Figure 3.13 that improves the CAG in area without changing the adjacency relations dramatically. The resulting floorplan of the iterative packing heuristic is alway calculated as $HP(G)$.

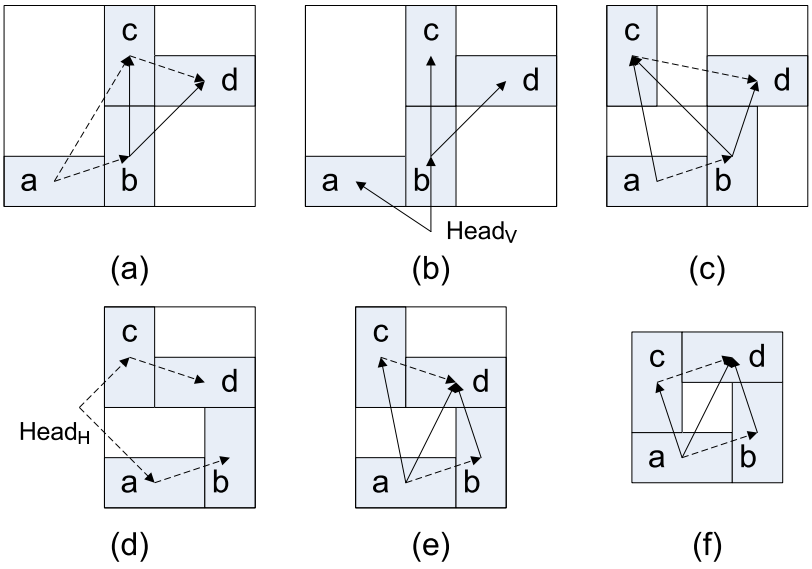

Figure 3.12. Improve a dissected floorplan by applying weaving and packing alternatively. (a) The initial dissect floorplan; (b) The V-packing and the V-LP tree of (a); (c) The dissected floorplan after V-Tree-Weave; (d) The H-packing and the H-LP tree of (c); (e) The dissected floorplan after H-Tree-Weave; (f) The V-packing of (e).

Algorithm Iterative Packing

Inputs

A CAG G.

Outputs

Improved CAG G.

1 Compute $HP(G)$, i.e., the the H-packing of the dissected floorplan.
2 $T_0 \leftarrow$ the H-LP tree; $W_0 \leftarrow$ the width.
3 Create a new CAG G_1 from T_0 by the H-Tree-Weaving algorithm.
4 Compute $VP(G_1)$.
5 $T_1 \leftarrow$ the V-LP tree; $H_1 \leftarrow$ the height.
6 Create a new CAG G_2 from T_1 by the V-Tree-Weaving algorithm.
7 **If** width of $HP(G_2) = W_0$ and and height of $HP(G_2) = H_1$: return.
8 $G \leftarrow G_2$. Go to line 1.

Figure 3.13. The iterative packing heuristic.

3.4. Experiments

3.4.1. CAG Floorplanning for Interconnects

Here we present the preliminary flow of CAG floorplanning for interconnects: an initial CAG is generated by quadratic programming and then optimized iteratively by a greedy heuristic.

Given a group of blocks with the interconnects and terminals (which are the fixed pins on the floorplan boundary), the initial CAG is generated as follows to have good interconnect characteristics. First, a single quadratic programming step depending on the interconnects is used to compute the relative positions of the blocks in the bounding box. The weights for the nets are all set to 1. The details of the quadratic programming can be found in placement works like [**38**]. After this, the blocks are sorted according to their x positions and placed in a column-by-column manner from left to right: once the height of a column exceeds the square root of the total area of all the blocks, a new column is created and blocks are added from bottom to top again. If a dummy block representing $Head_V$ is put below all the blocks, this column by column placement actually creates a V-LP tree rooted at $Head_V$ with the columns as paths in the tree. Then the V-Tree-Weaving algorithm is used to construct the initial CAG from the tree.

Once the initial CAG is obtained, we improve it iteratively through a randomized greedy improvement heuristic by applying randomized *move*s. In each iteration, the move is to randomly pick up two blocks and swap them. Currently we assume that the orientations are fixed and rotations are not taken into consideration. The intuition behind these randomized moves is that the interconnects are mostly affected by the relative

positions of blocks when the whitespace is limited. Good relative relations can be found by following good moves using a proper cost function. Then the CAG is packed via the iterative packing heuristic to improve the area without changing the current adjacency relations dramatically. After that, the cost function is evaluated and the cost is compared to the one before this iteration. If there is any improvement, the current move is accepted and the current CAG will be the starting point of the next iteration; if there is no improvement, the current move is rejected and the CAG before this iteration is restored as the starting point. In practice, this heuristic can be terminated by adding a limit on the rejecting rate or on the running time.

3.4.2. Experimental Setup

We implemented the CAG algorithms in the C++ language. The Parquet tool [**25**] version 4.0 is used as a comparison. Both programs are compiled with GCC 3.4.3 and run on a Linux machine with 933MHz Pentium III processor and 512M memory.

Three GSRC benchmarks with only hard blocks are used: n100, n200, and n300. The statistics of these benchmarks are shown in Table 3.1.

Table 3.1. Statistics of the benchmarks for CAG.

name	# blocks	# terminals	# nets	total area
n100	100	334	885	179.5K
n200	200	564	1585	175.7K
n300	300	569	1893	273.2K

The Parquet tool is running in the free-outline mode with the sequence pairs representation and starting with a quadratic programming solution. The other representation

in Parquet, which is B^*-tree, generates similar results according to the work [**36**] and thus is not compared here.

The cost function used is the weighted sum of the area and the HPWL. We use a weight ratio of 1 : 1. We implement the HPWL calculation in the same manner as Parquet for fairness.

3.4.3. Experimental Results

We ran our floorplanner for 10 times with a pre-set time limit for each benchmark. For comparison, we ran Parquet twice for each benchmark: the first one is to let it run 10 times with the same time limit as ours; the second one is to let it run 10 times with a longer time limit such that the results are with the same quality as the ones in the work [**36**]. The results are reported in Table 3.2. Results from our floorplanner are listed in the rows with the method "CAG" and those from Parquet are listed in the rows with the methods "Parquet A" and "Parquet B" respectively. For each benchmark and each group of the 10 runs, the "area" column shows the minimal/maximal area; the "HPWL" column shows the minimal/maximal HPWL; the "time" column shows the average running time in seconds; the "#moves" column shows the average number of the randomized moves for our approach and that of the perturbations for Parquet.

From the results it can be seen that the CAG approach can find better floorplans in much less time compared to the up-to-date simulated annealing floorplanner Parquet. Although the CAG approach still relies on randomized moves, the far less number of moves required to reach a optimized floorplan shows that the iterative packing heuristic enables

Table 3.2. Comparing CAG and Parquet.

name	method	area	HPWL	time(s)	#moves
n100	CAG	195.9K/204.5K	302.1K/312.8K	15.30	31.3K
	Parquet A	196.8K/206.0K	320.2K/342.9K	14.90	96.5K
	Parquet B	195.0K/203.5K	313.6K/338.5K	29.80	175.8K
n200	CAG	197.0K/205.4K	540.9K/553.3K	30.86	26.0K
	Parquet A	207.4K/218.2K	613.8K/647.9K	29.40	54.0K
	Parquet B	197.4K/202.5K	578.9K/624.5K	149.2	256.6K
n300	CAG	304.4K/315.6K	649.0K/665.8K	61.61	33.8K
	Parquet A	335.2K/351.0K	750.6K/800.0K	58.89	62.5K
	Parquet B	306.9K/314.6K	709.2K/757.3K	290.6	325.7K

efficient explorations of the solution space when the interconnects are taken into consideration. In addition, the iterative packing heuristic does not add significant overhead when the interconnect estimation is part of the cost function. These two factors add up to the reduction in running times with better floorplans.

An optimized floorplan for n100 along with the CAG is shown in Figure 3.14 to conclude this section.

3.5. Summary

In this chapter, we proposed to use CAG as the adjacency representation of the floorplanning problems. Algorithms were presented to construct floorplans as well as improve CAGs. A randomized greedy iterative heuristic was used to utilize the characteristics in the CAG approach and the experimental results were promising for both the quality and the running time compared to existing simulated annealing floorplanners. More research work into CAG are expected to be done in the future to fully utilize its advantages as an adjacency graph.

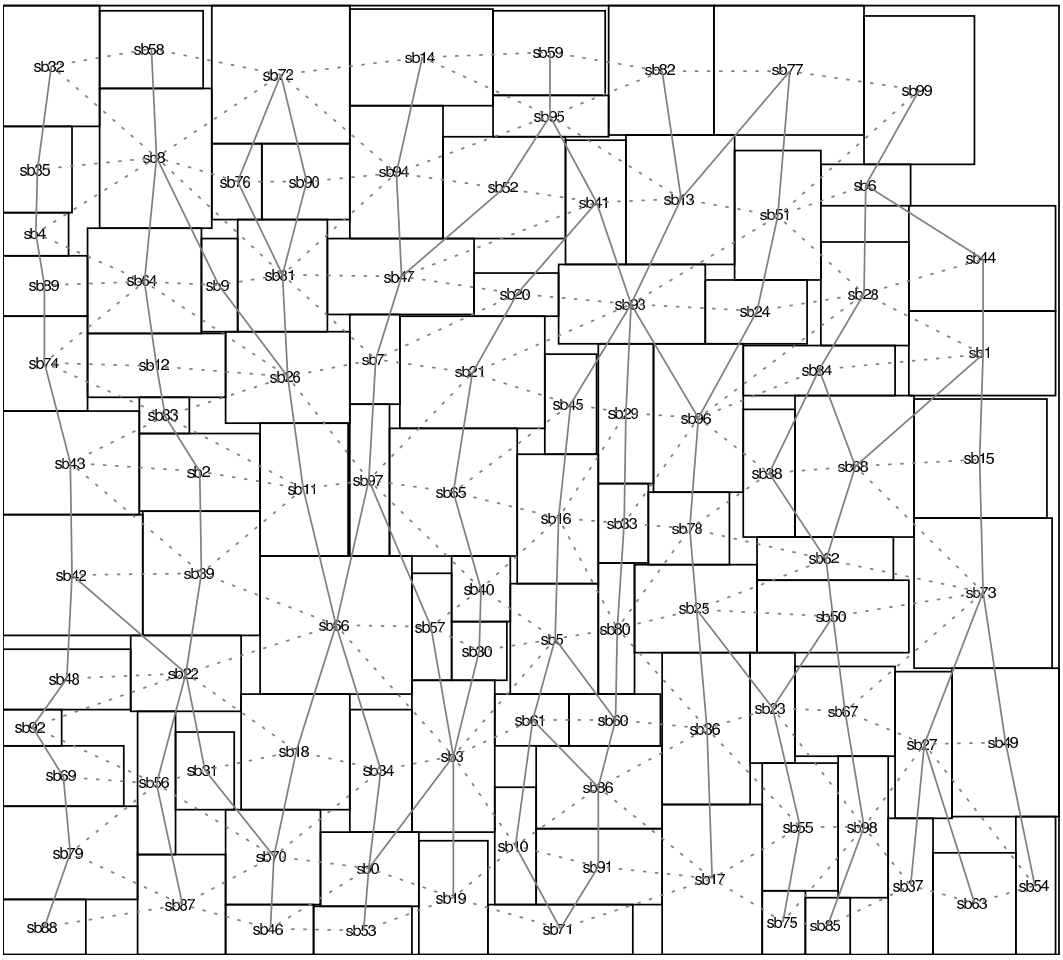

Figure 3.14. A floorplan of n100 with the CAG.

CHAPTER 4

Linear Constraint Graph for Floorplan Optimization with Soft Blocks

Many floorplan representations have been proposed for optimizations with simulated annealing (SA). Among the methods that convert from a representation to its physical floorplan, packing is widely used because of its simplicity and the ability to generate area optimal floorplans when the shapes of the modules are fixed. However, if there are soft blocks among the modules, or the objective of the floorplan optimization is beyond area minimization, or there are additional placement constraints, packing will not result in optimal floorplans. It is favorable in those cases to explore the solution space of all the non-overlapping floorplans under a given topology in a mathematical programming formulation using a constraint graph to avoid block overlapping [**39, 40, 41, 42, 43**]. It is critical to solve the optimization problem for each topology generated in SA efficiently. From the aspect of the topologies, the corresponding constraint graphs should have small sizes, i.e., to have as few vertices and edges as possible.

Constraint graphs have been studied since the early years of the floorplan research. Polar graphs that describe the geometric relations between rooms and the maximal line segments for rectangular dissections were introduced by Ohtsuki et al. [**44**] and were reviewed by Otten [**45**]. Although there is no method to perturb the polar graphs in SA, they can be explored by deriving them from mosaic floorplan representations, e.g. Twin

Binary Sequences (TBS) [**46**]. For the general floorplan problem of the block placement, Transitive Closure Graph (TCG) [**47**] and Adjacent Constraint Graph (ACG) [**10**] were proposed as constraint graph based floorplan representations. As TCG keeps all the transitive edges in the graph, the number of edges in a TCG is $\Theta(n^2)$ for n blocks. In ACG, the sizes of the constraint graphs are reduced intentionally by forbidding over-specifications, transitive edges, and the "crosses", which are special geometric relations that may result in $\Theta(n^2)$ edges in a constraint graph. However, there is no proof showing that the number of edges in an ACG would be $O(n)$. On the other hand, previous works [**39, 40, 41, 42, 43**] employed simpler approaches that generate the constraint graphs using the pair-wise geometric relations from either the sequence-pairs [**48**] or the physical floorplan. The number of edges in the constraint graphs generated by these approaches might be $\Theta(n^2)$ in the worse case and is $O(n \log n)$ [**49**] in the average case if the transitive edges are removed.

Our contribution in this chapter is to present a class of constraint graphs named *Linear Constraint Graph*s (LCG). LCG is the first general floorplan representation based on constraint graphs where the numbers of the vertices and the edges are linear to the number of the blocks, which improves upon the previous super-linear size bound in TCG and ACG. Intuitively, LCGs can be viewed as a combination of the polar graphs and ACGs: the "crosses" are avoided in one dimension as suggested by ACG and are avoided in the other by inserting "bars", which are similar to the maximal line segments in the polar graphs. For n blocks, an LCG contains at most $2n+3$ vertices and $6n+2$ edges. We construct an LCG by constructing its horizontal constraint graph first as a *Horizontal Adjacent Graph* (HAG). Generally speaking, HAG captures the horizontal relation between the blocks that

are close to each other and is planar. The vertical constraint graph is generated as the *Vertical cOmpanion Graph* (VOG) of the horizontal one, which ensures that every pair of blocks that are not separated horizontally are separated vertically. The operations we designed to perturb the LCGs has direct geometric meaning – such advantage is shared by the constraint-graph-based representations TCG and ACG. We focus on the application of LCG to the floorplan optimization problems with soft blocks. We emphasize that LCG is preferable when the constraint graphs are essential for the floorplan problem while LCG is an efficient representation that can be applied to solve general floorplan problems.

The rest of this chapter is organized as follows. In Section 4.1, definitions are reviewed. In Section 4.2, we show the motivation of our work. In Section 4.3, we define LCGs and show its properties. In Section 4.4, we present the operations to perturb LCGs in SA and introduce the framework for floorplan optimization with soft blocks. Experimental results are reported in Section 4.5. Section 4.6 concludes the chapter.

4.1. Preliminaries

For ease of presentation, we use $V(G)$ and $E(G)$ to denote the set of the vertices and the set of the edges of any graph G respectively.

A constraint graph describes the geometric relations between the blocks in a floorplan. A horizontal directed edge $e = (u, w)$ represents that u is to the left of w while a vertical one represents that u is below w. Four terminal vertices s_h,t_h,s_v,and t_v represent the four boundaries of the floorplan. Any pair of blocks are either separated horizontally or vertically in the constraint graph to avoid overlapping. Let B be the set of the rectangular

blocks representing the modules in a circuit. We define constraint graphs as follows formally.

Definition 4.1 (Constraint Graph)**.** *The tuple $G = (C_h, C_v)$ is a constraint graph of the blocks B iff:*

CG-1 *There are vertices s_h,t_h,s_v,t_v such that $B \cup \{s_h, t_h\} \subseteq V(C_h)$ and $B \cup \{s_v, t_v\} \subseteq V(C_v)$.*

CG-2 *Both C_h and C_v are directed acyclic graphs (DAG).*

CG-3 *$\forall u \in B$, there exists a directed path from s_h (and s_v) to t_h (and t_v) in C_h (and in C_v) containing u.*

CG-4 *For any pair of blocks, there is a directed path from one of them to the other either in C_h or in C_v.*

Let $V(G) = V(C_h) \cup V(C_v)$ and $E(G) = E(C_h) \cup E(C_v)$. The graph C_h and C_v are the horizontal and the vertical constraint graph of G respectively.

Recall that B is the set of the blocks. For every $b \in B$, let the width and the height of the block b be $w(b)$ and $h(b)$ respectively. For the dummy vertices besides the terminal vertices and the blocks in $V(G)$, we set $w(b) = h(b) = 0$. The floorplan F with the coordinates of the blocks given as the labelings x and y is *represented* by a constraint graph $G = (C_h, C_v)$ iff we can assign coordinates to the terminal vertices and the dummy vertices such that,

$$\begin{aligned} x(i) + w(i) \leq x(j), \quad & \forall (i,j) \in E(C_h), \\ y(i) + h(i) \leq y(j), \quad & \forall (i,j) \in E(C_v). \end{aligned} \tag{4.1}$$

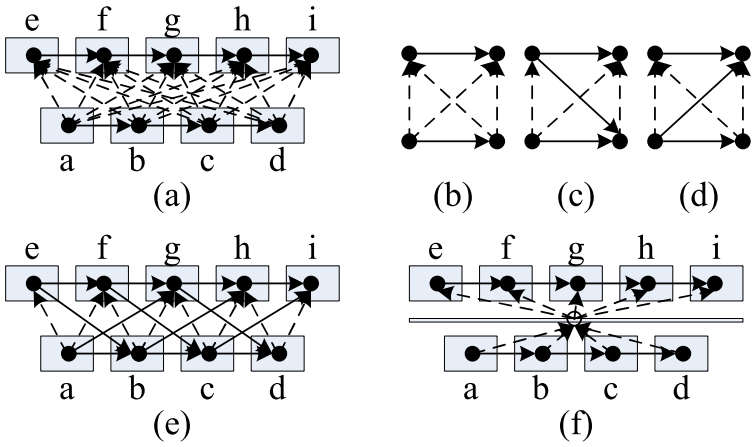

Figure 4.1. (a) A floorplan and the constraint graph with $\Theta(n^2)$ edges. (b) A vertical cross as defined by the work of ACG. (c) (d) Two alternatives for the vertical cross. (e) ACG reduces the number of the edges. (f) Alternatively, a bar can be inserted to reduce the number of the edges.

We define that a set of the constraint graphs is *complete* for a set of blocks, iff for every non-overlapping floorplan of the blocks, there is a constraint graph representing it and belonging to the set.

4.2. Motivation

To design a constraint graph based general floorplan representation is not easy. Adjacent Constraint Graph (ACG) [**10**] is the work most relevant to ours. ACGs are the constraint graphs satisfying the following three conditions: first, no over-specification, i.e. each pair of blocks are separated either horizontally or vertically but not both; second, no transitive edge, since the corresponding geometric relation is implied; third, no "crosses". The cross is a basic structure in the constraint graph that would result in $\Theta(n^2)$ edges in a constraint graph for n blocks satisfying the first two conditions in the worse case. An example of the worse case situation is shown in Figure 4.1 (a). A vertical cross is shown

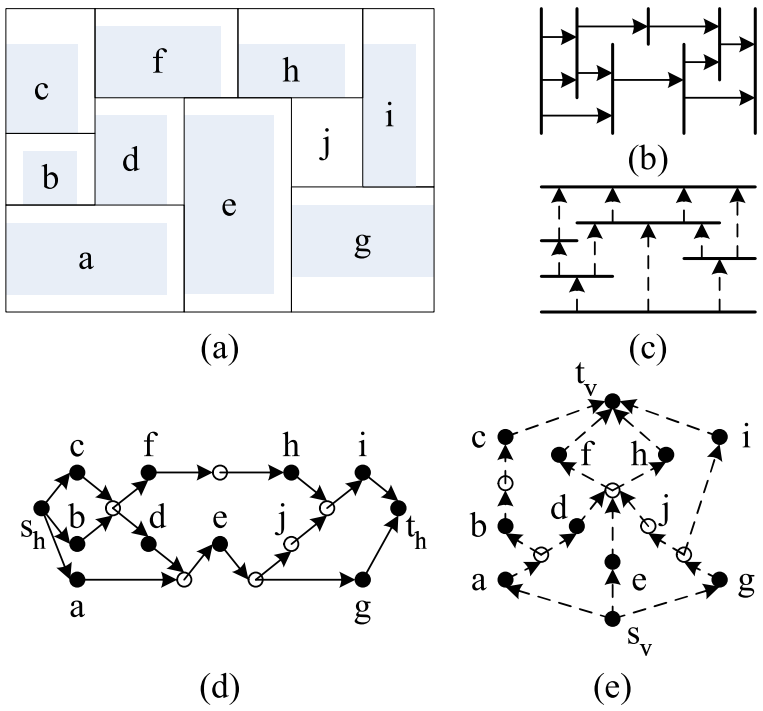

Figure 4.2. (a) Convert a block placement to a mosaic floorplan by inserting the dummy room j. (b)(c) The polar graphs. (e)(f) The constraint graph.

in Figure 4.1 (b). ACG avoids such crosses by using alternatives. It is proved that the non-overlapping blocks in a cross must satisfy one of the two groups of the constraints as shown in Figure 4.1 (c) and (d). The number of the edges is reduced in ACG as shown in Figure 4.1 (e). However, although ACGs try to keep the edges between adjacent blocks, there are still edges between the blocks that are not close, e.g. the edge (e, b) in Figure 4.1 (e). More complicated floorplan topology would require more sophisticated ACG. There is no proof showing that the number of the edges in an ACG would be $O(n)$.

Avoiding crosses by using alternatives is not the only way to overcome the worse case. Intuitively, for the example shown in Figure 4.1 (a), we may insert a horizontal "bar" to decompose the vertical relations in vertical crosses and a dummy vertex is inserted

to the constraint graph correspondingly as shown in Figure 4.1 (f). Similarly, we may insert vertical bars to decompose the horizontal relations. A more systematical approach is illustrated in Figure 4.2 that first converts the block placement to a mosaic floorplan by inserting the dummy room j as shown in Figure 4.2 (a). The polar graphs [**44, 45**] are then constructed in Figure 4.2 (b) and (c) where the vertices represent the maximal line segments, i.e. the bars, and the edges represent the rooms. Finally, the horizontal and the vertical constraint graphs are derived from the polar graphs by inserting a vertex representing each room on the corresponding edge. However, it is difficult to design operations that perturb the polar graphs because the dummy rooms and the line segments.

Our *Linear Constraint Graph*s (LCG) adopt an approach that combines the cross avoidance method from ACG and the method of inserting bars similar to that of the polar graphs. We avoid horizontal crosses by using the alternatives as in ACGs but avoid the vertical crosses by inserting horizontal bars. The horizontal constraint graph will contain no dummy vertices and be planar to make the perturbations as easy as possible. It is presented formally as the *Horizontal Adjacent Graph* (HAG). The horizontal bars and the vertical constraint graph will be implied by a given HAG, formally as the *Vertical cOmpanion Graph* (VOG). The LCG of the floorplan as shown in Figure 4.2 (a) is shown in Figure 4.3. This example is used throughout the chapter to illustrate the intuition behind LCG.

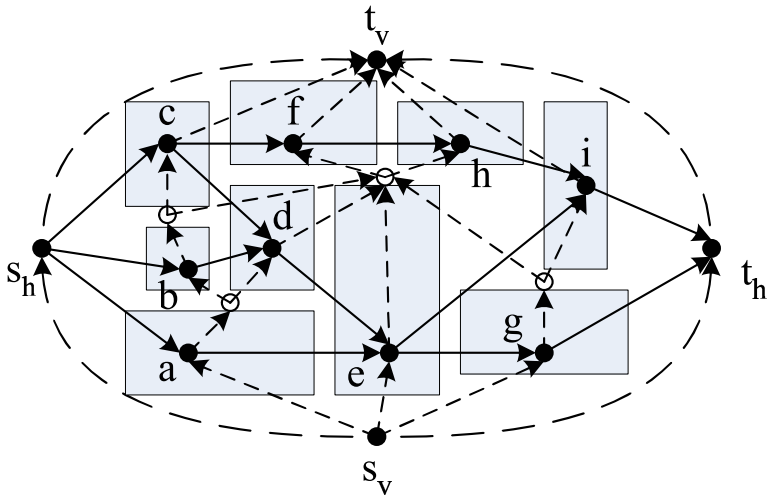

Figure 4.3. The LCG of the floorplan in Figure 4.2 (a).

4.3. Linear Constraint Graph

4.3.1. Horizontal Adjacent Graph

The edges in a horizontal constraint graph can be ordered by the vertical relations among the vertices. Define a DAG to be *ordered* if the outgoing edges and the incoming edges of every vertex are ordered. For an ordered DAG G and any vertex $u \in V(G)$, let $R(u)$ and $L(u)$ be the sequences of the ordered outgoing edges and the ordered incoming edges of u respectively, representing the right and the left neighbors of u sorted vertically from bottom to top. Let $R^+(u)$ and $R^-(u)$ (respectively $L^+(u)$ and $L^-(u)$) be the first and the last element in $R(u)$ (respectively $L(u)$). Let the other vertices besides u of the edges $R^+(u)$, $R^-(u)$, $L^+(u)$, and $L^-(u)$ be $r^+(u)$, $r^-(u)$, $l^+(u)$, and $l^-(u)$, respectively. For a constraint graph, the intuition of using the ordered DAG is to incorporate the vertical order into the horizontal constraint graph, i.e., those two sequences $R(u)$ and

$L(u)$ represent the right and the left neighbors of u sorted vertically from bottom to top, respectively.

The ordered DAGs are stored in a data structure as follows, which can be treated as "half" of the ACG data structure [**10**]. Each vertex u maintains two doubly linked list for $R(u)$ and $L(u)$. Each edge (u, w) stores two vertex pointers pointing to u and w, and four edge pointers pointing to the previous edges and the next edges in $R(u)$ and $L(w)$. Such data structure has the advantage that if a vertex u is given, then $R^+(u)$, $R^-(u)$, $L^+(u)$, and $L^-(u)$ can be accessed in constant time, $R(u)$ and $L(u)$ can be traversed in linear time, and edges can be inserted to or removed from $R(u)$ and $L(u)$ in constant time.

To achieve planarity in the horizontal constraint graph, we define the *above* and the *below* path as follows, which are essentially the boundaries of the faces that can be traversed from any edge. By such definition, we will derive the conditions for an ordered DAG to be planar, in which case the above paths would be the top boundaries of the faces while the below paths would be the bottom boundaries.

Definition 4.2 (Above and Below Paths)**.** *For any* $(u, w) \in E(G)$ *and the symbol* $\alpha \in \{+, -\}$*, the path* $P^\alpha(u, w) = (u_1, u_2, \ldots, u_{k+1})$ *is the* above *path for* $\alpha = +$ *or the* below *path for* $\alpha = -$ *iff:*

PAB-1 $\forall 1 \leq i \leq k$, $(u_i, u_{i+1}) \in E(G)$.

PAB-2 $\exists 1 \leq j \leq k$, $u_j = u$ *and* $u_{j+1} = w$.

PAB-3 $\forall 1 < i \leq k$, $u_{i+1} = r^\alpha(u_i)$; $\forall 1 \leq i < k$, $u_i = l^\alpha(u_{i+1})$.

PAB-4 $L(u_1) = \emptyset$ *or* $u_2 \neq r^\alpha(u_1)$; $R(u_{k+1}) = \emptyset$ *or* $u_k \neq l^\alpha(u_{k+1})$.

For $e = (u, w)$*, the above and the below path can be written alternatively as* $P^+(e)$ *and* $P^-(e)$*, respectively.*

The definition of the above and the below paths can be extended to the vertices that there is at least one edge incident on according to Lemma 4.1. For any vertex u such that $L(u) \neq \emptyset$ or $R(u) \neq \emptyset$, if $L(u) \neq \emptyset$, then define $P^+(u) = P^+(L^+(u))$ and $P^-(u) = P^-(L^-(u))$, otherwise define $P^+(u) = P^+(R^+(u))$ and $P^-(u) = P^-(R^-(u))$.

Lemma 4.1. *For vertex u, if $L(u) \neq \emptyset$ and $R(u) \neq \emptyset$, then $P^+(L^+(u)) = P^+(R^+(u))$ and $P^-(L^-(u)) = P^-(R^-(u))$.*

Proof. According to PAB-3, $L^+(u)$ is on the path $P^+(R^+(u))$ and $R^+(u)$ is on the path $P^+(L^+(u))$. Therefore $P^+(R^+(u))$ and $P^+(L^+(u))$ are the same path. Similarly, $P^-(R^-(u))$ and $P^-(L^-(u))$ are the same path. □

We define the *Horizontal Adjacent Graph* (HAG) as follows. The conditions HAG-1 and HAG-2 are from the requirement of the constraint graph. The condition HAG-3 ensures that there is no transitive edge in a HAG. The condition HAG-4 is essential for the HAGs to be planar.

Definition 4.3 (HAG)**.** *An ordered DAG C_h is a horizontal adjacent graph of the blocks B iff:*

HAG-1 *There are vertices s_h and t_h such that $B \cup \{s_h, t_h\} = V(C_h)$.*

HAG-2 *$\forall u \in B$, there exists a directed path from s_h to t_h in C_h containing u.*

HAG-3 *$\forall e \in E(C_h)$, both $P^+(e)$ and $P^-(e)$ contain at least two edges.*

HAG-4 *$\forall u \in V(C_h)$, let $R(u) = (e_1, \ldots, e_d)$. Then $\forall 1 < k \leq d$, there exists $u' \in V(C_h)$ with $L(u') = (e'_1, \ldots, e'_{d'})$ and $1 < j \leq d'$ such that $P^+(e_k) = P^+(e'_j)$ and $P^-(e_{k-1}) = P^-(e'_{j-1})$.*

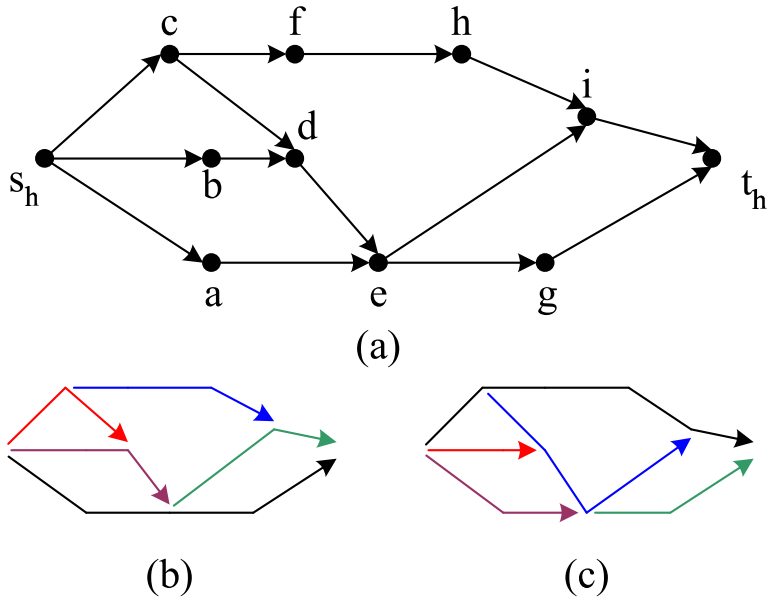

Figure 4.4. (a) A HAG. (b) The above paths. (c) The below paths.

The example of a HAG and all the above and the below paths are shown in Figure 4.4. The faces and their boundaries can be identified to understand the intuition behind the above and the below paths. The path pairs referred in the condition HAG-4 are highlighted with the same color. The exceptions are the top above path and bottom below path. We have the following lemma regarding them.

Lemma 4.2. *$P^+(s_h) = P^+(t_h)$ and $P^-(s_h) = P^-(t_h)$.*

Proof. Because of the symmetry, it is suffice to prove that $P^+(s_h) = P^+(t_h)$.

Let $L(t_h) = (e'_1, \ldots, e'_{d'})$. Let u be the first vertex of $P^+(t_h)$ with $R(u) = (e_1, \ldots, e_d)$. Let e_k be the first edge of $P^+(t_h)$ for some $1 \leq k \leq d$. We prove $k = 1$ by contradiction. If $k \neq 1$, then according to HAG-4, there exists $1 < j \leq d'$ such that $P^+(e_k) = P^+(e'_j)$. Therefore, $P^+(e'_1) = P^+(t_h) = P^+(e_k) = P^+(e'_j)$ for $j \neq 1$, which is impossible. Thus we

must have $k = 1$ and then $L(u) = \emptyset$ according to PAB-4. According to HAG-2, this is only possible for $u = s_h$. Therefore $P^+(s_h) = P^+(e_1) = P^+(t_h)$. □

Moreover, there is no transitive edge according to Lemma 4.3.

Lemma 4.3. *In a HAG C_h, if $(u, w) \in E(C_h)$, then there is no path from u to w in C_h other than the path containing only the edge (u, w).*

Proof. We prove the lemma by contradiction, i.e., to assume there exists such a path p.

Since p contains at least one vertex other than u and w, there are at least two outgoing edges of u and at least two incoming edges of w. According to HAG-3, both $P^+(u, w)$ and $P^= - (u, w)$ should contain at least two edges. Therefore, either $r^+(u) = w$ and $l^-(w) = u$, or $r^-(u) = w$ and $l^+(w) = u$. However, HAG-4 cannot be satisfied in either case. Thus the lemma is proved. □

4.3.2. The Top-Insert Lemma

To further explore the properties of a HAG, e.g. the size and the planarity, we present the Top-Insert Lemma that can be used to reason upon HAGs through mathematical induction.

It is straightforward that if $|B| = 1$, then there is only one HAG – assuming $B = \{b\}$, we can write the HAG as C_h^1 with $V(C_h^1) = \{s_h, b, t_h\}$ and $E(C_h^1) = \{(s_h, b), (b, t_h)\}$. Given a HAG and a new block, we build a new HAG using the InsertTop subroutine as shown in Figure 4.5. Generally speaking, this subroutine constructs a new HAG by inserting a new vertex to the top of an existing one. On line 1, the new block b is inserted to the vertex

set. If $E(C_h)$ contains the edge (a, c), it is removed on line 2 to satisfy the condition HAG-3. Two new edges are inserted on line 3 and line 4.

Subroutine InsertTop
Inputs C_h: a HAG. b: a new block. a, c: two vertices on $P^-(s_h)$ in C_h. **Output** Updated C_h.
1 Insert b to $V(C_h)$. 2 **If** $r^-(a) = c$: remove (a, c) from $E(C_h)$. 3 Insert (a, b) to $E(C_h)$ such that $R^-(a) = (a, b)$. 4 Insert (b, c) to $E(C_h)$ such that $L^-(c) = (b, c)$.

Figure 4.5. The InsertTop subroutine.

The correctness and the time complexity of the subroutine are stated in the following lemma.

Lemma 4.4. *Assume C_h is a HAG of the blocks B, $b \notin B$, a and c are on the below path $P^-(s_h)$ in C_h, and c is after a on $P^-(s_h)$. Then InsertTop updates C_h into a HAG of the blocks $B \cup \{b\}$ in constant time.*

Proof. The subroutine consumes constant time because of the data structure that stores the HAG.

To prove the correctness, first of all, we must have $(a, c) \in E(C_h)$ iff $r^-(a) \neq c$ according to PAB-3 and PAB-4. If $(a, c) \in E(C_h)$, then the paths $P^+(a, c)$ and $P^-(a, c)$ are changed by inserting the new vertex b while other above and below paths are not affected. If $(a, c) \notin E(C_h)$, then a new above path (a, b, c) is introduced. Let the sub-path in $P^-(s_h)$ from a to c be P'. Then P' becomes a new below path from a to c and the below path $P^-(s_h)$ is changed by replacing P' with (a, b, c). In either case, it is straightforward to verify the updated graph C_h is a HAG of the blocks $B \cup \{b\}$. □

For the ease of presentation, define T to be the function that represents the output of the InsertTop subroutine, i.e., let $T(C_h, b, a, c)$ be the updated HAG obtained by InsertTop(C_h, b, a, c). The following Top-Insert Lemma shows that every HAG can be built from C_h^1 using the InsertTop subroutine with proper parameters.

Lemma 4.5 (The Top-Insert Lemma)**.** *Let C_h be a HAG of the blocks B where $|B| > 1$. Then there exist vertices a, b, and c such that, first, a, b, and c are three consecutive vertices on $P^-(s_h)$ in C_h; second, $C_h = T(C'_h, b, a, c)$ for some HAG C'_h of the blocks $B - \{b\}$.*

Proof. We claim there is a vertex b on $P^-(s_h)$ such that b has exactly one incoming edge and one outgoing edge. Let $P^-(s_h) = (u_0, \ldots, u_{k+1})$ where $k \geq 1$, $u_0 = s_h$, and $u_{k+1} = t_h$. We prove the claim by contradiction, i.e., to assume that, $\forall 1 \leq j \leq k$, u_j has at least two incoming edges or at least two outgoing edges. Based on the assumption, we first prove that, $\forall 1 \leq j \leq k$, u_j has exact one incoming edge and at least two outgoing edges by induction on j. For $j = 1$, the vertex u_1 has no incoming edge other than (u_0, u_1) – otherwise $P^+(u_0, u_1)$ has only one edge, which violates HAG-3. Thus u_1 should have at least two outgoing edges. Suppose u_{j-1} has at least two outgoing edges for $j > 1$. Similar to the argument for the case $j = 1$, we have that u_j has only one incoming edge. Thus u_j should have at least two outgoing edges. Therefore, we proved that, $\forall 1 \leq j \leq k$, u_j has exact one incoming edge and at least two outgoing edges. On the other hand, because of the symmetry, we can also prove that, $\forall 1 \leq j \leq k$, u_j has exact one outgoing edge and at least two incoming edges. A contradiction is reached. Thus the claim holds and such vertex b exists.

Let (a, b) and (b, c) be the incoming and the outgoing edges respectively. Then a, b, and c are three consecutive vertices on $P^-(s_h)$ in C_h. We construct C'_h by first removing the vertex b and the edges (a, b) and (b, c) from C_h. If there is no path from a to c in C'_h, then we modify C'_h by inserting an edge (a, c) such that $r^-(a) = c$ and $l^-(c) = a$. It is straightforward to verify that, first, C'_h is a HAG of the blocks $B - \{b\}$; second, a and c are both on $P^-(s_h)$ in C'_h and a is before c; third, $C_h = T(G'_h, b, a, c)$. Therefore, we proved the lemma. □

According to the Top-Insert Lemma, we have,

Corollary 4.1. *Suppose C_h is a HAG of the blocks B, then $|V(C_h)| = |B| + 2$ and $|E(C_h)| \leq 2|B|$.*

Proof. It is implied by the Top-Insert Lemma. □

4.3.3. Vertical Companion Graph

Once we have a HAG as the horizontal constraint graph, we construct a vertical constraint graph accordingly that separates every pair of the blocks that is not separated horizontally by using dummy vertices, i.e. the horizontal bars. We define the *Vertical cOmpanion Graph* (VOG) recursively as follows, which will be the vertical constraint graph. First of all, for the HAG C^1_h of one block b, let the vertical companion graph be C^1_v satisfying that,

$$V(C^1_v) = \{s_h, t_h, b, s_v, t_v\},$$

$$E(C^1_v) = \{(s_v, s_h), (s_v, b), (s_v, t_h), (s_h, t_v), (b, t_v), (t_h, t_v)\}.$$

For the HAG C_h of the blocks B where $|B| > 1$, suppose $C_h = T(C'_h, b, a, c)$ for some HAG C'_h of the blocks $B - \{b\}$ according to the Top-Insert Lemma. Assume the VOG of C'_h to be C'_v. We construct the VOG of C_h using the CoInsertTop subroutine as shown in Figure 4.6. The intuition is to insert new edges such that for every block that is not separated horizontally with b, there is a path in C_v from the vertex to b. Note that in the subroutine, we assume for every $u \in V(C'_h)$, there are two unique vertices u^+ and u^- in $V(C'_v)$ such that both (u^+, u) and (u, u^-) belong to $E(C'_v)$. The validity of this assumption will be established when we proved the correctness of the subroutine in Lemma 4.6. On line 1, the vertex b is inserted to the VOG. If (a, c) is an edge in C_h, we insert one new edge to C_v on line 3 and insert another new edge on line 5 or 7 depending on whether $r^+(a) = c$ or $l^+(c) = a$. Otherwise, a dummy vertex f is inserted on line 9 before two edges are inserted on line 10. The end points of the outgoing edges from some of the vertices on $P^-(s_h)$ are replaced in the loop on line 12. There is a path in C_v from a^+ to f (through $r^-(a)$) if $a = l^+(r^-(a))$. Otherwise we insert the edge (a^+, f) on line 14. Similarly, the edge (c^+, f) is inserted on line 15 iff there is no path in C_v from c^+ to f.

The CoInsertTop subroutine is consistent because of the following lemma.

Lemma 4.6. *Suppose C_v is the VOG of a HAG C_h. Then, first, $V(C_h) \subset V(C_v)$ and C_v is a DAG. Second, $\forall u \in V(C_h)$, there is exactly one incoming edge (u^+, u) and one outgoing edge (u, u^-) in C_v, and both u^+ and u^- do not belong to $V(C_h)$. Third, for every vertex u on $P^-(s_h)$, $u^- = t_v$, and for every vertex u on $P^+(s_h)$, $u^+ = s_v$. Fourth, if (a, c) is an edge in $P^-(s_h)$ in C_h, then at least one of $r^+(a) = c$ and $l^+(c) = a$ holds. If both of them hold, then $a^+ = c^+$.*

Subroutine CoInsertTop

Inputs

C_h,C_v:a HAG and the VOG of it.
b :a new block.
a, c :two consecutive vertices on $P^-(s_h)$ in C_h.

Output Updated C_v.

```
1  Insert b to V(C_v).
2  If r^-(a) = c:
3    Insert (b, t_v) to E(C_v).
4    If r^+(a) = c:
5      Insert (a^+, b) to E(C_v).
6    Else:// must have l^+(c) = a
7      Insert (c^+, b) to E(C_v).
8  Else:
9    Insert a new vertex f to V(C_v).
10   Insert (f, b) and (b, t_v) to E(C_v).
11   Let P' be the sub-path of P^-(s_h) from a to c.
12   For each vertex u on P' except a and c:
13     Replace (u, t_v) with (u, f) in E(C_v).
14   If a ≠ l^+(r^-(a)): insert (a^+, f) to E(C_v).
15   If c ≠ r^+(l^-(c)): insert (c^+, f) to E(C_v).
```

Figure 4.6. The CoInsertTop subroutine.

Proof. It is straightforward to prove the lemma by induction on $|V(C_h)|$ according to the definition of VOG. □

An example of the VOG of the HAG shown in Figure 4.4 (a) is shown in Figure 4.7. We have the following corollary concerning the size of a VOG according to the CoInsertTop subroutine.

Corollary 4.2. *Suppose C_v is a VOG of the HAG of the blocks B, then $|V(C_v)| \leq 2|B| + 3$ and $|E(C_v)| \leq 4|B| + 2$.*

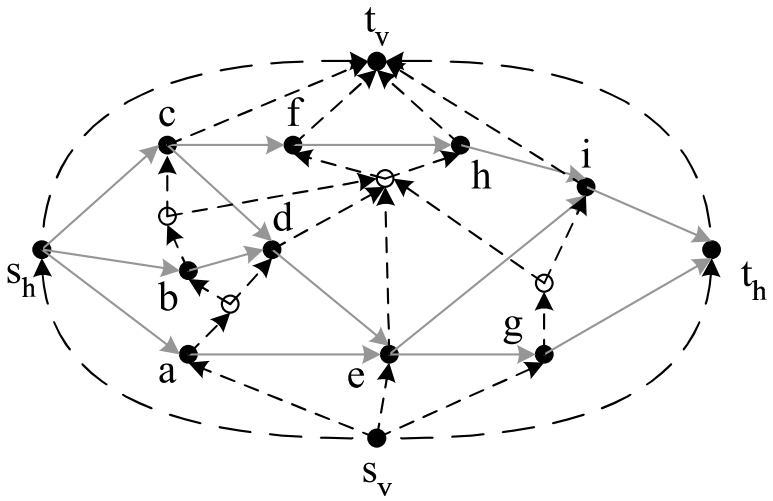

Figure 4.7. The VOG of the HAG shown in gray edges.

Proof. There are 5 vertices and 6 edges in C_v^1. Every time we insert a new vertex by the CoInsertTop subroutine, at most 1 dummy vertex and 4 edges are introduced. Thus the corollary holds. □

4.3.4. Linear Constraint Graph

Based on the definition of HAG and VOG, we define the *Linear Constraint Graph* (LCG) as follows.

Definition 4.4 (LCG)**.** *For the blocks B, the linear constraint graph G is a tuple (C_h, C_v), where C_h is a HAG of B and C_v is a VOG of C_h. Let $V(G) = V(C_h) \cup V(C_v)$ and $E(G) = E(C_h) \cup E(C_v)$.*

The following lemma states that an LCG is a constraint graph, which can be proved by showing inductively that the condition CG-4 holds.

Lemma 4.7. *An LCG is a constraint graph.*

Proof. We prove that CG-4 holds for any LCG $G = (C_h, C_v)$ for the blocks B by induction on $|B|$. If $|B| = 1$, obviously CG-4 holds since there is only one vertex in B.

Suppose CG-4 holds for $|B| = n-1$. For $|B| = n$, according to the Top-Insert lemma, assume $C_h = F(G'_h, b, a, c)$. Let G'_v be the VOG of G'_h and let C_v be obtained by the CoInsertTop subroutine. Let $P' = (u_0, \ldots, u_k)$ be the sub-path of $P^-(s_h)$ in G'_h from a to c, i.e. $u_0 = a$ and $u_k = c$. Let $w \in B - \{b\}$. According to Lemma 4.3, $\forall 0 \leq j < k$, there is no path from u_j to u_{j+1} in G'_h. According to Lemma 4.6, $\forall 0 \leq j \leq k$, $u_j^- = t_v$. Thus, $\forall 0 \leq j \leq k$, there is no path from u_j to w in G'_v. Therefore, if we consider the paths between w and u_j, $\forall 0 \leq j \leq k$, in G'_h and G'_v, there are 4 possible cases as follows. First, there is a path from w to a in G'_h or $w = a$. Then there is a path from w to b in C_h. Second, there is a path from c to w in G'_h or $w = c$. Then there is a path from b to w in C_h. Third, $w = u_j$ for some j satisfying $0 < j < k$. Then there is a path from w to b through b^+ in C_v. Fourth, there is a path from w to u_j in G'_v for some j satisfying $0 \leq j \leq k$. Recall (u_j^+, u_j) is the only incoming edge of u_j in G'_v. Then the path must contain u_j^+. Thus there is a path from w to b through u_j^+ in C_v. So we proved CG-4 holds for $w \in B - \{b\}$ and $u = b$. It is straightforward to verify that CG-4 holds for $w, u \in B - \{b\}$ in C_h and C_v according to the induction hypnoses. Therefore, CG-4 holds when $|B| = n$.

It is straightforward that CG-1, CG-2, and CG-3 hold for an LCG. Since we have shown that CG-4 hold for an LCG, we proved that an LCG is a constraint graph. □

If a non-overlapping floorplan is given, the FPToLCG algorithm shown in Figure 4.8 constructs the LCG representing the floorplan. The algorithm builds the LCG by inserting the blocks according to the ascending order of their y-coordinates. From the LCG

constructed on line 3 and 4, the subroutine CoInsertTop and InsertTop are called to insert new blocks on line 9 and 11. The labelings x and y are extended on line 3, 4, and 10. The below path $P^-(s_h)$ in C_h is maintained as P throughout the algorithm on line 5 and 12. It is implemented as an AVL tree or a Red-Black tree that stores the vertices on the path according to their x labeling.

Algorithm FPToLCG
Inputs A non-overlapping floorplan F.
Output The LCG $G = (C_h, C_v)$.

1 Sort the blocks B into $(b_1, b_2, \ldots, b_{|B|})$ according to the y-coordinates $y(b), \forall b \in B$.
2 $M \leftarrow \max_{b \in B}\{|x(b)|, |y(b)|, |x(b)+w(b)|, |y(b)+h(b)|\}$.
3 $V(C_h) \leftarrow \{s_h, b_1, t_h\}$, $E(C_h) \leftarrow \{(s_h, b_1), (b_1, t_h)\}$.
$(x(s_h), x(t_h), w(s_h), w(t_h)) \leftarrow (-M, M, 0, 0)$.
4 $V(C_v) \leftarrow V(C_h) \cup \{s_v, t_v\}$, $E(C_v) \leftarrow \bigcup_{u \in V(C_h)}\{(s_v, u), (u, t_v)\}$.
$(y(s_h), y(t_h), y(s_v), y(t_v)) \leftarrow (0, 0, -M, M)$.
5 The path $P \leftarrow (s_h, b_1, t_h)$.
6 **For** $i = 2$ to $|B|$:
7 $a \leftarrow \text{argmax}_{\{u: u \in P, x(u)+w(u) \le x(b_i)\}} x(u)$.
8 $c \leftarrow \text{argmin}_{\{u: u \in P, x(b_i)+w(b_i) \le x(u)\}} x(u)$.
9 $C_v \leftarrow$CoInsertTop(C_h, C_v, b_i, a, c).
10 **If** $r^-(a) \neq c$: $y(b_i^+) \leftarrow y(b_i)$.
11 $C_h \leftarrow$InsertTop(C_h, b_i, a, c).
12 Remove all the vertices between a and c in P.
Insert b_i to P between a and c.

Figure 4.8. The FPToLCG algorithm.

The invariant of the loop on line 6 is stated in the following lemma.

Lemma 4.8. *For $1 \le i \le |B|$, let $F^{(i)}$ be the floorplan consisting of the blocks $b_1, b_2, \ldots, b_i$ in F. For $1 \le i < |B|$, let $C_h^{(i)}$, $C_v^{(i)}$, and $P^{(i)}$ be the graphs C_v and C_h and the path P when entering the loop on line 6. Let $C_h^{(|B|)}$, $C_v^{(|B|)}$, and $P^{(|B|)}$ be the graphs C_v and C_h and the path P when the algorithm terminates. Then, $\forall 1 \le i \le |B|$, the following*

three claims hold: first, the graph $G^{(i)} = (C_h^{(i)}, C_v^{(i)})$ *is an LCG; second,* $P^{(i)} = P^-(s_h)$ *in* $C_h^{(i)}$*; third, Equation (4.1) holds for the constraint graph* $G^{(i)}$*, the floorplan* $F^{(i)}$*, and the labelings* x *and* y*.*

Proof. We then proof the lemma by induction on i. When $i = 1$, it is straightforward that all the three claims hold.

Suppose the three claims hold for $i = j-1$ where $1 < j \leq |B|$. Then $G^{(j)}$ and $P^{(j)}$ are obtained when leaving the loop with $i = j - 1$. Since $P^{(j-1)} = P^-(s_h)$ in $G_h^{(j-1)}$ and M is sufficient large as computed on line 2, we must have $s_h \in \{u : u \in P^{(j-1)}, x(u) + W_u \leq x(b_j)\}$ and $t_h \in \{u : u \in P^{(j-1)}, x(b_j) + W_{b_j} \leq x(u)\}$. Thus a and c are well defined. Since $G^{(j-1)}$ is an LCG, $G^{(j)}$ is an LCG according to Lemma 4.4 and 4.6. It can be verified that $P^{(j)} = P^-(s_h)$ in $G_h^{(j)}$. According to Lemma 4.7, $G^{(j)}$ is a constraint graph. Because the labelings x and y are never changed in the algorithm once assigned, to prove that the third claim holds, it is sufficient to show that Equation 4.1 holds for the new edges introduced by the InsertTop and CoInsertTop subroutines. According to line 7 and 8 where a and c are computed, Equation 4.1 holds for the two edges introduced by the InsertTop subroutine. It is straightforward to verify that Equation 4.1 holds for the edges introduced on line 3 and 10 of the CoInsertTop subroutine because of the order established on line 1, the assignment on line 10, and the sufficiently large M. According to line 7 and 8, for any vertex u between a and c in $P^{(j-1)}$, we must have $u \in B$ and $(F_{b_j}^x, F_{b_j}^x + W_{b_j}) \cap (F_u^x, F_u^x + W_u) \neq \emptyset$. According to line 1, $F_{b_j}^y \geq F_u^y$. Thus $F_{b_j}^y \geq F_u^y + W_u$ since F is non-overlapping. Then it can be verified that Equation 4.1 holds for the replaced edges in the loop on line 12 of the CoInsertTop subroutine. For the edges introduced on line 15 and 15, Equation 4.1 holds since $y(f) = F_{b_j}^y \geq F_a^y \geq y(a^+)$ and

$y(f) = F^y_{b_j} \geq F^y_c \geq y(c^+)$. Therefore, we have proved that all the three claims hold for $i = j$ and then the lemma holds. □

The correctness and the complexity of the FPToLCG algorithm are stated in the following lemma.

Lemma 4.9. *Assume F is a non-overlapping floorplan of the blocks B. The FP-ToLCG algorithm will terminate and generate an LCG $G = (C_h, C_v)$ that represents F. The time complexity is $O(|B| \log |B|)$ and and space complexity is $O(|B|)$.*

Proof. Obviously the algorithm will terminate. The algorithm generates an LCG representing the floorplan according to Lemma 4.8.

According to Lemma 4.8, and Corollary 4.1 and 4.2, it requires $O(|B|)$ storage to store C_h,C_v,and P. Therefore, the space complexity is $O(|B|)$.

The sorting on line 1 consumes $O(|B| \log |B|)$ time. The searching on line 7 and 8 consumes at most $O(\log |B|)$ time per iteration and then $O(|B| \log |B|)$ time totally. Every block is inserted to and removed from P for at most 1 time. Thus line 12 consumes $O(|B| \log |B|)$ time totally. According to Lemma 4.4, the InsertTop subroutine consumes constant time per iteration and then $O(|B|)$ time totally. Note that the number of the edges replaced on line 12 of the CoInsertTop subroutine is exactly the same as the number of the vertices removed from P. Thus the CoInsertTop subroutine consumes $O(|B|)$ time totally. All the other part of the algorithm consumes $O(|B|)$ time. Therefore, the time complexity is $O(|B| \log |B|)$. □

In summary, we have the following theorem as the major result of this chapter according to Lemma 4.7 and 4.9, and Corollary 4.1 and 4.2.

Theorem 4.1. *Let the set of all the LCGs of a set of blocks B be $\mathcal{L}_B$. Then $\mathcal{L}_B$ is complete for B and $\forall G \in \mathcal{L}_B$, $V(G) \leq 2|B| + 3$ and $E(G) \leq 6|B| + 2$.*

Proof. The theorem is implied by Lemma 4.7 and 4.9, and Corollary 4.1 and 4.2. □

4.4. LCG Floorplan Optimization

We perform floorplan optimization using LCGs by SA. Two most important issues are addressed in the following two sub-sections: one is to design operations that perturb the representation and the other is to evaluate the representation through a cost function. There are two most important issues for a floorplan representation to be used in such an iterative improvement heuristic. The first one is to design operations that perturb the representation such that every instance of the representation can be explored stochastically. The second one is to evaluate the representation through a cost function. These two issues are addressed in the following two sections in sequel.

4.4.1. Perturbations of LCG

Recall that an LCG consists of a HAG and a VOG. We design three operations that perturb LCGs. The first one is with the name exchange and exchanges two blocks in both the HAG and the VOG. The next two operations are designed to first perturb HAGs and then update the VOGs accordingly since the VOG can be derived from a HAG. In the operation with the name insertH, an edge is inserted between two vertices in the HAG which changes the vertical relation between them into a horizontal one. In the operation with the name removeH, an edge in the HAG is removed such that the horizontal relation

between the end points of the edge is changed to a vertical one. We only present the changes in HAGs for these two operations while omit the changes in VOGs for simplicity.

Suppose the LCG is $G = (C_h, C_v)$. The insertH operation is illustrated in Figure 4.9. Recall that for $u \in V(C_h)$, (u^+, u) and (u, u^-) are the only edges incident on u in C_v. Two vertices $a \in B$ and $b \in B$ are selected such that $a^+ = b^-$. Such vertices exist when C_h is not a single path from s_h to t_h. We can insert either (b, a) or (a, b) to C_h. Because of the symmetry, assume that the edge (b, a) should be inserted without loss of generality. Let $c = l^+(a)$ and $d = r^-(b)$. The following lemma holds for $P^+(a)$ and $P^-(b)$ before inserting (b, a).

Lemma 4.10. *If for $a, b \in V(C_h)$ we have $a^+ = b^-$, then the end points of $P^+(a)$ and $P^-(b)$ are the same.*

Proof. According to the CoInsertTop subroutine, it is straightforward to prove the lemma by induction on the number of the vertices in C_h. □

If c is the first vertex of $P^+(a)$ and $P^-(b)$, there is a path from c to b in C_h. Thus the edge (c, a) should be removed such that HAG-3 would not be violated after inserting (b, a). Similarly, if d is the last vertex of $P^+(a)$ and $P^-(b)$, the edge (b, d) should be removed. Finally, the edge (b, a) is inserted to C_h such that $r^-(b) = a$ and $l^+(a) = b$.

The removeH operation is illustrated in Figure 4.10. An edge $(b, a) \in E(C_h)$ with $a \in B$ and $b \in B$ is selected such that either $r^-(b) = a$ and $l^+(a) = b$, or $r^+(b) = a$ and $l^-(a) = b$. Such edge exists when there is an edge in C_h whose end points both belong to B. Because of the symmetry, assume that $r^-(b) = a$ and $l^+(a) = b$ without loss of generality. If a has more than one incoming edges in C_h, an optional new edge

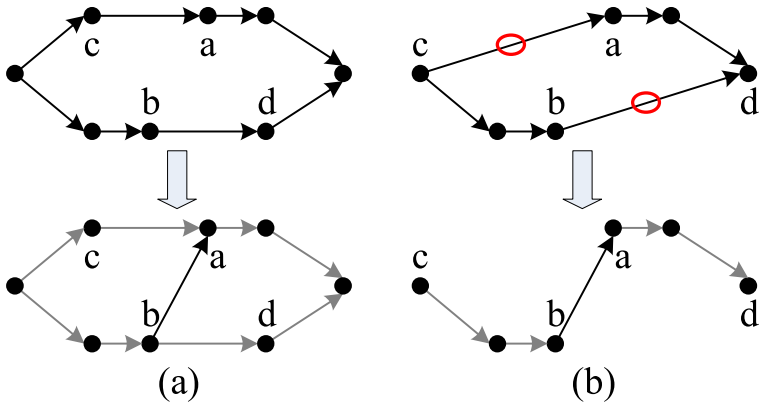

Figure 4.9. Insert a horizontal edge (b, a): (a) when c and d are not the end points of $P^+(a)$; (b) when c and d are the end points of $P^+(a)$, (c, a) and (b, d) should be removed.

(c, a) can be inserted to C_h where c is a vertex on $P^-(b)$ between the first vertex and b. Otherwise, if a has only one incoming edge, i.e. (b, a), a new edge (c, a) must be inserted where c is either the first vertex of $P^-(b)$ or a vertex on $P^-(b)$ between the first vertex and b. Similarly, if b has more than one outgoing edges, an optional new edge (b, d) can be inserted where d is a vertex on $P^+(a)$ between a and the last vertex; if b has only one outgoing edge, a new edge (b, d) must be inserted where d is either the last vertex of $P^+(a)$ or a vertex on $P^+(a)$ between a and the last vertex. Finally, the edge (b, a) is removed from C_h.

The correctness and the time complexity of the insertH and the removeH operations are stated in the following lemma.

Lemma 4.11. *Both the* insertH *and the* removeH *operations change an LCG into another one. The time complexity is $O(n)$ for n blocks.*

Proof. It can be verified that the conditions from HAG-1 to HAG-4 are not violated by the insertH and the removeH operations. Recall that $R^+(u)$, $R^-(u)$, $L^+(u)$, and $L^-(u)$

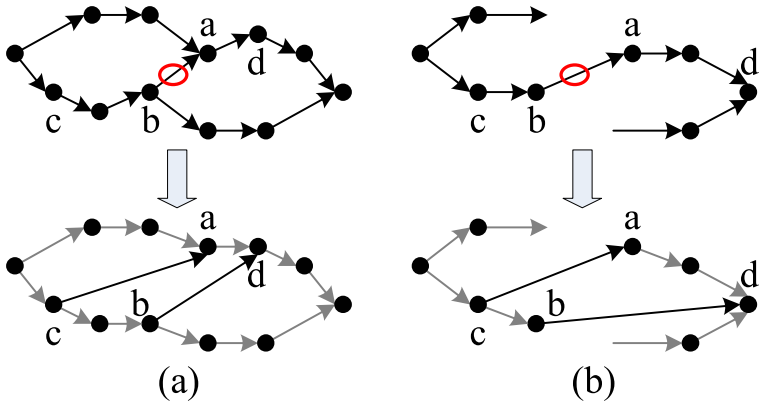

Figure 4.10. Remove the horizontal edge (b, a): (a) when a and b have at least 2 incoming and outgoing edges respectively, (c, a) and (b, d) are optional; (b) when a and b have only 1 incoming and outgoing edges respectively, (c, a) and (b, d) must be inserted.

can be accessed in constant time and edges can be inserted to or removed from $R(u)$ and $L(u)$ in constant time. Therefore, both operations take $O(n)$ time to finish. □

The following theorem states that the three operations exchange, insertH, and removeH are complete and efficient to perturb LCGs.

Theorem 4.2. *Applying the three operations* exchange, insertH, *and* removeH *can convert any LCG to any other LCG. For n blocks, it takes at most $3n$ operations for such conversions.*

Proof. An insertH operation can be reverted by a removeH operation. Moreover, a dummy vertex in the VOG of any LCG can be removed by the insertH operation as shown in Figure 4.9 (b) without introducing new dummy vertices. Thus we would obtain an LCG with no dummy vertex in the VOG from any LCG of n blocks by applying the insertH operation at most $n - 1$ times. The HAGs in such LCGs consist of a single path from s_h to t_h and the conversions among them can be done by at most n exchange operation.

Then we can obtain any LCG from such LCGs via reverting the insertH operations by the removeH operations. Therefore, applying the three operations can convert any LCG to any other LCG and it takes at most $3n$ operations for such conversions. □

4.4.2. Floorplan Optimization with Soft Blocks

When there are soft blocks, to evaluate a floorplan the block shapes should be determined first. Generally speaking, a mathematical programming problem can be formulated by using the constraint graph as the constraints to optimize the decision variables which are the block shapes. We follow the approach of Lin et al. [**41**] to formulate the problem and to solve the problem by Lagrangian relaxation. This approach was previously proposed by Young et al. [**39**] and was improved in the work [**41**]. We only introduce the relevant part in this section while the details and the reviews of the previous works should be found in the works [**39**] [**41**].

Recall that B is the set of the blocks. Suppose that each block $b \in B$ has an area $A(b)$. Let the decision variables be the block widths $w(b)$ within the range $[L(b), U(b)]$, $\forall b \in B$, and the labellings x and y. For an LCG $G = (C_h, C_v)$, the following problem $\mathrm{P}_{\mathrm{peri}}$ is formulated to optimize the perimeter of the floorplan bounding box.

Problem 4.1 ($\mathrm{P}_{\mathrm{peri}}$)**.**

$$
\begin{aligned}
\textit{Minimize} \quad & (x(t_h) - x(s_h)) + (y(t_v) - y(s_v)) \\
\textit{s.t.} \quad & x(i) + w(i) \leq x(j),\ \forall (i,j) \in E(C_h) \wedge i \neq s_h, \\
& x(s_h) \leq x(j),\ \forall (s_h, j) \in E(C_h),
\end{aligned}
$$

$$y(i) + \frac{A(i)}{w(i)} \leq y(j),\ \forall (i,j) \in E(C_v) \wedge i \in B,$$
$$y(i) \leq y(j),\ \forall (i,j) \in E(C_v) \wedge i \notin B,$$
$$L(i) \leq w(i) \leq U(i),\ \forall i \in B.$$

It was shown in [**41**] that $\mathrm{P}_{\mathrm{peri}}$ is a convex programming formulation under the variable transformation $\mathrm{logw}(b) = \log w(b)$, $\forall b \in B$, and thus can be solved by Lagrangian relaxation as follows. Let λ and μ be the vectors of the Lagrangian multipliers $\lambda_{i,j}$, $\forall (i,j) \in E(C_h)$, and $\mu_{i,j}$, $\forall (i,j) \in E(C_v)$, respectively. Define

$$\lambda_i \triangleq \sum_{(i,j) \in E(C_h)} \lambda_{i,j}, \qquad \forall i \in B,$$
$$\mu_i \triangleq \sum_{(i,j) \in E(C_v)} \mu_{i,j}, \qquad \forall i \in B.$$

Let w be the vector of the block widths and F_{peri} be defined as

$$F_{\mathrm{peri}}(\lambda, \mu, w) \triangleq \sum_{i \in B} \Big(\lambda_i w(i) + \mu_i \frac{A(i)}{w(i)} \Big).$$

The Lagrangian subproblem is to compute Q_{peri} defined as

$$Q_{\mathrm{peri}}(\lambda, \mu) \triangleq \min_{L(b) \leq w(b) \leq U(b), \forall b \in B} F_{\mathrm{peri}}(\lambda, \mu, w).$$

The $\mathrm{P}_{\mathrm{peri}}$ problem can be solved by solving a simplification of the Lagrangian dual problem of it, which is the following $\mathrm{LD}_{\mathrm{peri}}$ problem.

Problem 4.2 ($\mathrm{LD}_{\mathrm{peri}}$)**.**

$$
\begin{aligned}
\textit{Maximize} \quad & Q_{peri}(\lambda, \mu) \\
\textit{s.t.} \quad & \sum_{(i,j)\in E(C_h)} \lambda_{i,j} = \sum_{(j,k)\in E(C_h)} \lambda_{j,k}, \\
& \forall j \in V(C_h) \wedge j \neq s_h \wedge j \neq t_h, \\
& \sum_{(i,j)\in E(C_v)} \mu_{i,j} = \sum_{(j,k)\in E(C_v)} \mu_{j,k}, \\
& \forall j \in V(C_v) \wedge j \neq s_v \wedge j \neq t_v, \\
& \sum_{(i,t_h)\in E(C_h)} \lambda_{i,t_h} = \sum_{(s_h,k)\in E(C_h)} \lambda_{s_h,k} = 1, \\
& \sum_{(i,t_v)\in E(C_v)} \mu_{i,t_v} = \sum_{(s_v,k)\in E(C_v)} \mu_{s_v,k} = 1, \\
& \lambda \geq 0, \mu \geq 0.
\end{aligned}
$$

As $Q_{\mathrm{peri}}(\lambda, \mu)$ is in general not differentiable, Young et al. [**39**] proposed to apply subgradient optimizations to solve the $\mathrm{LD}_{\mathrm{peri}}$ problem. On the other hand, Lin et al. [**41**] proposed a trust-region method to optimize $Q_{\mathrm{peri}}(\lambda, \mu)$. In each iteration of this method, a min-cost network-flow problem on the constraint graph is formulated based on the current pair of the multipliers λ and μ and a step size Δ and is solved to generate a new pair of the multipliers. Depending on the improvement of $Q_{\mathrm{peri}}(\lambda, \mu)$ in comparison to the expected one based on the first order approximation, either the new pair of the multipliers would be rejected and the step size would be decreased, or the new pair of the multipliers would be accepted and the step size would be increased or remain the same. We use the approach in the work [**41**] because of its efficiency.

Table 4.1. Results of area optimization for LCG.

		SP+TR			LCG+TR		
name	n	ds(%)	t(s)	$\|E\|$	ds(%)	t(s)	$\|E\|$
apte	9	**0.04**	34	40	**0.04**	21	34
xerox	10	**0.08**	43	47	**0.08**	41	38
hp	11	**0.09**	41	54	0.13	26	41
ami33	33	0.28	383	347	**0.24**	179	125
ami49	49	**0.24**	694	679	0.27	319	182

4.5. Experimental Results

We obtain the code of the floorplanner in the work [**41**], which was based on the code of the floorplanner in the work [**39**] and used CS2 version 4.3 [**50**] as the min-cost network flow solver. We replace the sequence-pair representation in the code with our LCG representation which is implemented in C++. Both the code of the work [**41**] and our code are compiled by GCC version 3.4 and run on a Linux workstation with two 927MHz Pentium III processors and 512MB memory. The same setting of simulated annealing is used in both code.

We follow Lin et al. [**41**] to setup the experiments. There are 5 benchmarks derived from the MCNC benchmark suite. The aspect ratio of each block is between 0.5 and 2. Area optimization is performed with the cost function being the perimeter of the floorplan bounding box. Wire length optimization is performed with the cost function being the summation of the perimeter and the average half-perimeter wire length of all the nets.

For each of the benchmarks and each optimization, we run each program for 5 times and compare the best results. The results from area optimization and wire length optimization are reported in Table 4.1 and 4.2 respectively. For each benchmark, the name and the number of the blocks are shown in the columns “name” and “n” respectively. The

Table 4.2. Results of wire length optimization for LCG.

	SP+TR			LCG+TR		
name	ds(%)	wl(mm)	t(s)	ds(%)	wl(mm)	t(s)
apte	0.09	125.29	35	**0.08**	**125.28**	26
xerox	0.21	**145.16**	46	**0.15**	152.69	36
hp	0.26	43.42	37	**0.22**	**42.60**	27
ami33	0.50	57.89	349	**0.49**	**52.46**	236
ami49	1.17	290.89	615	**0.64**	**272.23**	442

results from our approach are shown in the columns "LCG+TR". The results from the approach by Lin et al. [**41**] are shown in the columns "SP+TR". For both the area and the wire length optimizations, the deadspace in percentage and the running time in seconds are reported in the columns "ds(%)" and "t(s)" respectively. We observe that more than 95% of the runtime is spent to solve the $\mathrm{LD}_{\mathrm{peri}}$ problem by the trust-region method in both approaches. The average numbers of edges of the constraint graphs generated in SA are reported in the columns "$|E|$" for area optimization. These numbers are similar for wire length optimization and thus are omitted. The wire lengths in millimeter are reported in the columns "wl(mm)" for wire length optimization.

It can be seen from the tables that although the results of the approach by Lin et al. [**41**] are already almost optimal, our approach can improve the qualities for 1 benchmark for area optimization and 4 benchmarks for wire length optimization in much less time while the qualities for other cases are similar. It is clear that there are always less edges in the constraint graphs in our approach and the gap between the number of edges in our approach and that in the approach by Lin et al. [**41**] increases as the size of the benchmark increases. Note in the works [**39, 41**], when the constraint graphs were constructed from the sequence-pairs, the transitive edges were removed. This implies that in

a similar approach to maintain TCGs without transitive edges, the average numbers of edges would be similar to those of the approach by Lin et al. [**41**], which are more than those of our approach.

4.6. Summary

In this chapter, we proposed the *Linear Constraint Graph*s (LCG) as a general floorplan representation based on constraint graphs. For n blocks, we showed that each LCG has at most $2n + 3$ vertices and at most $6n + 2$ edges. We proved that LCGs can represent any non-overlapping floorplans. We designed operations that have direct geometric meaning to perturb LCGs in simulated annealing and proved such perturbations are sufficient to explore all the LCGs stochastically. The advantages of LCGs is confirmed by the experimental results.

CHAPTER 5

Gate Sizing by Lagrangian Relaxation Revisited

The transistor sizing, gate sizing, and wire sizing problems [**51, 52, 53, 54, 55, 56, 57**] are important problems in VLSI design because they allow to explore the trade-offs between the performance and the cost of the system. Since all these problems share the same structure that the timing constraints are formulated as a system of difference inequalities involving the delays of each individual components and the arrival times, we call them collectively as the sizing problems.

Most research works on the sizing problem use a convex delay model for individual components. It had been shown in the work TILOS by Fishburn et al. [**51**] that a few problem formulations concerning the total size and the clock period are convex programming problems under such delay model. Convex programming problems have the advantage that a local optimum is a global one and they have been studied for decades (see [**58, 59**] for references). In TILOS, the Elmore delay model [**60**] was used for transistor delays and a heuristic that sizes the transistors iteratively according to the sensitivities of the critical path delay to the transistor sizes was proposed to find an optimum. The Elmore delays are special cases of posynomials, which are a class of convex functions under the logarithm variable transformation. Sapatnekar et al. [**53**] applied the algorithms that solve general convex programming problems to solve the sizing problem for the delays as the posynomials of the sizes. Kasamsetty et al. [**61**] proposed to use the generalized posynomials, which are also convex under the logarithm variable transformation, to approximate the

delays more accurately than the Elmore delay model and solved the sizing problem with this model in the optimization framework of the work [**53**]. However, the experimental results in the works [**53**] and [**61**] showed that the general algorithms were not efficient for the sizing problems with even less than 1000 sizable components.

The special structure in the sizing problem that the timing constraints are formulated as a system of difference inequalities had been exploited by Chen et al. [**55**] to design an algorithm that solves the gate and wire sizing problem by Lagrangian relaxation. The structure allows to simplify the Lagrangian dual problem using the Karush-Kuhn-Tucker (KKT) conditions. The dual problem was solved by subgradient optimizations. The gate and wire delay model used in this work was the Elmore delay model and thus the Lagrangian subproblem was a convex optimization problem with simple constraints, which can be solved efficiently. Although the approach was efficient for sizing adders in the work [**55**], Tennakoon et al. [**57**] showed that for general circuits, it is very hard to choose proper initial solution and step sizes for subgradient optimizations to converge practically. Heuristics were developed in [**57**] to obtain a good initial solution and to speed up the convergence of subgradient optimizations. However, it is not clear whether the heuristics can be extended to handle more sophisticated and accurate convex delay models, e.g. the ones in the work [**61**].

The ever-increasing complexity in modern VLSI systems demands efficient and effective sizing algorithms to handle the sophisticated convex delay models and the tremendous number of sizable components. The difficulties in the previous works motivate us to design new algorithms for the sizing problem. We revisit the Lagrangian relaxation based

approach [**55**]. This particular approach is of our interests because the special structure of the sizing problem is exploited. Our contributions in this chapter include:

(1) We formulate the Generalized Convex Sizing (GCS) problem that unifies the sizing problems and applies to sequential circuits with clock skew optimization.

(2) We identify a class of the GCS problems called the proper GCS problems. We transform the simultaneous sizing and clock skew optimization problem into a proper GCS problem.

(3) We revisit the approach to formulate the Lagrangian dual problem by Lagrangian relaxation and to simplify the dual problem. Several misunderstandings are corrected and the approach is extended to handle general convex delay models.

(4) We prove that the objective function in the simplified dual problem of a proper GCS problem is differentiable and then design the DualFD algorithm to solve the proper GCS problems by the method of feasible directions and min-cost network flow.

(5) We derive the necessary and sufficient condition for the GCS problem to be feasible. This condition is applied to check the feasibility of the circuits in our experiments.

We focus on the continuous sizing problems in this chapter where the sizes of the components can be any values within certain continuous ranges. On the other hand, if the components are not continuous sizable, e.g. the gates must be chosen from a discrete gate library, a discrete sizing problem should be solved [**62, 63, 64**]. In such case, the optimal solution of the corresponding continuous problem is valuable. A discrete solution can be obtained by rounding the continuous solution as proposed by Chuang et al. [**62**],

or through dynamic programming guided by the continuous solution as proposed by Hu et al. [**64**].

The rest of this chapter is organized as follows. In Section 5.1, we examine the convex gate delay model and propose the Generalized Convex Sizing (GCS) problem. In Section 5.2, Lagrangian relaxation based approaches that formulate the dual problems are revisited. In Section 5.3, we present our DualFD algorithm that solves the proper GCS problems. Experimental results are reported in Section 5.4. Section 5.5 concludes the chapter.

5.1. Problem Formulation

5.1.1. The Generalized Convex Sizing Problem

Although gate sizing is commonly applied to the combinational part of a circuit, for a sequential circuit with flip-flops (FF) as the storage units, the clock period and the clock skews can be incorporated into the timing constraints as follows.

$$\begin{aligned} t_i + d_{i,j} &\le t_j, && \forall (i,j) \in E, \\ t_{\mathrm{pi}} = a_{\mathrm{pi}}, t_{\mathrm{po}} &= r_{\mathrm{po}}, && \forall \mathrm{pi} \in \mathrm{PI}, \mathrm{po} \in \mathrm{PO}, \\ t_{\mathrm{Q}_k} = s_k, t_{\mathrm{D}_k} &= s_k + T, && \forall k \in \mathrm{FF}. \end{aligned} \tag{5.1}$$

Here the topology of the combinational part of the circuit is represented by a directed acyclic graph (DAG) $G = (V, E)$. The vertices represent ports and the edges represent interconnects and timing arcs. The variable t_v is the arrival time at the vertex v. The function $d_{i,j}$ is the delay of the edge (i, j). It is either the delay of a wire or a timing arc

in a gate. The constants $a_{\rm pi}$ and $r_{\rm po}$ are the arrival times and the required arrival times of the primary input and output ports respectively. The clock period is T. For the FF k, s_k is the clock skew, and D_k and Q_k are the data input and output ports respectively. Note that extra vertices and edges can be introduced into the graph G to model practical timing issues, e.g., FF setup time, clock skew uncertainty, and safety margin. We assume that such issues, if necessary, are already incorporated.

Equation (5.1) can be expressed in a more consistent manner by extending G as follows. Two vertices I and O are added to V. The edges $(\mathrm{I}, \mathrm{pi})$ and $(\mathrm{po}, \mathrm{O})$ are added to E with $d_{\mathrm{I,pi}} = a_{\rm pi}$ and $d_{\mathrm{po},O} = T - r_{\rm po}$ for all pi $\in$ PI and po $\in$ PO. Then the edges $(\mathrm{I}, \mathrm{Q}_k)$ and $(\mathrm{D}_k, \mathrm{O})$ are added to E with $d_{\mathrm{I},\mathrm{Q}_k} = s_k$ and $d_{\mathrm{D}_k,\mathrm{O}} = -s_k$ for each $k \in$ FF. Finally the edge (O, I) is added to E with $d_{\mathrm{O,I}} = -T$. We will refer the extended graph as G for the ease of presentation when there is no ambiguity. The extended graph G of an example circuit is shown in Figure 5.1. This circuit has two primary input ports PI_1 and PI_2, one primary output port PO_1, and one FF with data input port D_1 and data output port Q_1. Let $\mathbf{t}$ be the vector of all the t_i. The following theorem relates Equation (5.1) to the inequalities derived from the the extended graph.

Theorem 5.1. *There exist arrival times* $\mathbf{t}$ *to satisfy Equation (5.1) if and only if there exist arrival times* t'_v, $\forall v \in V$, *in the extended graph* G *to satisfy the following inequalities,*

$$t'_i + d_{i,j} \leq t'_j, \forall (i,j) \in E.$$

Proof. Assume there exist arrival times $\mathbf{t}$ to satisfy Equation (5.1). Let $t'_v = t_v$ for every vertex v in the original graph. Let $t'_I = 0$ and $t'_O = T$. Then it can be verified that,

$$t'_i + d_{i,j} \leq t'_j, \forall (i,j) \in E.$$

On the other hand, assume there exist arrival times t'_v, $\forall v \in V$, to satisfy the inequalities derived from the extended graph. We assign the arrival times $\mathbf{t}$ to the vertices in the original graph as follows: for every pi $\in$ PI, let $t_{\text{pi}} = a_{\text{pi}}$; for every po $\in$ PO, let $t_{\text{po}} = r_{\text{po}}$; for every $k \in$ FF, let $t_{\text{Q}_k} = s_k$ and $t_{\text{D}_k} = s_k + T$; for the remaining vertices, say v, let $t_v = t'_v - t'_I$. Then it can be verified that Equation (5.1) holds.

Therefore, we have proved that there exist arrival times $\mathbf{t}$ to satisfy Equation (5.1) if and only if there exist t'_v, $\forall v \in V$, to satisfy the inequalities derived from the extended graph G. □

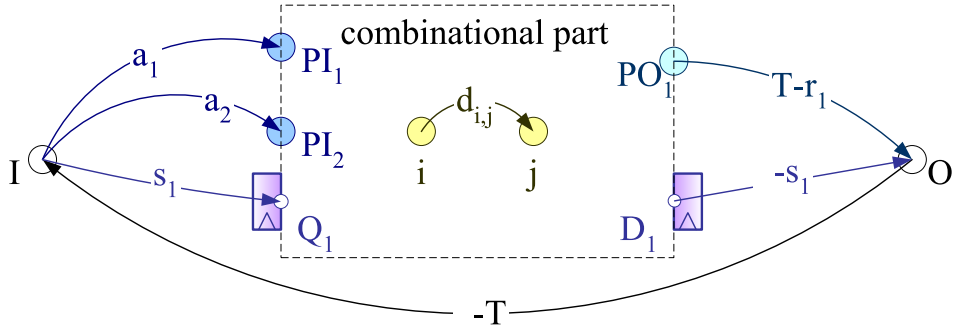

Figure 5.1. Timing of a sequential circuit.

Theorem 5.1 motivates us to formulate the *Generalized Convex Sizing* (GCS) problem with the constraints derived from the extended graph G.

Problem 5.1 (Generalized Convex Sizing)**.** *Let $G = (V, E)$ be a directed graph representing the structure of a system with the parameters $\mathbf{x} = (x_1, x_2, \ldots, x_n)^\top$ belonging to*

the set

$$\Omega \triangleq \{\mathbf{x} : l_k \leq x_k \leq u_k, \forall 1 \leq k \leq n\},$$

where l_k *and* u_k, $\forall 1 \leq k \leq n$, *are constants. The edge delays* $d_{i,j}$, $\forall (i,j) \in E$, *and the objective function* C *are twice differentiable convex functions for* $\mathbf{x} \in \Omega$. *Solve that,*

$$\begin{aligned} \textit{Minimize} \quad & C(\mathbf{x}) \\ \textit{s.t.} \quad & t_i + d_{i,j}(\mathbf{x}) - t_j \leq 0, \forall (i,j) \in E, \qquad (5.2) \\ & \mathbf{x} \in \Omega. \qquad (5.3) \end{aligned}$$

The decision variables in the GCS problem are $(\mathbf{x}, \mathbf{t})$. The GCS problem is a convex programming problem (see Chapter 4.2 [**58**]) since the objective function C and the left-hand-side of Equation (5.2) are all convex functions and the set Ω is a convex set. Define $(\mathbf{x}, \mathbf{t})$ to be feasible if Equation (5.2) and (5.3) are satisfied. Define $\mathbf{x}$ to be feasible if there exists a $\mathbf{t}$ such that $(\mathbf{x}, \mathbf{t})$ is feasible. Denote the set of all the feasible $\mathbf{x}$ by $\mathcal{X}$. It is straightforward that $\mathcal{X}$ is a convex set.

Although it is straightforward to determine if a particular $(\mathbf{x}, \mathbf{t})$ is feasible by verifying Equation (5.2) and (5.3), we rely on the the following theorem to determine if a particular $\mathbf{x}$ is feasible.

Theorem 5.2. *For a GCS problem,* $\mathbf{x} \in \Omega$ *is feasible if and only if there is no positive cycle in* G *with respect to the edge weights* $d_{i,j}(\mathbf{x})$.

Proof. If $\mathbf{x}$ is feasible, then there exists $\mathbf{t}$ such that Equation (5.2) holds. Thus for every cycle $\mathcal{C}$ in G, we have,

$$\sum_{(i,j)\in\mathcal{C}} d_{i,j}(\mathbf{x}) = \sum_{(i,j)\in\mathcal{C}} (t_i + d_{i,j}(\mathbf{x}) - t_j) \leq 0.$$

On the other hand, if there is no positive cycle in G with respect to the edge weights $d_{i,j}(\mathbf{x})$, we can apply the Bellman-Ford algorithm (see Chapter 24.1 [**29**]) to obtain $\mathbf{t}$ that satisfies Equation (5.2). Thus $\mathbf{x}$ is feasible. □

It is clear that the GCS problems are not restricted to the sizing problem where the delay functions are the posynomials of sizes and the convexity is established through geometric programming. One important group of the convex delay functions that are not posynomials of sizes is the affine functions of $\mathbf{x}$, i.e., in the form of $\mathbf{b}^\top\mathbf{x} + c$ for some vector $\mathbf{b}$ and scalar c. This allows to treat the clock skews and the clock period as decision variables in a GCS problem if necessary.

5.1.2. Proper GCS Problems

We are interested in a particular class of GCS problems, named *proper* GCS problems, since it can be efficiently solved by our DualFD algorithm as presented in the later sections. We will provide their definition and investigate their basic properties in this subsection and show how to formulate the sequential sizing problem that performs simultaneous sizing and clock skew optimization as a proper GCS problem in the next subsection.

The *proper* GCS problems are defined as follows.

Definition 5.1. *A GCS problem is proper if and only if the Hessian matrix of its objective function is positive definite at any* $\mathbf{x} \in \Omega$.

It should be pointed out that the objective function of a proper GCS problem must be a strictly convex function while the reverse is not true since the Hessian matrix of a twice differentiable strictly convex function is not always positive definite. For example, suppose $\mathbf{x} = (x_1, x_2)$ and $\Omega = \{(x_1, x_2) : -1 \leq x_1 \leq 1, -1 \leq x_2 \leq 1\}$. The objective function $C(\mathbf{x}) = x_1^4 + x_2^4$ is strictly convex but at $(x_1, x_2) = (0, 0)$, its Hessian matrix is a zero matrix and thus is not positive definite. On the other hand, for the strictly convex objective function $C(\mathbf{x}) = x_1^2 + x_2^2$, its Hessian matrix is always positive definite.

The definition of the proper GCS problems only depends on the property of the objective function but not that of the delay functions. Many practical sizing problems are the proper GCS problems as shown in the following theorem.

Theorem 5.3. *Assume that the objective function C of a GCS problem is a posynomial of the variables $e^{x_k}, \forall 1 \leq k \leq n$, i.e.*

$$C(\mathbf{x}) = \sum_{i=1}^{l} w_i \; e^{\mathbf{a}_i^\top \mathbf{x}},$$

where $w_i > 0$, $\forall 1 \leq i \leq l$. Then the GCS problem is proper if and only if $\mathrm{rank}(A) = n$, *where $A = (\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_l)$ is a matrix with n rows and l columns.*

Proof. Let H_C be the Hessian matrix of C. Since $C(\mathbf{x})$ is convex as required by the definition of the GCS problems, $H_C(\mathbf{x})$ is positive semidefinite for any $\mathbf{x} \in \Omega$. Assuming $\mathbf{a}_i = (a_i^1, a_i^2, \ldots, a_i^n)^\top$, we have that

$$\frac{\partial^2 C(\mathbf{x})}{\partial x_j \partial x_k} = \sum_{i=1}^{l} w_i a_i^k a_i^j e^{\mathbf{a}_i^\top \mathbf{x}}, \forall 1 \leq j \leq n, 1 \leq k \leq n.$$

Define

$$\Lambda(\mathbf{x}) \triangleq \text{diag}(w_i\ e^{\mathbf{a}_i^\top \mathbf{x}}, w_2\ e^{\mathbf{a}_2^\top \mathbf{x}}, \ldots, w_l\ e^{\mathbf{a}_l^\top \mathbf{x}}),$$

which is a diagonal matrix with positive diagonal elements. Then

$$H_C(\mathbf{x}) = A\Lambda(\mathbf{x})A^\top.$$

Thus, $\forall \mathbf{z} \in \mathbb{R}^n$,

$$\mathbf{z}^\top H_C(\mathbf{x})\mathbf{z} = \mathbf{z}^\top (A\Lambda(\mathbf{x})A^\top)\mathbf{z} = (A^\top \mathbf{z})^\top \Lambda(\mathbf{x})(A^\top \mathbf{z}). \tag{5.4}$$

If $H_C(\mathbf{x})$ is positive definite, then because of Equation (5.4) we can claim that $A^\top \mathbf{z} = 0$ has no solution other than $\mathbf{z} = 0$. So $\text{rank}(A) = n$.

On the other hand, if $\text{rank}(A) = n$, then $A^\top \mathbf{z} \neq 0$ for any $\mathbf{z} \neq \mathbf{0}$. Thus $\mathbf{z}^\top H_C(\mathbf{x})\mathbf{z} \neq 0$ for any $\mathbf{z} \neq \mathbf{0}$ according to Equation (5.4). So $H_C(\mathbf{x})$ is positive definite.

Therefore, we proved that the GCS problem is proper if and only if $\text{rank}(A) = n$. □

The following corollary applies to the GCS problems where the variables are the logarithms of the sizes and objective functions are the positive weighted summation of the sizes.

Corollary 5.1. *For a GCS problem, if*

$$C(\mathbf{x}) = \sum_{k=1}^{n} w_k\ e^{x_k},$$

where $w_k > 0$, $\forall 1 \leq k \leq n$, *then the problem is proper.*

Proof. It is implied by Theorem 5.3. □

5.1.3. Simultaneous Sizing and Clock Skew Optimization as a Proper GCS Problem

The clock skew optimization problem was studied by Fishburn [**19**] to optimize the circuit utilizing the delay variations from the clock source to FFs. The simultaneous sizing and clock skew optimization problem for general sequential circuits was studied by Chuang et al. [**62**]. They proposed an algorithm to solve the problem considering both the long path (setup) and the short path (hold) conditions by formulating a linear programming problem using the piece-wise-linear (PWL) approximations of the convex delays. However, if we consider a path p from Q_i to D_j for some FF i and j with the non-linear convex delay d_p, the short path condition,

$$-s_i - d_p + s_j + \text{hold-time} \leq 0,$$

is not convex because the left-hand-side of the inequality is not convex. Thus the PWL approximation may result in suboptimal solutions. We propose to consider the long path conditions only and assume that a post-processing algorithm, e.g. [**65**], will repair the violated short path conditions. Similar optimization flow was applied in [**54**] for acyclic pipelines.

Suppose that each clock skew s_k to be optimized belongs to a pre-defined range $[s_k^-, s_k^+]$, i.e.,

$$s_k^- \leq s_k \leq s_k^+,$$

where s_k^- and s_k^+ are constants. Assume that the objective function is the positive weighted summation of the sizes. It is straightforward that the simultaneous sizing and clock skew

optimization problem is a GCS problem where the clock skews to be optimized are among the variables $\mathbf{x}$ and the pre-defined ranges are part of the set Ω. However, as the clock skew variables do not appear in the objective function, the Hessian matrix of the objective function is not always positive definite and thus the problem is not proper. We overcome this difficulty by eliminating the clock skew variables and transforming the problem into a proper one as stated in Theorem 5.4. There are two advantages of this transformation: first, the number of the variables are reduced; second, since the transformed problem is a proper GCS problem, our DualFD algorithm presented later is applicable.

Theorem 5.4. *The simultaneous sizing and clock skew optimization problem can be transformed into a proper GCS problem without introducing variables representing the clock skews to be optimized.*

Proof. There is a valid skew assignment for the FF k if and only if there exists t_{I}, t_{O}, t_{Q_k}, t_{D_k}, and s_k satisfying

$$(t_{\mathrm{I}} + s_k \leq t_{\mathrm{Q}_k}) \wedge (t_{\mathrm{D}_k} - s_k \leq t_{\mathrm{O}}) \wedge (s_k^- \leq s_k \leq s_k^+).$$

That is,

$$[t_{\mathrm{D}_k} - t_{\mathrm{O}}, t_{\mathrm{Q}_k} - t_{\mathrm{I}}] \cap [s_k^-, s_k^+] \neq \emptyset,$$

which is equivalent to

$$(t_{\mathrm{D}_k} - s_k^+ \leq t_{\mathrm{O}}) \wedge (t_{\mathrm{I}} + s_k^- \leq t_{\mathrm{Q}_k}) \wedge (t_{\mathrm{D}_k} - t_{\mathrm{Q}_k} \leq t_{\mathrm{O}} - t_{\mathrm{I}}). \tag{5.5}$$

We transform the problem by modifying the graph G. An edge $(\mathrm{D}_k, \mathrm{Q}_k)$ is added with $d_{\mathrm{D}_k,\mathrm{Q}_k} = -T$. The delays are changed such that $d_{\mathrm{D}_k,\mathrm{O}} = -s_k^+$ and $d_{\mathrm{I},\mathrm{Q}_k} = s_k^-$.

Recall that $t_{\mathrm{O}} - t_{\mathrm{I}} \leq T$. According to Equation (5.5), any feasible $\mathbf{t}$ with the unmodified G will be feasible with the modified G. On the other hand, for any feasible $\mathbf{t}$ with the modified graph G, t_{O} can be increased to $t_{\mathrm{I}} + T$ without violating the constraints and changing the cost. Then Equation (5.5) holds. Thus the clock skew s_k can be chosen as any value in the non-empty set $[t_{\mathrm{D}_k} - t_{\mathrm{O}}, t_{\mathrm{Q}_k} - t_{\mathrm{I}}] \cap [s_k^-, s_k^+]$.

Therefore, the clock skew variables have been eliminated under such transformation and the problem after the transformation is proper. □

For the example circuit shown in Figure 5.1, we show the constraints before the transformation in Figure 5.2 and those after the transformation in Figure 5.3.

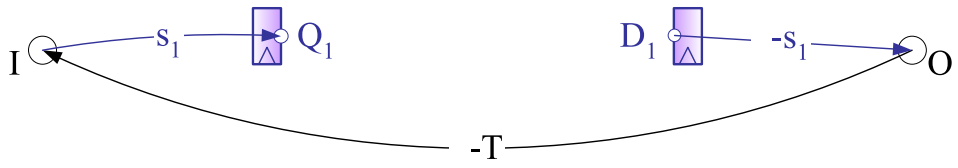

Figure 5.2. Timing of the FF 1 with its clock skew variable.

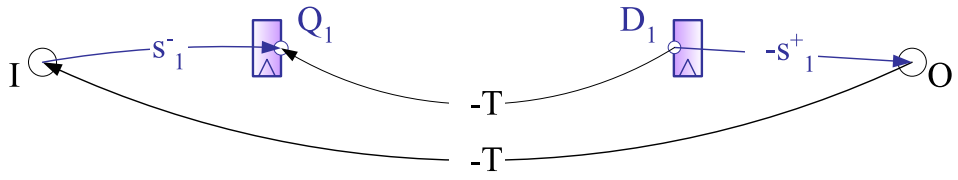

Figure 5.3. Timing of the FF 1 without its clock skew variable.

5.2. Solving GCS via Lagrangian Dual Problems

We follow Chen et al. [**55**] to formulate the Lagrangian dual problem of the GCS problem and to simplify the dual problem. Although the formulations are similar, we revisit the assumptions and correct the misunderstandings.

5.2.1. The Lagrangian Dual Problem

Let $f_{i,j}$ be the Lagrangian multipliers associated with each inequality in Equation (5.2). Let $\mathbf{f}$ be the vector for all the $f_{i,j}$. Let $L^*(\mathbf{x}, \mathbf{t}, \mathbf{f})$, $L(\mathbf{f})$, and $\mathcal{N}$ be the Lagrangian function, the Lagrangian dual function, and the set of the non-negative multipliers respectively, i.e.,

$$\begin{aligned} L^*(\mathbf{x}, \mathbf{t}, \mathbf{f}) &\triangleq C(\mathbf{x}) + \sum_{(i,j)\in E} f_{i,j}(t_i + d_{i,j}(\mathbf{x}) - t_j), \\ L(\mathbf{f}) &\triangleq \inf\{L^*(\mathbf{x}, \mathbf{t}, \mathbf{f}) : \mathbf{x} \in \Omega, \mathbf{t} \in \mathbb{R}^{|V|}\}, \\ \mathcal{N} &\triangleq \{\mathbf{f} : f_{i,j} \geq 0, \forall (i,j) \in E\}. \end{aligned}$$

The Lagrangian dual problem *D-GCS* is formulated as follows.

Problem 5.2 (D-GCS)**.**

$$\begin{aligned} \textit{Maximize} \quad & L(\mathbf{f}) \\ \textit{s.t.} \quad & \mathbf{f} \in \mathcal{N}. \end{aligned}$$

Recall that $\mathcal{X}$ is the set of all the feasible $\mathbf{x}$. The weak duality theorem (see Chapter 6.2 [**59**]) states that the duality gap is non-negative, i.e.,

$$\inf\{C(\mathbf{x}) : \mathbf{x} \in \mathcal{X}\} \geq \sup\{L(\mathbf{f}) : \mathbf{f} \in \mathcal{N}\}. \tag{5.6}$$

The approach that solves the GCS problem by solving the D-GCS problem requires a zero duality gap, i.e.,

$$\inf\{C(\mathbf{x}) : \mathbf{x} \in \mathcal{X}\} = \sup\{L(\mathbf{f}) : \mathbf{f} \in \mathcal{N}\}. \tag{5.7}$$

If there is a strictly feasible solution for the GCS problem, i.e. if there is a feasible solution such that the inequalities in Equation (5.2) all hold strictly, then we can apply the strong duality theorem (see Chapter 6.2 [**59**]), which is also known as Slater's constraint qualification, to obtain the following theorem.

Theorem 5.5. *If there is a strictly feasible solution to the GCS problem, then the duality gap is zero and there exists a saddle point* $(\mathbf{x}, \mathbf{t}, \mathbf{f})$ *such that: first,* $\mathbf{f}$ *is the optimal solution of the D-GCS problem; second,* $(\mathbf{x}, \mathbf{t})$ *is the optimal solution of the GCS problem; third,* $C(\mathbf{x}) = L^*(\mathbf{x}, \mathbf{t}, \mathbf{f}) = L(\mathbf{f})$.

Proof. It is implied by the strong duality theorem (see Chapter 6.2 [**59**]). □

Theorem 5.5 guarantees that in the presence of a strictly feasible solution, the GCS problem can be solved by solving the D-GCS problem. There are two misunderstandings when Chen et al. [**55**] applied the strong duality theorem to obtain a similar result. First, if transformations are necessary to convert a problem into a convex programming problem, the Lagrangian dual problem should be derived from the transformed problem instead of the original problem. Only then the strong duality theorem can be applied. More specifically, since Chen et al. [**55**] claimed the convexity through geometric programming, the Lagrangian dual problem should be derived from the geometric programming formulation. Here we show that it is not necessary to establish the convexity through geometric programming and thus the Lagrangian dual problem can be formulated as the work [**55**]. Second, the strong duality theorem requires the existence of a strictly feasible solution. For the GCS problems without a strictly feasible solution, Theorem 5.5 will not apply and Equation (5.7) should be established through other theories.

We establish Equation (5.7) via the result of Rockafellar [**66**], which is stated in the following theorem.

Theorem 5.6. *A convex function $g(\mathbf{x})$ satisfies the regularity condition if and only if $g(\mathbf{x}) = \mathbf{b}^\top\mathbf{x} + c$ or $g(\mathbf{x}) = h(A\mathbf{x}) + \mathbf{b}^\top\mathbf{x} + c$ for some finite strictly convex function $h(\mathbf{y})$, matrix A, vector $\mathbf{b}$, and scalar c.*

If the objective function C and the delay functions $d_{i,j}$ all satisfy the regularity condition, then the duality gap is zero.

Proof. It was proved by Rockafellar [**66**]. □

Although Theorem 5.6 guarantees a zero duality gap without requiring a strictly feasible solution, it does not guarantee any saddle point as in Theorem 5.5. For a GCS problem without a strictly feasible solution, the D-GCS problem may have no finite solution, i.e., $L(\mathbf{f}') \neq \sup\{L(\mathbf{f}) : \mathbf{f} \in \mathcal{N}\}$, $\forall \mathbf{f}' \in \mathcal{N}$.

5.2.2. Simplifying the Lagrangian Dual Problem

The Lagrangian function L^* can be rewritten as

$$L^*(\mathbf{x}, \mathbf{t}, \mathbf{f}) = C(\mathbf{x}) + \sum_{(i,j)\in E} f_{i,j} d_{i,j}(\mathbf{x}) + \sum_{k\in V}\Big(\sum_{(k,j)\in E} f_{k,j} - \sum_{(i,k)\in E} f_{i,k}\Big)t_k,$$

Let $\mathcal{F}$ be the set of the multipliers satisfying the flow conservation constraints, i.e.,

$$\mathcal{F} \triangleq \{\mathbf{f} : \sum_{(i,k)\in E} f_{i,k} = \sum_{(k,j)\in E} f_{k,j}, \forall k \in V\}.$$

If $\sum_{(k,j)\in E} f_{k,j} \neq \sum_{(i,k)\in E} f_{i,k}$ for some $k \in V$, then $\forall M \in \mathbb{R}$, t_k can be chosen such that $L^*(\mathbf{x}, \mathbf{t}, \mathbf{f}) < M$. In such cases, $L(\mathbf{f}) = -\infty$. Therefore, the dual problem D-GCS is

simplified by introducing the flow conservation constraints, which is stated as the following *FD-GCS* problem.

Problem 5.3 (FD-GCS)**.**

$$\begin{aligned} \textit{Maximize} \quad & L(\mathbf{f}) \\ \textit{s.t.} \quad & \mathbf{f} \in \mathcal{F} \cap \mathcal{N}. \end{aligned}$$

Note that the same FD-GCS problem has been obtained in the work [**55**] and the simplification allows the work [**55**] to maximize $L(\mathbf{f})$ on $\mathcal{F} \cap \mathcal{N}$ by applying subgradient optimizations. However, our reasoning in simplifying the D-GCS problem into the FD-GCS problem is different. The work [**55**] derived the flow conservation constraints of the multipliers from the KKT conditions of the optimal solutions. Since the KKT conditions are sufficient but not necessary for an optimal solution, it is possible that there is no feasible solution satisfying such conditions for some sizing problems and thus the derivation in [**55**] was not correct. We circumvent the difficulty by excluding the $\mathbf{f}$ satisfying $L(\mathbf{f}) = -\infty$.

We call the $\mathbf{f}$ *dual feasible* if and only if $\mathbf{f} \in \mathcal{F} \cap \mathcal{N}$. As the function $L(\mathbf{f})$ only takes finite values for dual feasible multipliers, the information regarding the objective function in both D-GCS and FD-GCS problems is "lost" for the non-negative multipliers that do not satisfy the flow conservation constraints, and thus it is very difficult to obtain good properties of the objective function in order to perform optimizations. This motivates us to "fill the vacancy" by assigning finite values to the objective function for the non-negative multipliers that do not satisfy the flow conservation constraints. Formally, for

$\mathbf{f} \in \mathcal{N}$, define

$$\begin{aligned} P_{\mathbf{f}}(\mathbf{x}) &\triangleq C(\mathbf{x}) + \sum_{(i,j)\in E} f_{i,j} d_{i,j}(\mathbf{x}), \\ Q(\mathbf{f}) &\triangleq \inf\{P_{\mathbf{f}}(\mathbf{x}) : \mathbf{x} \in \Omega\}. \end{aligned}$$

We formulate the *SD-GCS* problem as follows.

Problem 5.4 (SD-GCS)**.**

$$\begin{aligned} \textit{Maximize} \quad & Q(\mathbf{f}) \\ \textit{s.t.} \quad & \mathbf{f} \in \mathcal{F} \cap \mathcal{N}. \end{aligned}$$

It is straightforward that the FD-GCS problem and the SD-GCS problem are different since their objective functions are different, although both objective function take the same value for all dual feasible $\mathbf{f}$. Both problems are equivalent to the D-GCS problem as stated in the following theorem.

Theorem 5.7. *The D-GCS problem, the FD-GCS problem, and the SD-GCS problem are equivalent.*

Proof. For any $\mathbf{f} \notin \mathcal{F}$, since $L^*(\mathbf{x}, \mathbf{t}, \mathbf{f})$ can be arbitrarily small, $L(\mathbf{f}) \leq L(\mathbf{f}')$ holds for all $\mathbf{f}' \in \mathcal{F}$. Thus the D-GCS problem and the FD-GCS problem are equivalent.

Since $P_{\mathbf{f}}(\mathbf{x}) = L^*(\mathbf{x}, \mathbf{t}, \mathbf{f})$ for all $\mathbf{f} \in \mathcal{F}$, $\mathbf{x} \in \Omega$, and $\mathbf{t} \in \mathbb{R}^{|V|}$, $Q(\mathbf{f}) = L(\mathbf{f})$ holds for all $\mathbf{f} \in \mathcal{F}$. Thus the FD-GCS problem and the SD-GCS problem are equivalent.

Therefore, the D-GCS problem, the FD-GCS problem, and the SD-GCS problem are equivalent. □

The two most important benefits for us to formulate the SD-GCS problem are as follows. First of all, the SD-GCS problems are convex programming problems because of the following lemma.

Lemma 5.1. *Let* $\mathbf{d}$ *be the vector of all the delays* $d_{i,j}$. $Q(\mathbf{f})$ *is a concave function on* $\mathcal{N}$ *and* $\mathbf{d}(\mathbf{x_f})$ *is the subgradient.*

Proof. Assume that both $\mathbf{f}$ and $\mathbf{f}'$ belong to $\mathcal{N}$. Then

$$\begin{aligned} Q(\mathbf{f}') &= \inf\{P_{\mathbf{f}'}(\mathbf{x}) : \mathbf{x} \in \Omega\} \\ &\leq P_{\mathbf{f}'}(\mathbf{x_f}) \\ &= P_{\mathbf{f}}(\mathbf{x_f}) + \mathbf{d}(\mathbf{x_f})^\top(\mathbf{f}' - \mathbf{f}) \\ &= Q(\mathbf{f}) + \mathbf{d}(\mathbf{x_f})^\top(\mathbf{f}' - \mathbf{f}) \end{aligned}$$

Therefore, $Q(\mathbf{f})$ is a concave function on $\mathcal{N}$ and $\mathbf{d}(\mathbf{x_f})$ is the subgradient. □

Second, as proved later, the objective function $Q(\mathbf{f})$ is differentiable on $\mathcal{N}$ for the SD-GCS problems derived from proper GCS problems. Thus we are able to design the DualFD algorithm to solve the proper GCS problems through solving their corresponding SD-GCS problems.

5.2.3. A Trivial Example

One could be deceived to think that the situations mentioned in the previous sections, i.e. there is no saddle point or there is no feasible solution satisfying the KKT conditions, only happen for "corner" cases of the sizing problems. However, such situations may occur for

extremely trivial circuits and for more complicated circuits containing those trivial ones as the sub-circuits.

Consider the following problem that might be formulated from optimizing a single inverter with the size a.

$$\begin{aligned} \text{Minimize} \quad & a \\ \text{s.t.} \quad & t_1 + a \le t_2,\ t_2 + \frac{1}{a} \le t_3,\ t_3 \le t_1 + 2, \\ & \frac{1}{2} \le a \le 2. \end{aligned}$$

The problem is a GCS problem of the variable $x = \ln a$. There is only one feasible solution $x = 0$ which is also the optimal solution while there is no strictly feasible solution. Consider the following two KKT conditions,

$$\begin{aligned} \frac{\partial L^*}{\partial x} &= e^x + f_{1,2}e^x - f_{2,3}e^{-x} = 0, \\ \frac{\partial L^*}{\partial t_2} &= f_{2,3} - f_{1,2} = 0. \end{aligned}$$

They cannot be satisfied simultaneously for the optimal solution $x = 0$.

On the other hand, for the dual feasible $\mathbf{f}$, there should exist $\beta \ge 0$ such that $\beta = f_{1,2} = f_{2,3} = f_{3,1}$. Define

$$\begin{aligned} q(\beta) &\triangleq \inf\{e^x + \beta(e^x + e^{-x} - 2) : -\ln 2 \le x \le \ln 2\}, \\ x_\beta &\triangleq \operatorname{argmin}_{-\ln 2 \le x \le \ln 2}(e^x + \beta(e^x + e^{-x} - 2)). \end{aligned}$$

For the SD-GCS problem, it is simplified as maximizing $q(\beta)$ for $\beta \geq 0$. The $q(\beta)$ and x_β can be computed as

$$(q(\beta), x_\beta) = \begin{cases} (\frac{1+\beta}{2}, -\ln 2), & \text{if } 0 \leq \beta < \frac{1}{3}, \\ \left(\frac{2}{\sqrt{1+1/\beta}+1}, \ln\sqrt{\frac{\beta}{\beta+1}}\right), & \text{if } \beta \geq \frac{1}{3}. \end{cases}$$

So $Q(\mathbf{f}) < 1$ for any $\mathbf{f} \in \mathcal{F} \cap \mathcal{N}$ and thus there is no saddle point. Note that there is no x_β being feasible for any $\beta \geq 0$.

5.3. Solving the Simplified Dual Problems

5.3.1. Solving the Lagrangian Subproblem

Solving either the FD-GCS problem or the SD-GCS problem requires solving the Lagrangian subproblem first, i.e. to compute $L(\mathbf{f})$ or equivalently $Q(\mathbf{f})$ for a given dual feasible $\mathbf{f}$, which is in turn equivalent to minimize $P_{\mathbf{f}}(\mathbf{x})$ for $\mathbf{x} \in \Omega$. The following lemma states the property of $P_{\mathbf{f}}(\mathbf{x})$.

Lemma 5.2. *For any $\mathbf{f} \in \mathcal{N}$, $P_{\mathbf{f}}(\mathbf{x})$ is a twice differentiable convex function on Ω.*

Proof. Recall that the delay functions $d_{i,j}(\mathbf{x})$ and the objective function $C(\mathbf{x})$ are twice differentiable convex function on Ω. For any $\mathbf{f} \in \mathcal{N}$, $P_{\mathbf{f}}(\mathbf{x})$ is a weighted summation of twice differentiable convex functions with nonnegative weights. Therefore, for any $\mathbf{f} \in \mathcal{N}$, $P_{\mathbf{f}}(\mathbf{x})$ is a twice differentiable convex function on Ω. □

Let the optimal solution of the Lagrangian subproblem be $\mathbf{x_f}$, i.e.,

$$\mathbf{x_f} \triangleq \text{argmin}_{\mathbf{x}\in\Omega} P_{\mathbf{f}}(\mathbf{x}).$$

Note that $\mathbf{x_f}$ is not necessarily unique. According to Lemma 5.2, the optimal solution can be computed by existing convex programming algorithms (e.g. Chapter 8 [**59**]). Then $Q(\mathbf{f})$ is equal to $P_{\mathbf{f}}(\mathbf{x_f})$.

For the Elmore delay model, Chen et al. [**55**] proposed a greedy algorithm that iteratively sizes each gate and wire segment to solve the LRS/μ problem, which is similar to minimize $P_{\mathbf{f}}(\mathbf{x})$. Generally speaking, this algorithm is a descent method that uses coordinate axes as the search directions. Chen et al. proved that if the algorithm starts with all the gates and wire segments at their minimum sizes, then it converges to the optimal solution. We show that such algorithm can be extended to handle arbitrary convex delays and prove that the algorithm will converge to the optimal solution for any initial solution.

Given any $\mathbf{x} \in \Omega$ and any permutation π of the coordinate directions, let $\mathrm{LRS}_{\mathbf{f}}(\mathbf{x}, \pi)$ be the vector obtained by sequentially minimizing $P_{\mathbf{f}}$ on Ω along the directions following the ordering defined by π in one iteration of the greedy algorithm. Then for a given permutation π, $\mathrm{LRS}_{\mathbf{f}}(\mathbf{x}, \pi)$ is continuous for $\mathbf{x}$. The following theorem holds for the sequence obtained by applying the algorithm to any initial solution.

Theorem 5.8. *For any initial solution* $\mathbf{x}(0) \in \Omega$*, each accumulation point of the sequence obtained by the iterative greedy algorithm, i.e.,* $\mathbf{x}(i+1) = LRS_{\mathbf{f}}(\mathbf{x}(i), \pi_i)$*,* $i = 0, 1, \ldots$*, minimizes* $P_{\mathbf{f}}$ *on* Ω*.*

Proof. Since Ω is compact, let $\mathbf{x}^* \in \Omega$ be an accumulation point and assume the infinite sub-sequence of $\mathbf{x}(i)$ indexed by the set $\mathcal{I}$ converges to $\mathbf{x}^*$. Because the number of the permutations of the coordinate directions is finite, there exists a subset $\mathcal{I}'$ of $\mathcal{I}$ with infinite number of elements satisfying that, $\forall i \in \mathcal{I}'$, $\mathbf{x}(i+1)$ is obtained from $\mathbf{x}(i)$

with the same permutation π'. Moreover, since $\mathbf{x}(i+1) \in \Omega$ for every $i \in \mathcal{I}'$ and Ω is compact, there exists a subset $\mathcal{I}''$ of $\mathcal{I}'$ with infinite number of elements such that the infinite sequence $\mathbf{x}(i+1)$, $i \in \mathcal{I}''$, converges to a point $\mathbf{x}' \in \Omega$. For $i \in \mathcal{I}''$, since $\mathbf{x}(i+1) = \mathrm{LRS}_{\mathbf{f}}(\mathbf{x}(i), \pi')$, we have $\mathbf{x}' = \mathrm{LRS}_{\mathbf{f}}(\mathbf{x}^*, \pi')$ when $i \to +\infty$. On the other hand, since $P_{\mathbf{f}}$ is continuous and Ω is compact, $P_{\mathbf{f}}$ is lower-bounded on Ω. Because $P_{\mathbf{f}}(\mathbf{x}(i))$ is non-increasing when i increases, $P_{\mathbf{f}}(\mathbf{x}(i))$ converges when $i \to +\infty$. Therefore, as both $\mathbf{x}^*$ and $\mathbf{x}'$ are the accumulation points, we have $P_{\mathbf{f}}(\mathbf{x}') = P_{\mathbf{f}}(\mathbf{x}^*)$. So,

$$P_{\mathbf{f}}(\mathrm{LRS}_{\mathbf{f}}(\mathbf{x}^*, \pi')) = P_{\mathbf{f}}(\mathbf{x}^*).$$

Since $P_{\mathbf{f}}$ is differentiable and convex on Ω according to Lemma 5.2, we have that $\mathbf{x}^*$ minimizes $P_{\mathbf{f}}$ on Ω. □

The advantage of our approach is as follows. When the simplified dual problems are solved iteratively by improving the dual feasible $\mathbf{f}$, intuitively $\mathbf{f}$ would be changed by a small amount from an iteration to another and then the changes in $\mathbf{x}_{\mathbf{f}}$ would be small. Therefore, the algorithm may converge faster if it starts with the previous $\mathbf{x}_{\mathbf{f}}$, while the convergence is guaranteed by Theorem 5.8.

5.3.2. Solving FD-GCS by Subgradient Optimizations

The FD-GCS problem was solved by subgradient optimizations [**55, 57**] because its objective function $L(\mathbf{f})$ is not differentiable in general. We introduce the algorithm with some modifications as the SubGrad algorithm below, which will be used as a comparison to our DualFD algorithm as presented in the later subsections.

Starting from any dual feasible $\mathbf{f}$, the SubGrad algorithm iteratively improves $\mathbf{f}$ and computes $\mathbf{x_f}$ until convergence. In the beginning of each iteration, the Lagrangian multipliers are updated as suggested by the work [**55**] using the step size ρ_k and the subgradient $t_i + d_{i,j}(\mathbf{x_f}) - t_j$, where k is the iteration number and t_i are the arrival times. Note that a heuristic was proposed in the work [**57**] to replace the above standard updating method. That heuristic was intended for faster convergences with a good initial $\mathbf{f}$ obtained from a pre-processing step and is not suitable for the SubGrad algorithm for the following reasons. First, the theoretical convergence of subgradient optimizations is not guaranteed by the heuristic. Second, our empirical study showed that the heuristic only converges when the initial multipliers $\mathbf{f}$ are near optimal. For the extreme case where the initial multipliers are all 0, the heuristic will not make any progress since all the multipliers that are 0 initially will remain 0.

After the Lagrangian multipliers are updated, they would not necessarily be dual feasible. Thus $\mathbf{f}$ should be projected to the closest one that is dual feasible. As it is not clear how exactly such projection was done in the work [**55**], the SubGrad algorithm follows the work [**57**] to perform projection by distributing incoming flows proportionally to the outgoing edges according to the existing outgoing flows. Note that this proportional projection method only applies to the GCS problems in which removing one edge from G, e.g. the O to I edge, results in a DAG; otherwise, it may fail, e.g. for the simultaneous sizing and clock skew optimization problem.

At the end of the iteration, the Lagrangian subproblem is solved to compute $\mathbf{x_f}$ for the current $\mathbf{f}$. To claim convergence, Chen et al. [**55**] suggested to terminate the iterations when $C(\mathbf{x_f}) - L(\mathbf{f})$ is less than a pre-defined small positive error bound. However, this

is only correct if $\mathbf{x_f}$ is feasible since the duality gap is defined between a feasible $\mathbf{x}$ and a dual feasible $\mathbf{f}$ as shown in Equation (5.6) and (5.7). Otherwise, the algorithm can be terminated prematurely. For example, if the objective function is the total gate and wire size, then for the multipliers to be all 0, we must have that all the gate and wire segments are at their minimum sizes and thus $C(\mathbf{x}) - L(\mathbf{f}) = 0$. In such situation, there is no guarantee that the timing constraints would be satisfied. In the SubGrad algorithm, when $\mathbf{x_f}$ is feasible, we claim convergence if $C(\mathbf{x_f}) - L(\mathbf{f})$ is small enough; otherwise, we claim convergence if the changes in $\mathbf{x_f}$, $\mathbf{f}$, and $L(\mathbf{f})$ are marginal.

Although the convergence is claimed for both works [**55, 57**], it is not clear how to prove such convergence according to the discussions above since the convergence of subgradient optimizations depends on the choice of the step sizes and the projection method (see Chapter 8.9 [**59**]). We should also point out that even when theoretical convergence is guaranteed, practical convergence of the subgradient optimizations is difficult and usually requires a good initial solution and a good step size sequence (see Chapter 8.9 [**59**]). Because the SubGrad algorithm is not the focus of this chapter, we leave further improvements to future researches.

5.3.3. Detecting Infeasible GCS Problems

The GCS problem can be infeasible, e.g. the timing constraints of a circuit cannot be met no matter how the gates and wires are sized. Since the objective function $C(\mathbf{x})$ is continuous on the compact set Ω, it is upper-bounded. We have the following theorem.

Theorem 5.9. *Suppose C is upper-bounded by $U \in \mathbb{R}$ on Ω. If the duality gap is zero, then the GCS problem is feasible if and only if $Q(\mathbf{f}) \leq U$, $\forall \mathbf{f} \in \mathcal{F} \cap \mathcal{N}$.*

Proof. Suppose $(\mathbf{x}^*, \mathbf{t}^*)$ is a feasible solution, then we have that $\forall \mathbf{f} \in \mathcal{F} \cap \mathcal{N}$,

$$\begin{aligned} Q(\mathbf{f}) &\leq P_{\mathbf{f}}(\mathbf{x}^*) \\ &= C(\mathbf{x}^*) + \sum_{(i,j)\in E} f_{i,j}(t_i^* + d_{i,j}(\mathbf{x}^*) - t_j^*) \\ &\leq C(\mathbf{x}^*) \\ &\leq U. \end{aligned}$$

On the other hand, if $\forall \mathbf{f} \in \mathcal{F} \cap \mathcal{N}$, $Q(\mathbf{f}) \leq U$, then there is at least one feasible solution; otherwise the duality gap is not zero. Therefore, the theorem holds. □

Theorem 5.9 provides a method to check whether the GCS problem is feasible when maximizing $Q(\mathbf{f})$ in our DualFD algorithm presented in the following subsections. Since $L(\mathbf{f})$ will take the same value as $Q(\mathbf{f})$ for any dual feasible $\mathbf{f}$, we also integrate such feasibility checking method into the SubGrad algorithm.

For a sizing problem where $Q(\mathbf{f}) > U$ for some dual feasible $\mathbf{f}$, by investigating the non-zero multipliers, one can identify the troublesome part of the circuit that makes it infeasible. Other optimization techniques could be performed to that part to make the circuit feasible and we leave this as a future research topic.

5.3.4. Objective Functions of Proper SD-GCS Problems are Differentiable

Define the SD-GCS problem to be proper if the GCS problem is proper. We prove the following theorem stating that Q is differentiable for proper SD-GCS problems.

Theorem 5.10. *For a proper SD-GCS problem, the objective function $Q(\mathbf{f})$ is differentiable for $\mathbf{f} \in \mathcal{N}$ and the gradient is $\mathbf{d}(\mathbf{x_f})$ where $\mathbf{x_f}$ is the only vector in Ω that minimizes $P_{\mathbf{f}}(\mathbf{x})$.*

Proof. We first claim that, in a proper GCS problem, for any $\mathbf{f} \in \mathcal{N}$, there is a single $\mathbf{x_f} \in \Omega$ satisfying that $Q(\mathbf{f}) = P_{\mathbf{f}}(\mathbf{x_f})$. We prove the claim by contradiction. Assume $Q(\mathbf{f}) = P_{\mathbf{f}}(\mathbf{x}') = P_{\mathbf{f}}(\mathbf{x}'')$ for some $\mathbf{x}' \neq \mathbf{x}''$, $\mathbf{x}' \in \Omega$, $\mathbf{x}'' \in \Omega$, and $\mathbf{f} \in \mathcal{N}$. Define

$$\mathbf{y}^{\gamma} \triangleq (1-\gamma)\mathbf{x}' + \gamma\mathbf{x}''.$$

Then $\mathbf{y}^{\gamma} \in \Omega$, $\forall 0 \leq \gamma \leq 1$. Since $P_{\mathbf{f}}(\mathbf{x})$ is convex according to Lemma 5.2, we have that $\forall 0 \leq \gamma \leq 1$,

$$Q(\mathbf{f}) = (1-\gamma)P_{\mathbf{f}}(\mathbf{x}') + \gamma P_{\mathbf{f}}(\mathbf{x}'') \geq P_{\mathbf{f}}(\mathbf{y}^{\gamma}).$$

Since $Q(\mathbf{f}) \leq P_{\mathbf{f}}(\mathbf{y}^{\gamma})$ by the definition of Q, we must have,

$$P_{\mathbf{f}}(\mathbf{y}^{\gamma}) = Q(\mathbf{f}), \forall 0 \leq \gamma \leq 1.$$

Since $P_{\mathbf{f}}(\mathbf{x})$ is twice differentiable according to Lemma 5.2, let $H_{P_{\mathbf{f}}}$ be its Hessian matrix. Let $\mathbf{z}^0 = \mathbf{y}^{\frac{1}{2}}$ and $\mathbf{z} = \mathbf{x}'' - \mathbf{x}'$. Then $\mathbf{z}^0 \in \Omega$ and $\mathbf{z} \neq 0$. Therefore,

$$\mathbf{z}^{\top}\nabla P_{\mathbf{f}}(\mathbf{z}^0) = \lim_{\lambda \to 0} \frac{P_{\mathbf{f}}(\mathbf{z}^0 + \lambda\mathbf{z}) - P_{\mathbf{f}}(\mathbf{z}^0)}{\lambda} = \lim_{\lambda \to 0} \frac{P_{\mathbf{f}}(\mathbf{y}^{\frac{1}{2}+\lambda}) - P_{\mathbf{f}}(\mathbf{y}^{\frac{1}{2}})}{\lambda} = 0,$$

and,

$$\frac{\mathbf{z}^{\top}H_{P_{\mathbf{f}}}(\mathbf{z}^0)\mathbf{z}}{2} = \lim_{\lambda \to 0} \frac{P_{\mathbf{f}}(\mathbf{z}^0 + \lambda\mathbf{z}) - P_{\mathbf{f}}(\mathbf{z}^0) - \lambda\mathbf{z}^{\top}\nabla P_{\mathbf{f}}(\mathbf{z}^0)}{\lambda^2} = \lim_{\lambda \to 0} \frac{P_{\mathbf{f}}(\mathbf{y}^{\frac{1}{2}+\lambda}) - P_{\mathbf{f}}(\mathbf{y}^{\frac{1}{2}}) - 0}{\lambda^2} = 0. \tag{5.8}$$

On the other hand, it is straightforward that because of the convexity, the Hessian matrixes $H_{d_{i,j}}$ of all the delays $d_{i,j}$ are positive semidefinite for any $\mathbf{x} \in \Omega$. Then

$$\mathbf{z}^\top H_{d_{i,j}}(\mathbf{z}^0)\mathbf{z} \geq 0.$$

Recall that the Hessian matrix $H_C(\mathbf{x})$ of the objective function C of the proper GCS problem is positive definite for any $\mathbf{x} \in \Omega$. Then

$$\mathbf{z}^\top H_C(\mathbf{z}^0)\mathbf{z} > 0.$$

Therefore,

$$\begin{aligned} \mathbf{z}^\top H_{P_\mathbf{f}}(\mathbf{z}^0)\mathbf{z} &= \mathbf{z}^\top H_C(\mathbf{z}^0)\mathbf{z} + \sum_{(i,j)\in E} f_{i,j}(\mathbf{z}^\top H_{d_{i,j}}(\mathbf{z}^0)\mathbf{z}) \\ &> 0. \end{aligned}$$

This contradicts Equation (5.8). Thus our claim holds.

Since in a proper GCS problem, for any $\mathbf{f} \in \mathcal{N}$, there is a single $\mathbf{x_f} \in \Omega$ satisfying that $Q(\mathbf{f}) = P_\mathbf{f}(\mathbf{x_f})$, according to Theorem 6.3.3 [**59**], $Q(\mathbf{f})$ is differentiable for $\mathbf{f} \in \mathcal{N}$ and the gradient is $\mathbf{d}(\mathbf{x_f})$. □

5.3.5. Solving Proper SD-GCS Problems via Method of Feasible Directions and Min-Cost Network Flow

As the objective function $Q(\mathbf{f})$ of a proper SD-GCS problem has the gradient $\mathbf{d}(\mathbf{x_f})$ according to Theorem 5.10, we apply the method of feasible directions (see Chapter 10 [**59**]) to solve the proper SD-GCS problem.

For any dual feasible $\mathbf{f}$, the vector $\Delta\mathbf{f}$ is an improving feasible direction if and only if there exists $\lambda > 0$ such that

$$Q(\mathbf{f}) < Q(\mathbf{f} + \lambda\Delta\mathbf{f}),$$

and,

$$\mathbf{f} + \lambda\Delta\mathbf{f} \in \mathcal{N} \cap \mathcal{F}.$$

An improving feasible direction can be found by solving the following direction finding (DF) problem. The intuition is to maximize the first order approximation of Q in a dual feasible neighborhood.

Problem 5.5 (DF)**.**

$$\begin{aligned} \textit{Minimize} \quad & -\mathbf{d}(\mathbf{x_f})^\top \Delta\mathbf{f} \\ \textit{s.t.} \quad & \mathbf{f} + \Delta\mathbf{f} \in \mathcal{F}, \\ & \max\{-u, -f_{i,j}\} \leq \Delta f_{i,j} \leq u, \forall (i,j) \in E. \end{aligned}$$

In the DF problem, the improving feasible direction $\Delta\mathbf{f}$ are the decision variables, $\mathbf{f}$ should be dual feasible, and u is a positive constant. It can be verified that the DF problem is a min-cost network flow problem (see [**67**]): the variables $\Delta\mathbf{f}$ are the flows on the edges in the graph G, the mass balance constraints are $\mathbf{f} + \Delta\mathbf{f} \in \mathcal{F}$, and the flow bound constraints are $\max\{-u, -f_{i,j}\} \leq \Delta f_{i,j} \leq u$, $\forall (i,j) \in E$. This special property of the DF problem comes from the special structure in the GCS problem where the timing constraints are formulated as a system of difference inequalities. Note that although a min-cost network flow problem can be solved by general linear programming techniques

since it is a special linear programming problem, it is usually more efficient to apply algorithms specifically designed for min-cost network flow problems.

Since $\mathbf{f}$ is dual feasible, $\Delta\mathbf{f} = \mathbf{0}$ is always a feasible solution of the DF problem and the optimal objective is always non-positive. The following theorem relates the optimal solution of the DF problem to either an improving feasible direction or the optimal solution of the GCS problem.

Theorem 5.11. *Suppose $\Delta\mathbf{f}^*$ is the optimal solution of the DF problem. If the optimal objective is 0, i.e., $-\mathbf{d}(\mathbf{x_f})^\top \Delta\mathbf{f}^* = 0$, then there exists a vector $\mathbf{t}$ such that $(\mathbf{x_f}, \mathbf{t})$ is the optimal solution of the GCS problem; otherwise $\Delta\mathbf{f}^*$ is an improving feasible direction.*

Proof. We first assume that the optimum objective is 0. Consider the flow network constructed from the graph G with all the arc capacities being $[0, +\infty)$ and the arc costs being $d_{i,j}(\mathbf{x_f})$. Since $\mathbf{f}$ is a nonnegative flow on G, let G_R be the residual network derived from the flow $\mathbf{f}$ as follows. The network G_R has the same vertex set as G. If there is an edge (i, j) in G, then there is an arc (i, j) in G_R with the arc cost $d_{i,j}(\mathbf{x_f})$. Moreover, if $f_{i,j} > 0$, then there is an arc (j, i) in G_R with the arc cost $-d_{i,j}(\mathbf{x_f})$. We claim that there is no positive cycle in G_R with respect to the arc costs; otherwise incrementing the flow along the positive cycle would result in a feasible solution of the DF problem with negative objective and thus violates the assumption of the 0 optimal objective. Thus, by applying the Bellman-Ford algorithm on G_R, we can obtain the arrival times $\mathbf{t}$ satisfying that,

$$\begin{aligned} t_i + d_{i,j}(\mathbf{x_f}) - t_j \leq 0, \quad & \forall (i,j) \in E, \\ t_j - d_{i,j}(\mathbf{x_f}) - t_i \leq 0, \quad & \forall ((i,j) \in E) \wedge (f_{i,j} > 0). \end{aligned} \tag{5.9}$$

It implies that,

$$(f_{i,j} = 0) \vee (t_i + d_{i,j}(\mathbf{x_f}) - t_j = 0), \forall (i,j) \in E.$$

Therefore,

$$\begin{aligned} Q(\mathbf{f}) &= C(\mathbf{x_f}) + \sum_{(i,j)\in E} f_{i,j} d_{i,j}(\mathbf{x_f}) \\ &= C(\mathbf{x_f}) + \sum_{(i,j)\in E} f_{i,j}(t_i + d_{i,j}(\mathbf{x_f}) - t_j) \\ &= C(\mathbf{x_f}) \end{aligned}$$

Since Equation (5.9) implies that $(\mathbf{x_f}, \mathbf{t})$ is a feasible solution of the GCS problem, together with Equation (5.6) we can claim that $(\mathbf{x_f}, \mathbf{t})$ is the optimal solution of the GCS problem.

We then assume that the optimum objective is negative, i.e., $-\mathbf{d}(\mathbf{x_f})^\top \Delta\mathbf{f}^* < 0$. Since $\mathbf{d}(\mathbf{x_f})$ is the gradient of $Q(\mathbf{f})$ according to Theorem 5.10, we have,

$$\lim_{\lambda \to 0} \frac{Q(\mathbf{f} + \lambda \Delta \mathbf{f}^*) - Q(\mathbf{f})}{\lambda} = \mathbf{d}(\mathbf{x_f})^\top \Delta \mathbf{f}^* > 0.$$

Thus there exists a positive λ_0 such that,

$$Q(\mathbf{f} + \lambda \Delta \mathbf{f}^*) - Q(\mathbf{f}) > 0, \forall 0 < \lambda < \lambda_0. \tag{5.10}$$

On the other hand, define

$$\lambda_{\max} \triangleq \min_{\Delta f^*_{i,j} < 0} -\frac{f_{i,j}}{\Delta f^*_{i,j}},$$

or let $\lambda_{\max} = +\infty$ if $\Delta f^*_{i,j} \geq 0$, $\forall (i,j) \in E$. Then,

$$\mathbf{f} + \lambda \Delta \mathbf{f}^* \in \mathcal{N} \cap \mathcal{F}, \forall 0 < \lambda \leq \lambda_{\max}. \tag{5.11}$$

Equation (5.10) and (5.11) imply that $\Delta\mathbf{f}^*$ is an improving feasible direction. □

ALGORITHM DualFD	
Inputs The GCS problem and N.	
Outputs Optimal $\mathbf{f}$ and $\mathbf{x_f}$.	
1	$\mathbf{f} \leftarrow 0$. Compute $\mathbf{x_f}$.
2	**For** $i = 1$ to N:
3	Solve the DF problem for optimal $\Delta\mathbf{f}^*$.
4	Claim optimality if $\mathbf{d}(\mathbf{x_f})^\top \Delta\mathbf{f}^*$ is small enough.
5	Perform a line search on $Q(\mathbf{f} + \lambda\Delta\mathbf{f}^*)$ with $0 < \lambda \leq \lambda_{\max}$ for an increase in Q. Claim the problem to be infeasible if $Q(\mathbf{f} + \lambda\Delta\mathbf{f}^*) > U$ for some λ during the line search.
6	Update $\mathbf{f}$. Compute $\mathbf{x_f}$.
7	Claim optimality if the changes in $\mathbf{f}$, $\mathbf{x_f}$, and $Q(\mathbf{f})$ are marginal, or the duality gap is small enough if a feasible solution is known.

Figure 5.4. The DualFD Algorithm.

We design the DualFD algorithm as shown in Figure 5.4 to solve the proper SD-GCS problem. It starts with the dual feasible $\mathbf{f} = \mathbf{0}$ and iteratively improves $Q(\mathbf{f})$ by finding an improving feasible direction and performing a line search. On line 1 and 6, the vector $\mathbf{x_f}$ is computed by solving the Lagrangian subproblem. On line 5, a number of the Lagrangian subproblems should be solved to compute $\mathbf{x}_{\mathbf{f}+\lambda\Delta\mathbf{f}^*}$ and then $Q(\mathbf{f} + \lambda\Delta\mathbf{f}^*)$ can be obtained for line search. Since determining the exact λ that maximizes $Q(\mathbf{f} + \lambda\Delta\mathbf{f}^*)$ would be time consuming, we perform an inexact line search by Armijo's Rule (see Chapter 8.3 [**59**]) with two parameters ϵ and α, whose typical values are $\epsilon = 0.2$ and $\alpha = 2$. During the line search, the feasibility of the GCS problem is checked every time when Q is computed according to Theorem 5.9. The algorithm terminates when the problem is claimed to be infeasible on line 5, or a pre-defined rounds N of iterations are performed, or the optimal solution is found according to Theorem 5.11 on line 4, or the changes in $\mathbf{f}$, $\mathbf{x_f}$, and $Q(\mathbf{f})$ are marginal, or the duality gap is known and is small enough on line 7.

It is guaranteed in the DualFD algorithm that the objective function Q will increase strictly each iteration, which is not guaranteed by the SubGrad algorithm based on subgradient optimizations. Applying conventional successive linear programming (SLP) approaches (see Chapter 10.3 [**59**]) to solve the proper SD-GCS problem would share similar properties. However, our DualFD algorithm utilizes the special structure in the problem. First, it is not necessary to formulate a penalty problem because all the constraints in the SD-GCS problems are linear. Second, the DF problem can be solved by efficient min-cost network flow algorithms instead of through slower general linear programming techniques. Third, our DualFD algorithm guarantees to find an increase in the objective function Q by a line search where the SLP approaches require a trust-region framework.

5.4. Experiments

5.4.1. Experimental Setup

We implement the DualFD algorithm in C++. The DF problem is solved by the min-cost network flow solver CS2 version 4.3 [**50**]. The code is compiled by GCC version 3.4 and runs on a Linux workstation with two 927MHz Pentium III processors and 512M memory. We use the sequential circuits from the ISCAS89 benchmark suite as our test cases. The Elmore delay model is used for gate delays. We minimize the total gate sizes under a given clock period bound T_0 with two settings. In the first setting of gate sizing without clock skew optimization, the clock skews are not optimized and are fixed at 0. In the second setting of simultaneous gate sizing and clock skew optimization, the clock skews are optimized within the range $[0, \frac{T_0}{2}]$. In both settings, the arrival times at the primary input ports are 0 and the required arrival times at the primary output ports are T_0. We

associate different delay functions $d_{i,j}$ with each timing arc from an input port i to an output port j of a given gate. For each timing arc (i, j), a weight $w_{i,j}$ is introduced. The driving resistance associated with each timing arc (i, j) in each gate is $R^{nom}w_{i,j}/x$ and the load capacitance is $C^{nom}w_{i,j}x$, where R^{nom} and C^{nom} are the characteristic parameters of the gate and x is the size of the gate. An utility program reads the circuits and generates the proper GCS problems to be solved. Transistor sizing, wire sizing, and wire capacitances are not currently integrated into the utility program and they can be added straightforwardly if necessary. We emphasize that our chapter focuses on algorithmic developments for the sizing problem. Such experimental setup would suffices our purpose of algorithm evaluation.

The statistics of the circuits are shown in Table 5.1. We report the size of each circuit in the columns "# vertices" and "# edges". The number of the sizable gates is shown in the column "# vars". The number of the flip-flops is shown in the column "# dffs". For each circuit, the clock period bound T_0 is computed as half of the minimum clock period that the circuit can operate assuming that each gate takes its average size and all the clock skews are 0. Note that we do not know whether there is a feasible solution for each GCS problem at this point.

To determine if a circuit is feasible, we use Theorem 5.9 and compute U as the total gate size assuming that each gate takes its maximum size. To evaluate the algorithms, we need to collect feasible sizing solutions during the optimization to compute the duality gap. We also need to determine the minimum feasible clock period, i.e. the minimum clock period that a circuit can operate, for the final sizing solutions. For the experimental setting of gate sizing without clock skew optimization, the computation is straightforward since

Table 5.1. Statistics of the circuits for gate sizing.

name	# vertices	# edges	# vars	# dffs
s27	40	51	13	3
s208	291	374	104	8
s298	396	540	133	14
s344	470	607	175	15
s349	475	615	176	15
s382	511	685	179	21
s386	527	756	165	6
s420	585	748	212	16
s444	580	777	202	21
s510	668	902	217	6
s526	685	963	214	21
s641	993	1195	398	19
s713	1059	1298	412	19
s820	1076	1655	294	5
s832	1086	1680	292	5
s838	1161	1484	422	32
s953	1214	1670	424	29
s1196	1590	2259	547	18
s1238	1601	2343	526	18
s1423	1988	2573	731	74
s1488	2062	2868	659	6
s1494	2062	2880	653	6
s5378	7386	9046	2958	179
s9234	14028	16681	5808	211
s13207	20456	24473	8589	638
s15850	24564	29158	10306	534
s35932	47827	63569	17793	1728
s38417	57509	69099	23815	1636
s38584	54901	70133	20679	1426

removing the edge (O, I) from G results in a DAG. For any sizing solution, the minimum feasible clock period is computed as the longest path delay following the topological order of the DAG (see Chapter 24.1 [**29**]). The sizing solution is feasible if and only if this clock period is not larger than T_0. On the other hand, for the experimental setting of simultaneous gate sizing and clock skew optimization, we implement a heuristic

improvement of the Bellman-Ford algorithm [**68**] and apply Theorem 5.2 to check if a sizing solution is feasible. Computing the minimum feasible clock period under this setting is a minimum cost-to-time ratio cycle problem (see Chapter 5.7 [**67**]). Since this computation is only required for the final sizing solutions, we perform binary search on the clock period to determine the minimum feasible one.

5.4.2. Gate Sizing without Clock Skew Optimization

For comparison, we implement the SubGrad algorithm in C++. We do not implement the pre-processing heuristics in the work [**57**] because we want to compare the algorithms that solve the simplified dual problems while the pre-processing heuristics apply to both the DualFD and the SubGrad algorithms. The code of the SubGrad algorithm is compiled with the same compiler and runs on the same machine as that of the DualFD algorithm. The results are compared in Table 5.2.

For each circuit, we first run the SubGrad algorithm. The algorithm is allowed to run at most 600 seconds. The results are reported under the columns "SubGrad" as follows. Same to the DualFD algorithm, we start the SubGrad algorithm with $\mathbf{f} = \mathbf{0}$. We use a step size in the form c/k where k is the current iteration number. Multiple settings of c are manually tested. The best final solution in terms of the minimum feasible clock period T and the value of the function $L(\mathbf{f})$ is reported. The total gate size is shown in the column "area". The value of the function $L(\mathbf{f})$ is shown in the column "dual". The ratio of T to the clock period bound T_0 is shown in the column "T/T_0". The number of the Lagrangian subproblems being solved and the total runtime in seconds are shown

in the columns "LRS" and "t" respectively. The circuits "s420", "s838", and "s35932" marked by * in the table are determined to be infeasible according to Theorem 5.9.

Then, we run the DualFD algorithm for each circuit once and obtain the final sizing solution. The algorithm is allowed to run at most 200 iterations and 300 seconds. The results are reported under the columns "DualFD" as follows. The columns "area", "T/T_0", "LRS", and "t" have similar meanings as those of the SubGrad algorithm. The value of the function $Q(\mathbf{f})$ is shown in the column "dual". The number of the DF problem being solved is shown in the column "DF". Note that the circuits "s420" and "s838" are determined to be infeasible by the DualFD algorithm. Moreover, we use the feasible sizing solution with the minimum total gate size to estimate the duality gap. A zero duality gap proves that the sizing solution is an optimal solution according to the weak duality theorem. If there is a feasible sizing solution with total gate size area_{fea}, the upper-bound of the relative duality gap, i.e. $(\text{area}_{\text{fea}} - \text{dual})/\text{area}_{\text{fea}}$, is reported in the column "gap"; otherwise, a "-" symbol is shown. It is clear that among the 26 circuits except the 3 infeasible ones, we have proved that for 16 of them, the final sizing solutions of the DualFD algorithm are within 0.5% of the optimal solutions. Note that for the remaining 10 circuits, it is possible that they are as close to the optimal solutions – we simply cannot prove that without the knowledge of feasible solutions.

It can be seen from Table 5.2 that the DualFD algorithm consistently generates results with better quality in less runtime. For almost all the circuits, the solutions of the DualFD algorithm meet the timing specifications better than the SubGrad algorithm. For most circuits where the solutions of both algorithms are with similar minimum feasible clock period, the ones of the DualFD algorithm are with smaller total gate size. Moreover, the

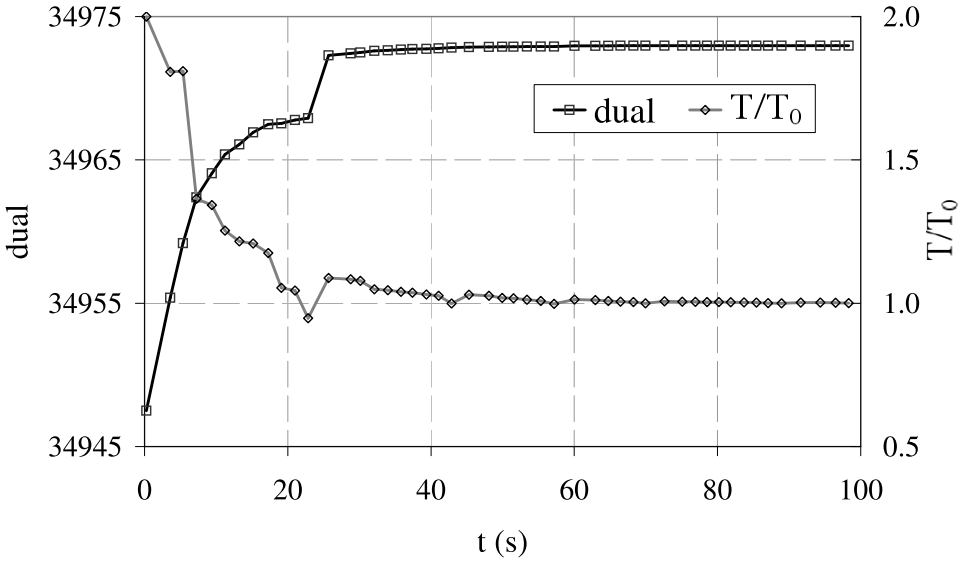

Figure 5.5. Convergence of s38584 for DualFD w/o clock skew optimization.

solutions of the DualFD algorithm are usually larger in the "dual" columns – although the columns represent the values of different functions in both algorithms, they can be compared directly since both functions take the same value for the dual feasible $\mathbf{f}$ and they are the objective functions to be maximized. It should be pointed out that the reported runtime of the SubGrad algorithm represents a single run of the algorithm. In our experiments, we usually need to run the SubGrad algorithm around 20 times for each circuit in order to locate a good step size for the algorithm to converge. However, for the DualFD algorithm, it is not necessary for us to tune the parameters manually. Thus the overall runtimes of the SubGrad algorithm are significantly longer than those of the DualFD algorithm. We believe that the advantage of the DualFD algorithm is due to the fact that it is more effective to adjust the timing of the whole circuit by the min-cost network flow technique, than by the subgradient updating based only on local delay and arrival time information.

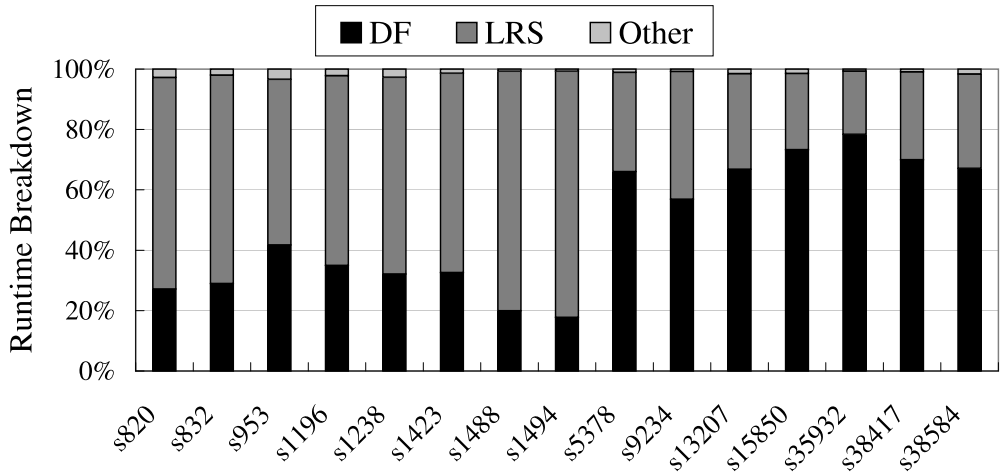

Figure 5.6. Runtime breakdown for DualFD w/o clock skew optimization.

We plot the convergence sequence of the circuit "s38584" in Figure 5.5. The runtime breakdowns of the DualFD algorithm for the largest 15 circuits are shown in Figure 5.6. The algorithm spends most of the runtime on solving the DF problems (shown as "DF") and the Lagrangian subproblems (shown as "LRS") while spending only a small portion on the other parts (shown as "Other"). The runtime spent on solving the DF problems gradually dominates the runtime spent on solving the Lagrangian subproblems for the Elmore delay model as the circuit sizes increase. If a more sophisticated but accurate delay model is used in the future, the runtime spent on solving the Lagrangian subproblems would increase but the runtime spent on solving the DF problems would remain unchanged since the complexity of the DF problem depends most on the size of the graph G. Thus, considering the difference in the number of the Lagrangian subproblems that should be solved in both the DualFD and the SubGrad algorithms, we expect that the runtime ratio of the DualFD algorithm to the SubGrad algorithm will become more significant if a more sophisticated delay model is used.

Table 5.2. Results comparison between the DualFD and the SubGrad algorithms w/o clock skew optimization.

	SubGrad					DualFD						
name	area	dual	T/T_0	LRS	t(s)	area	dual	T/T_0	DF	LRS	t(s)	gap
s27	90.9	90.9	1.000	374	0.08	90.8	90.9	1.000	15	50	0.01	0.04%
s208	244.1	327.0	1.581	7082	4.70	856.5	1074.1	1.014	200	1049	1.75	-
s298	243.4	242.2	1.001	548	0.48	242.5	242.5	1.000	35	190	0.18	0.02%
s344	315.9	333.9	1.690	5820	5.07	445.6	445.8	1.001	200	1102	2.06	0.37%
s349	314.0	332.2	1.688	5647	5.01	447.4	447.6	1.000	200	1098	2.09	0.24%
s382	316.8	313.4	0.990	924	0.84	315.7	315.7	1.000	23	139	0.16	0.01%
s386	297.6	346.6	1.853	6182	6.58	1231.7	1361.0	1.018	200	712	6.97	-
s420*	2711.3	23920.2	2.573	8	0.06	1471.0	24461.9	2.589	1	9	0.03	-
s444	361.1	360.6	1.000	151	0.19	360.7	360.6	1.000	42	228	0.35	0.01%
s510	407.7	399.5	1.620	5873	8.72	480.1	479.8	1.000	200	802	2.43	0.14%
s526	385.7	378.8	1.061	812	1.23	383.6	383.6	1.000	34	158	0.32	0.02%
s641	707.6	731.9	1.506	5375	11.37	888.7	888.9	1.000	56	217	1.10	0.22%
s713	746.4	782.5	1.648	5571	13.66	1408.3	1409.5	1.000	82	445	3.34	2.73%
s820	525.0	564.2	1.738	9839	27.21	1081.5	1093.3	1.004	200	729	7.00	-
s832	520.0	561.4	1.727	9713	27.38	1060.1	1073.9	1.006	200	728	7.25	-
s838*	6499.3	63276.9	4.720	3	0.05	2916.0	80458.3	4.769	1	9	0.07	-
s953	746.3	753.8	1.190	3574	11.58	775.9	775.5	1.002	200	866	4.56	0.06%
s1196	1012.3	988.2	1.417	8176	40.65	1089.3	1088.2	1.001	200	853	8.59	0.20%
s1238	983.5	913.4	1.441	9019	45.56	1080.4	1079.9	1.000	200	877	9.66	0.09%
s1423	1577.4	1122.2	1.113	7449	48.75	1669.6	1670.3	1.000	200	705	11.06	0.03%
s1488	1150.0	1214.5	1.813	6908	41.52	2049.4	2106.4	1.011	200	712	24.08	-
s1494	1250.4	1304.4	1.632	8252	60.34	2153.2	2316.0	1.024	200	709	25.39	-
s5378	5508.8	5455.1	1.345	14419	362.23	5877.3	6092.4	1.045	200	666	65.90	-
s9234	9982.9	9899.5	1.555	8773	397.48	12995.0	15568.7	1.076	200	824	175.41	-
s13207	14527.6	14522.2	1.433	2825	179.38	14608.7	14608.4	1.000	200	926	135.78	0.00%
s15850	17874.6	17351.2	1.131	7406	600.06	17768.6	17766.7	1.001	200	747	178.68	0.09%
s35932*	167.9K	1792.3K	1.546	12	15.80	33613.8	44583.1	1.802	79	234	302.56	-
s38417	42133.6	38631.8	1.478	2383	600.04	42789.8	45575.3	1.129	91	358	302.27	-
s38584	34974.5	34962.0	1.014	1732	319.01	34973.2	34973.0	1.000	49	248	100.25	0.00%

* The circuits are infeasible.

5.4.3. Simultaneous Gate Sizing and Clock Skew Optimization

Our utility program generates the proper GCS problems for simultaneous gate sizing and clock skew optimization according to Section 5.1.3. For each circuit, we run the DualFD algorithm once and obtain the final sizing solution. The algorithm is allowed to run at most 200 iterations and 300 seconds. The results are reported in Table 5.3. All the columns have the similar meanings as in Table 5.2 for those of the DualFD algorithm for gate sizing without clock skew optimization. The number of the DF problem being solved and the number of the Lagrangian subproblems being solved are similar to those shown in Table 5.2 and are thus not reported. Among the 29 circuits, "s420" and "s838" marked by * are determined to have no feasible solution. According to the column "gap", we have proved that for 17 of the remaining 27 circuits, the final sizing solutions are within 0.5% of the optimial solutions. The runtime breakdowns are shown in Figure 5.7 and exhibit similar trend as that in Figure 5.6. The increases in the runtime shown as "Other", which are spent on the parts other than solving the DF problems and the Lagrangian subproblems, are due to the computations that check whether the sizing solutions are feasible and that determine the minimum feasible clock period for the final sizing solutions.

In addition, we compare the final sizing solutions in Figure 5.8 to those of gate sizing without clock skew optimization obtained by the DualFD algorithm. The three infeasible circuits "s420", "s838", and "s35932" are excluded from the comparison. We show the ratio of the total gate size (shown as "area") and the minimum feasible clock period (shown as "T") with clock skew optimization to that without clock skew optimization. The benefit of allowing clock skew optimization can be seen from the figure. For most

Table 5.3. Results of the DualFD algorithm w/ clock skew optimization.

name	area	dual	T/T_0	t(s)	gap
s27	90.8	90.9	1.000	0.02	-
s208	840.3	1083.0	1.111	2.03	-
s298	227.6	227.6	1.000	0.04	0.00%
s344	317.5	317.5	1.001	1.50	0.05%
s349	319.4	319.1	1.000	1.54	0.05%
s382	305.3	305.3	1.000	0.45	0.00%
s386	1204.5	1287.8	1.017	7.47	-
s420*	1471.0	24461.9	2.589	0.04	-
s444	344.8	344.8	1.000	0.13	0.01%
s510	457.3	456.2	1.001	3.08	0.33%
s526	369.6	369.5	1.000	1.74	0.02%
s641	858.7	858.8	1.000	0.48	0.18%
s713	1399.2	1400.4	1.000	3.50	0.45%
s820	871.3	871.0	1.000	7.31	-
s832	873.8	873.5	1.000	7.99	-
s838*	2916.0	80458.3	4.769	0.07	-
s953	741.3	740.9	0.999	4.63	0.05%
s1196	1088.0	1087.5	1.000	9.11	0.07%
s1238	1078.8	1078.7	1.000	9.68	0.14%
s1423	1444.6	1444.6	1.000	1.78	0.01%
s1488	1916.6	1915.5	1.000	23.55	-
s1494	1970.1	1969.3	1.000	25.18	-
s5378	5882.9	6066.2	1.039	72.90	-
s9234	10076.1	10076.1	1.000	80.52	0.03%
s13207	14579.5	14579.3	1.001	136.48	0.00%
s15850	17527.1	17526.6	1.001	170.52	0.01%
s35932	157.1K	168.2K	1.403	302.13	-
s38417	41001.1	40965.6	1.001	301.19	-
s38584	34970.8	34969.1	1.006	301.82	0.01%

* The circuits are infeasible.

circuits, the total gate size is reduced without increasing the minimum feasible clock period.

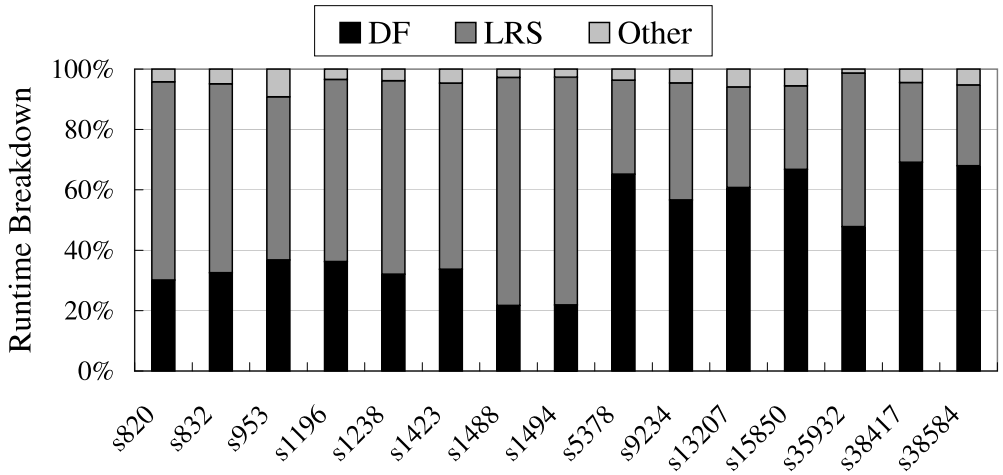

Figure 5.7. Runtime breakdown for DualFD w/ clock skew optimization.

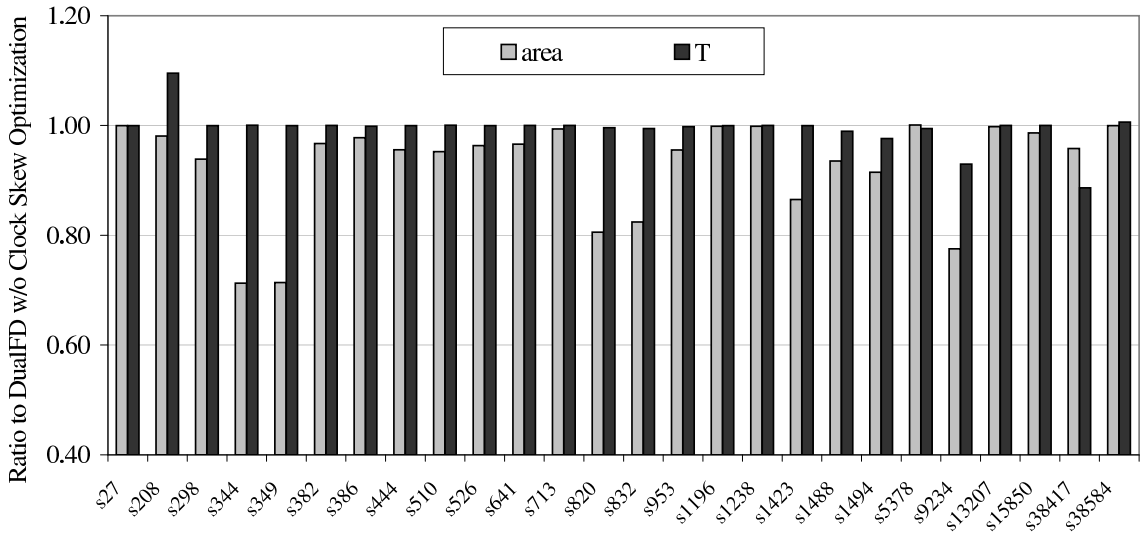

Figure 5.8. Total gate sizes and minimum feasible clock periods for DualFD w/ clock skew optimization.

5.5. Summary

In this work, we revisited the approach to solve the gate sizing problem via Lagrangian relaxation. We formulated a generalized sizing problem GCS, identified a class of proper

GCS problems, and presented a method to handle simultaneous sizing and clock skew optimization as a proper GCS problem. We established conditions for the objective function of the simplified dual problem to be differentiable and for the GCS problem to be feasible. We designed an algorithm based on the method of feasible directions and min-cost network flow to solve the proper GCS problem. The efficiency and effectiveness of our new approach was confirmed by the experimental results.

CHAPTER 6

An Efficient Incremental Algorithm for Min-Area Retiming

Retiming [**69**] is one of the most powerful sequential transformations that relocates the flip-flops (FFs) in a circuit while preserving its functionality. As relocating the FFs could balance the longest combinational paths and reduce the circuit states, the clock period and the FF area (or number) in a circuit can be reduced through retiming optimizations. As the minimum clock period (min-period) retiming minimizes the clock period, and thus might significantly increase the FF area, the minimum area (min-area) retiming minimizes the FF area under a given clock period, thus could be used to minimize the FF area even under the minimum clock period. Therefore, the min-area retiming problem is more important for sequential circuit design, but of higher complexity [**70**].

All existing provable approaches to min-area retiming follow the basic ideas of Leiserson and Saxe [**69**]. Given a circuit represented as a graph of n vertices and m edges, the minimum number of FFs between any two vertices and the maximum delay over the paths of the minimum number of FFs are first computed. Then, besides one constraint for each edge requiring that the FF number is nonnegative, for each pair of the vertices whose computed path delay is larger than the given clock period, i.e. the timing critical path, a constraint is generated requiring that there is at least one FF between them. Minimizing the FF area under those constraints formulates a dual of the min-cost network flow problem. Since each constraint forms an arc in the flow network, the number of arcs

in the network is usually $\Theta(n^2)$. Even though polynomially solvable, min-cost network flow computation (see [**67**]) over a dense graph is usually expensive on large problems.

Shenoy et al. [**70**] were among the first to consider a practical implementation of the min-area retiming algorithm. They found that the storage requirement to compute the timing critical paths and the number of constraints are the bottleneck and proposed techniques to reduce memory usage and to prune some redundant constraints. *Minaret*, proposed by Maheshwari et al. [**71**], further pruned redundant constraints to reduce the size of the flow network by exploring the equivalence of retiming and clock skew optimization as proposed in ASTRA [**72**]. However, even with these pruning techniques, as experimental results indicate, the flow networks could still be very dense compared to the original circuit graphs. Our experiments with Minaret showed that a circuit with more than 180K gates had to formulate and solve a min-cost network flow problem with more than 122M arcs, which used up all the 2GB virtual memory on our machine.

Recently, Zhou [**73**] proposed an efficient incremental algorithm for min-period retiming which iteratively moves FFs to decrease the clock period while guarantees to find the optimal solution in a short time. To overcome the expenses of existing approaches to min-area retiming, Singh et al. [**74**] also proposed to incrementally move FFs in the circuit. However, since they only allow moves that are better in cost and feasible in timing, their approach is a heuristic that may end up with a suboptimal solution. An efficient incremental algorithm with provably optimal solution is evasive till now.

In this chapter, we fill the gap with an efficient algorithm named *iMinArea* that solves the min-area retiming problem incrementally and optimally. Contrary to existing algorithms, iMinArea directly attacks the FF area minimization problem instead of its

dual network flow problem. Starting with the circuit satisfying the clock period constraint, iMinArea will iteratively reduce the number of FFs by moving FFs backward over some gates with fanouts larger than fanins. If a fanout edge currently has no FF or is on a timing critical path requesting at least one FF, such a move may require FF moves over other gates. iMinArea dynamically maintains such relations among the gates as a forest. If there is a cluster of gates whose fanouts are larger than its fanins, the number of FFs can be reduced by moving one FF over the cluster; otherwise, the current solution is optimal. An outstanding feature of iMinArea is that a critical timing constraint is dynamically generated only when it is needed. Maintaining a forest on the gates, it requests only linear storage (that is, $O(n)$) on top of the circuit graph. As can be expected, iMinArea is extremely efficient in handling large circuits. For the same circuit with more than 180K gates that failed Minaret, our iMinArea algorithm solved the min-area retiming problem with 65MB memory in less than 1 minutes, which is at least 100 times faster than Minaret and uses at most $\frac{1}{30}$th of the memory used by Minaret.

The rest of this chapter is organized as follows. The retiming problems are introduced in Section 6.1. Our algorithmic idea is presented in Section 6.2. The iMinArea algorithm is proposed in Section 6.3. After experimental results are given in Section 6.4, Section 6.5 concludes the chapter.

6.1. Problem Formulation

As in Leiserson and Saxe [**69**], a synchronous sequential circuit is modeled by a directed graph $G = (V, E)$ whose vertices V represent combinational gates and whose edges E represent signals between vertices. Nonnegative gate delays are given as vertex weights

$d : V \rightarrow \mathbb{R}^*$ and the nonnegative numbers of FFs on the signals are given as edge weights $w : E \rightarrow \mathbb{N}$. A special host vertex, the edges from host to the primary inputs, and the edges from the primary outputs to host, are introduced into the graph to represent interfaces with the external environment. Given such a graph, the *min-area retiming problem* asks for an FF relocation $w' : E \rightarrow \mathbb{N}$ such that the total FF area in the circuit is minimum while it works under a given clock period ϕ.

Conventionally, to guarantee that w' is a relocation of w, a retiming is given by a vertex labeling $r : V \rightarrow \mathbb{Z}$ representing the number of FFs moved backward over each gate from fanouts to fanins. Given r, the FF number on the edge (u, v) after retiming is $w_r(u, v) = w(u, v) + r(v) - r(u)$. A retiming r is *valid* iff the FF number of every edge is nonnegative,

$$\mathrm{P0}(r): \qquad (\forall (u, v) \in E : w_r(u, v) \geq 0).$$

For a circuit to work under a given clock period ϕ, the maximum combinational path delay in the circuit can be at most ϕ. To compute the maximum path delay, we introduce a vertex labeling $t : V \rightarrow \mathbb{R}$ to represent the arrival time at the output of each gate. A valid retiming r is *feasible* for ϕ iff the following condition holds for some arrival times t,

$$\begin{aligned} \mathrm{P1}(r, \phi): \qquad & \big(\forall (u, v) \in E : (w_r(u, v) > 0) \vee (t(v) \geq d(v) + t(u))\big) \\ & \wedge \big(\forall v \in V : d(v) \leq t(v) \leq \phi\big). \end{aligned}$$

Note that the host vertex and the edges entering and leaving host should be ignored in the above condition.

The total FF number is $\sum_{e \in E} w_r(e)$. For any vertex $v \in V$, let $\mathrm{FI}(v)$ and $\mathrm{FO}(v)$ be the sets of the fanins and the fanouts of v respectively. To minimize the total FF number is equivalent to maximize the quantity $\sum_{v \in V}(|\mathrm{FO}(v)| - |\mathrm{FI}(v)|)r(v)$. More generally, let $b : V \to \mathbb{R}$ represent the reduction in FF area if one FF is moved from the fanouts of the given vertex to its fanins. The FF area reduction for the retiming r is $\sum_{v \in V} b(v)r(v)$. With these notations, the min-area retiming problem can be formally stated as follows.

Problem 6.1 (Min-Area Retiming)**.**

$$\begin{aligned} \textit{Maximize} \quad & \sum_{v \in V(G)} b(v)r(v), \\ \textit{s.t.} \quad & \mathrm{P0}(r) \wedge \mathrm{P1}(r, \phi). \end{aligned}$$

For ease of presentation, we extend b to any graph $X = (V_X, E_X)$ with $V_X \subseteq V$ and any $I \subseteq V$ by defining $b(X) \triangleq \sum_{v \in V_X} b(v)$, $b(I) \triangleq \sum_{v \in I} b(v)$, and $b(\emptyset) \triangleq 0$. We assume that $b(G) = 0$ without loss of generality and that the min-area retiming problem is bounded.

More complicated retiming problems can be solved in the same formulation of Problem 6.1. One example is to consider the sharing of the FFs at the fanouts of a gate. As proposed by Leiserson et al. [**69**], this scenario is handled by including additional constraints in $\mathrm{P0}(r)$ and setting the labeling b accordingly. Let $\mathrm{wmax}(u) = \max_{(u,v) \in E} w(u, v)$ and assume that all the fanout edges of u have the same breadth $\beta(u)$, which is the costs of adding a FF along each edge. For each vertex u where the FFs at the fanouts of u should be shared, a dummy vertex u_m is introduced. For each fanout v of u, the breadth

of the edge (u, v) is changed to $\frac{\beta(u)}{|\mathrm{FO}(u)|}$ and one constraint is added to P0(r) by adding the edge (v, u_m) to G with $w(v, u_m) = \mathrm{wmax}(u) - w(u, v)$ and the breadth $\frac{\beta(u)}{|\mathrm{FO}(u)|}$.

6.2. Algorithm Overview

In this section, we present the general idea behind our incremental algorithm without delving into technical details, which will be presented in the next section.

As a motivation and comparison to our algorithm, we first discuss Leiserson and Saxe's approach to the min-area retiming [**69**]. Two $n \times n$ matrices W and D are first computed to capture the critical timing constraints, and based on them, a dual of the min-cost network flow problem is formulated and solved. For any vertex pair (u, v), $W(u, v)$ is the minimum number of FFs along any path from u to v, and $D(u, v)$ is the maximum delay of the paths from u to v with $W(u, v)$ FFs. If $D(u, v) > \phi$, then there is a timing critical path from u and v and a critical timing constraint requiring at least 1 FF on the path should be generated. The dual of the min-cost network flow problem is formulated to maximize the FF area reduction subject to the nonnegative FF number requirement and all the critical timing constraints. As W and D would usually be much denser than the circuit graph, the flow network would be dense when the given clock period is tight. Despite the many efforts [**70, 71**] to reduce the storage requirement for computing the critical timing constraints and to prune the redundant constraints, the large number of constraints is still the bottleneck for solving the min-area retiming problems.

To totally avoid the bottleneck, our algorithm does not compute the matrices W or D at all. The feasibility of clock period ϕ is checked by dynamically updating the gate arrival times and comparing them with ϕ, as in [**69, 73**]. The objective in Problem 6.1

indicates that, in order to improve a given solution, some vertices with $b > 0$ must have their r increased. However, a vertex may not be independent: if $w_r(u, v) = 0$, increasing $r(u)$ requests increasing $r(v)$ at the same time. It is not hard to maintain such a relation. However, a more involved case happens when the increase of r over a path extends it to be longer than ϕ. Incremental arrival time updating can identify such a situation, and we will keep a relation between the source u and sink v of the violating path. It is revealing to note that we have $D(u, v) > \phi$ and $r(v) + W(u, v) - r(u) = 1$ for such u and v. In other words, *our algorithm dynamically identifies timing arcs in Leiserson and Saxe's flow network and only identifies the currently tight ones that "lie on the road to improvement".* The relations thus identified on normal circuit edges and on tight timing arcs are called *active constraints*. They will force vertices with $b > 0$ to be bundled with vertices with $b \leq 0$. When there is still a bundle I with $b(I) > 0$, the objective can be improved by increasing r on I; otherwise, the current retiming is already optimal.

Algorithmic Idea Incremental Min-Area Retiming	
1	Find a feasible retiming r under clock period ϕ.
2	$A \leftarrow \emptyset$.
3	**Loop**:
4	Find a positive vertex set I closed under A.
5	**If** no such I exists:
6	Stop, r is optimal.
7	**Else if** $w_r(u, v) = 0$ for an edge (u, v) leaving I:
8	Add (u, v) to A.
9	**Else if** $t(v) > \phi$ in r_I for some vertex v:
10	Add $(q(v), v)$ to A.
11	**Else**:
12	$r \leftarrow r_I$; update A.

Figure 6.1. Idea of incremental min-area retiming.

The flow of our algorithm is shown in Figure 6.1. A feasible retiming r for the clock period ϕ and a set of active constraints A are maintained throughout the algorithm. An initial feasible retiming r on line 1 may be obtained by any efficient fixed period retiming algorithms [**69, 73**]. We call a vertex set I *closed* under active constraints A if $\forall(u, v) \in A$, $u \in I \Rightarrow v \in I$. A vertex set I is *positive* if $b(I) > 0$, meaning that the increment of r on I will reduce the FF area. We use r_I to denote the new retiming after the increment. However, such an increase may violate the nonnegative FF number requirement on an edge leaving I – an active constraint is added in this case on line 8. Such an increase may also violate the timing constraint if a path longer than ϕ is created. We use $q(v)$ to record the source of the critical path to v. If $t(v) > \phi$ in r_I, an active constraint $(q(v), v)$ is added on line 10. If a positive I is found that will not generate more active constraints, the FF area can be reduced by increasing r on I, as on line 12. If, with the growth of active constraints, there is no positive I closed under A, then r is claimed optimal.

The above idea seems natural but the difficulty to realize it lies in how to effectively and efficiently maintain the active constraints A. Keeping every identified active constraint in A is not efficient since it might make $|A|$ very large. On the other hand, if not careful, removing some active constraints from A may not lead to algorithm convergence, since it is possible to have active constraints cycling in and out of A. We successfully overcome the difficulty by maintaining A as a *regular forest*, which is a forest with special properties. The details are presented in the next section; we only note here that $|A|$ is at most $n - 1$ while the termination of the algorithm is guaranteed. Similar ideas of maintaining constraints as a forest have been used in other works [**75, 76**]. However, a major contribution by our algorithm is to incrementally handle dynamically generated constraints in a

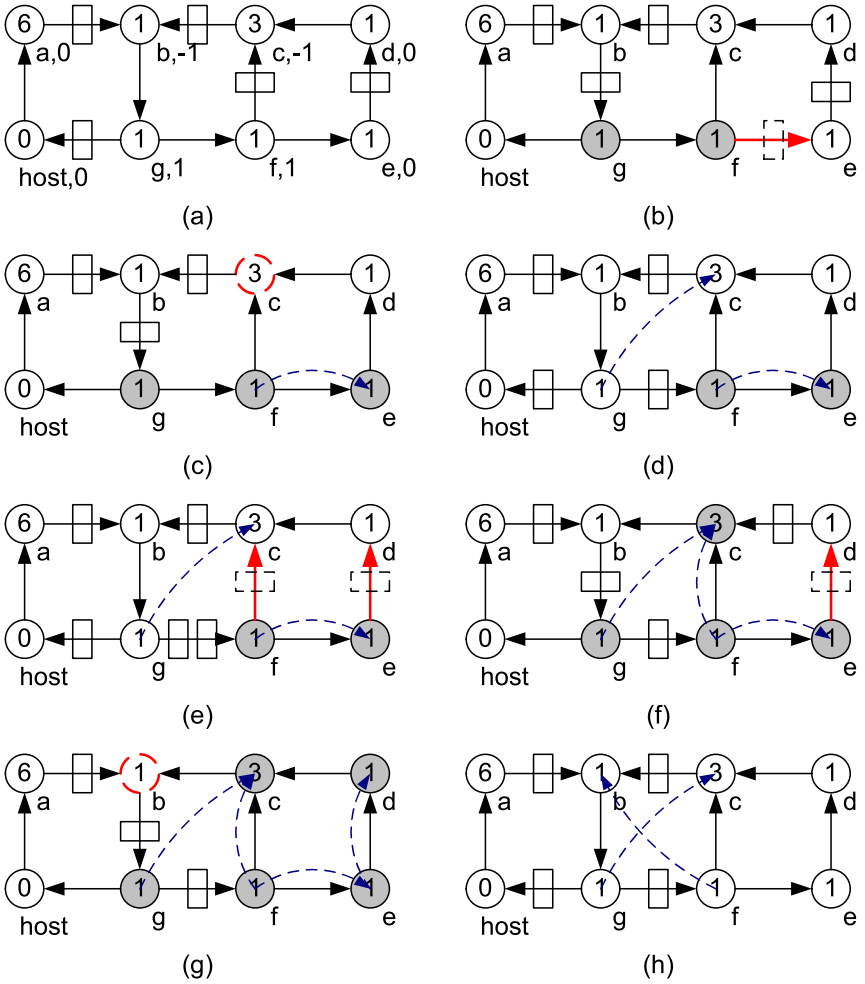

Figure 6.2. An example of our incremental min-area retiming algorithm. The labeling b is shown in (a) after gate names. Gate delays are inside each gate. The clock period is 6. Dotted arcs are active constraints. Exactly one FF is moved from the fanouts of the gray gates to their fanins.

regular forest, which can not be done by any existing algorithm. Therefore, our algorithm is much more efficient when it is expensive to generate all the constraints.

We use an example as illustrated in Figure 6.2 to further clarify our idea of incremental min-area retiming. Detailed executing information is listed in the following table.

	A, I	comments
1	$\emptyset$, $\{g, f\}$	$w_(f, e) + r(e) - r(f) = 0$.
2	$\{(f, e)\}$, $\{g, f, e\}$	$t(c) = 7 > 6$, $q(c) = g$.
3	$\{(f, e), (g, c)\}$, $\{f, e\}$	$r \leftarrow r_I$.
4	$\{(f, e), (g, c)\}$, $\{f, e\}$	$w(f, c) + r(c) - r(f) = 0$.
5	$\{(f, e), (g, c), (f, c)\}$, $\{f, e, g, c\}$	$w(e, d) + r(d) - r(e) = 0$.
6	$\{(f, e), (g, c), (f, c), (e, d)\}$, $\{f, e, g, c, d\}$	$t(b) = 7 > 6$, $q(b) = f$.
7	$\{(g, c), (f, b)\}$, No positive I	r is optimal.

6.3. Algorithm Description

6.3.1. Regular Forests

Consider a forest F with vertices V consisting of rooted trees. For any vertex $v \in V$, let T_v be the subtree rooted at v. For any non-root vertex $v \in V$, let p_v be its parent. A labeling $B : V \to \mathbb{R}$ is maintained such that $B(v) = b(T_v)$. For any non-root vertex $v \in V$, a direction is assigned to the edge $\{p_v, v\}$ such that an active constraint can be derived from the edge. A labeling $\mathrm{U}(v)$ is used to maintain the direction: if $\mathrm{U}(v) = \text{true}$, then (v, p_v) is the active constraint; if $\mathrm{U}(v) = \text{false}$, then (p_v, v) is the active constraint. Let $A(F)$ be the set of the active constraints derived from the edges of F. We define a tree T to be *regular* iff for any vertex v of T that is not the root of T, the following conditions hold, which are illustrated in Figure 6.3,

(1) if $b(T) > 0$, then $\big(\mathrm{U}(v) \wedge (B(v) > 0)\big) \vee \big(\neg\mathrm{U}(v) \wedge (B(v) \leq 0)\big)$;

(2) if $b(T) = 0$, then $\big(\mathrm{U}(v) \wedge (B(v) > 0)\big) \vee \big(\neg\mathrm{U}(v) \wedge (B(v) < 0)\big)$;

(3) if $b(T) < 0$, then $\big(\mathrm{U}(v) \wedge (B(v) \geq 0)\big) \vee \big(\neg\mathrm{U}(v) \wedge (B(v) < 0)\big)$.

We define a tree to be *almost* regular if the inequalities $B(v) < 0$ and $B(v) > 0$ in the above conditions are substituted with $B(v) \leq 0$ and $B(v) \geq 0$ respectively. We further define the forest to be *regular* if all the trees in the forest are regular.

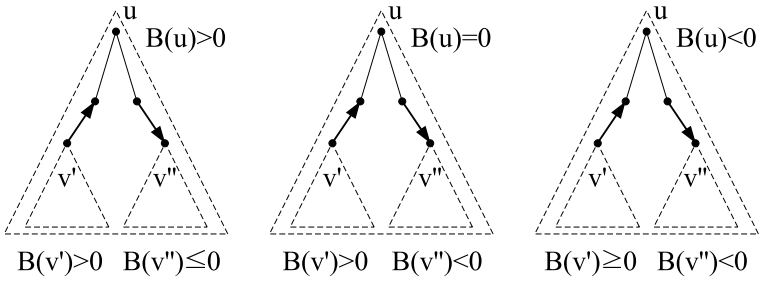

Figure 6.3. Regular trees.

We call a tree T *positive* (respectively *zero* and *negative*) iff $b(T) > 0$ (respectively $b(T) = 0$ and $b(T) < 0$). Let $P(F)$ (respectively $Z(F)$ and $N(F)$) be the set of all the positive trees (respectively zero trees and negative trees) in F. Let the vertices in $P(F)$ be $V_{P(F)}$. If $P(F) \neq \emptyset$, then $I = V_{P(F)}$ is positive and closed under $A(F)$. Actually, the following lemma states that such I is the one with the maximum $b(I)$.

Lemma 6.1. *Let I' be a vertex set that is closed under $A(F)$. Then $b(I') \leq b(V_{P(F)})$.*

Proof. Let T be an almost regular tree with vertices V_T. Let $A(T)$ be the set of the active constraints derived from the edges of T. We claim that for any vertex set $I \subseteq V_T$ closed under A(T),

$$b(V_T - I) \geq 0. \tag{6.1}$$

Consider the forest obtained by removing all the edges between $V_T - I$ and I in the tree T. Suppose there are $k \geq 1$ tree(s) in the forest. We prove the above claim by induction on k.

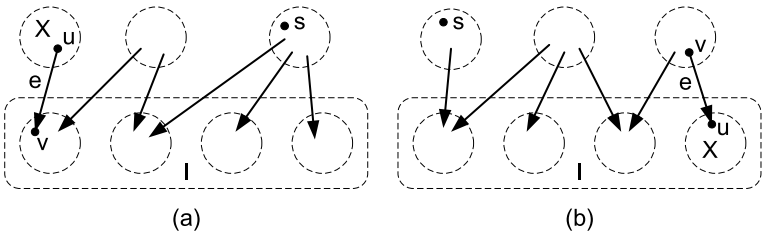

Figure 6.4. Proof of Lemma 6.1.

First of all, if $k = 1$ then either $I = \emptyset$ or $I = V_T$. It is obvious that $b(V_T - I) \geq 0$. Assume that the claim holds for any $1 \leq k \leq l$ for some positive integer l. Suppose I is closed under $A(T)$ and $k = l + 1$. Let s be the root of the tree T. Consider the forest obtained by removing all the edges between $V_T - I$ and I. Since $k \geq 2$, we can always identify a tree X with vertices V_X in the forest such that, first, $s \notin V_X$; second, there is exactly one edge e that connects a vertex $u \in V_X$ to a vertex $v \in V_T - V_X$. If (u, v) is an active constraint as shown in Figure 6.4 (a), then $V_X \subseteq V_T - I$. It is straightforward that $I \cup V_X$ is closed under $A(T)$. Therefore, applying the induction hypothesis, we must have $b\big(V_T - (I \cup V_X)\big) \geq 0$. Since $s \notin V_X$, $b(X) = B(u) \geq 0$. Thus,

$$b(V_T - I) = b\big(V_T - (I \cup V_X)\big) + b(X) \geq 0$$

On the other hand, if (v, u) is an active constraints as shown in Figure 6.4 (b), then $V_X \subseteq I$. It is straightforward that $I - V_X$ is closed under $A(T)$. Therefore, applying the induction hypothesis, we must have $b\big(V_T - (I - V_X)\big) \geq 0$. Since $s \notin V_X$, $b(X) = B(u) \leq 0$. Thus,

$$b(V_T - I) = b\big(V_T - (I - V_X)\big) - b(X) \geq 0$$

In summary, we always have $b(V_T - I) \geq 0$ for any I closed under $A(T)$ and $k = l + 1$. With induction, we proved the above claim.

Since any regular tree is an almost regular tree, according to Equation (6.1) and the definitions of the regular trees, we have that if T is a positive regular tree, then

$$b(V_T - I) \geq 0, \forall I \subseteq V_T \text{ closed under } A(T); \tag{6.2}$$

and that if T is a zero or negative regular tree, then

$$b(I) \leq 0, \forall I \subseteq V_T \text{ closed under } A(T). \tag{6.3}$$

For the regular forest F and the vertex set I' closed under $A(F)$, we have that $I' \cap V_T$ is closed under $A(T)$ for any tree T in F. Therefore, applying Equations (6.2) and (6.3), we have that,

$$\begin{aligned} b(I') &= \sum_{T \in F} b(I' \cap V_T) \\ &= \sum_{T \in P(F)} \Big(b(V_T) - b\big(V_T - (I' \cap V_T)\big) \Big) + \sum_{T \in Z(F) \cup N(F)} b(I' \cap V_T) \\ &\leq \sum_{T \in P(F)} b(V_T) = b(P(F)). \end{aligned}$$

□

On the other hand, If $P(F) = \emptyset$, we can claim optimality as in the following lemma,

Lemma 6.2. *Suppose that $A(F)$ is a set of the active constraints of a feasible retiming r for the clock period ϕ, i.e., $\forall (u, v) \in A(F)$, either $w_r(u, v) = 0$, or $(D(u, v) > \phi) \wedge (r(v) +$*

$W(u,v) - r(u) = 1$). *If $P(F) = \emptyset$, then r is the optimal solution of the min-area retiming problem.*

Proof. We define the labeling $\gamma : A(F) \to \mathbb{Z}$ as follows. If $(p_v, v) \in A(F)$ for some non-root vertex $v \in V$, then let $\gamma(p_v, v) = -B(v)$; if $(v, p_v) \in A(F)$ for some non-root vertex $v \in V$, then let $\gamma(v, p_v) = B(v)$. Recall that we assume $b(G) = 0$. Since $P(F) = \emptyset$, there is no negative tree in F and every tree is a zero tree. Thus we have $\forall (u,v) \in A(F)$, $\gamma(u,v) > 0$, and $\forall v \in V$,

$$b(v) = \sum_{(v,j) \in A(F)} \gamma(v,j) - \sum_{(i,v) \in A(F)} \gamma(i,v).$$

Suppose r' is a feasible retiming for ϕ. Since either $w_r(u,v) = 0$ or $(D(u,v) > \phi) \wedge (r(v) + W(u,v) - r(u) = 1)$, $\forall (u,v) \in A(F)$, we must have that,

$$r'(u) - r'(v) \leq r(u) - r(v), \forall (u,v) \in A(F).$$

Therefore,

$$\begin{aligned} \sum_{v \in V} b(v) r'(v) &= \sum_{(u,v) \in A(F)} \gamma(u,v)(r'(u) - r'(v)) \\ &\leq \sum_{(u,v) \in A(F)} \gamma(u,v)(r(u) - r(v)) \\ &= \sum_{v \in V} b(v) r(v). \end{aligned}$$

Thus r is optimal. □

We store the forest F in an adjacency list data structure using $O(n)$ storage. We assume that there are two operations that can be completed with $O(n)$ time and space:

the first one is CreateTree(F, v), which either removes the edge $\{p_v, v\}$ from the forest if v is not a root, or keep F unchanged if v is a root; the second one is MergeTree(F, u, v), which assumes that v is the root of a tree not containing u and makes u the parent of v. We design the subroutine ChangeRoot(F, v) as shown in Figure 6.5 to update the regular forest F in order to make v the root of a tree without introducing additional active constraints into $A(F)$. In each iteration of the **For** loop on line 2, v_{i-1} is the root of the tree containing v, v_i is a child of v_{i-1}, and v is in the subtree rooted at v_i. The subtree rooted at v_i is cut off from the tree rooted at v_{i-1} on line 4. In order to keep the vertices of $P(F)$, $Z(F)$, and $N(F)$ unchanged, we assign v_{i-1} to be a child of v_i on line 7 if necessary. The correctness of the ChangeRoot subroutine is stated in the following lemma.

Subroutine ChangeRoot

Inputs

F: a regular forest. v: a vertex.

Outputs

Updated F where v is the root of a tree.

```
1  Let (v_0, v_1, ..., v_l) with v_l = v be the path from the root of the tree
   containing v to v in F.
2  For i = 1 to l:
3    b_T ← B(v_{i-1}). B(v_{i-1}) ← B(v_{i-1}) − B(v_i).
4    CreateTree(F, v_i).
5    If (b_T > 0) ∧ U(v_i) ∧ (B(v_{i-1}) > 0)
       or (b_T < 0) ∧ ¬U(v_i) ∧ (B(v_{i-1}) < 0):
6      Continue.
7    MergeTree(F, v_i, v_{i-1}), B(v_i) ← b_T, U(v_{i-1}) ← ¬U(v_i).
```

Figure 6.5. The ChangeRoot subroutine.

Lemma 6.3. *The invariants of the* **For** *loop on line 2 are that, first, the regular forest F is regular; second, $A(F)$ contains no new active constraint; third, the vertices of $P(F)$,*

$Z(F)$, and $N(F)$ are not changed. When the subroutine terminates, v is the root of a tree in F.

Proof. We prove the invariants by induction. First, when we enter the loop with $i = 1$, obviously the invariants hold. Assume that the invariants hold when we enter the loop with $i = k$. When we leave the loop, if the condition on line 5 holds, then the edge $\{v_i, v_{i-1}\}$ is removed from F and one active constraint is removed from $A(F)$. It can be verified that F remains regular and the vertices of $P(F)$, $Z(F)$, and $N(F)$ are not changed. On the other hand, if the condition on line 5 does not hold, then $A(F)$ remains unchanged when we leave the loop. It can also be verified that the invariants hold. Therefore, the invariants will hold when we enter the loop with $i = k + 1$.

Since $v_l = v$, v is the root of a tree when the subroutine terminates. □

Subroutine UpdateForest	
Inputs	
F:a regular forest.	
u :a vertex belongs to $V_{P(F)}$.	
v :a vertex does not belong to $V_{P(F)}$.	
Outputs	
Updated regular forest F with (u, v) added to $A(F)$.	
1	ChangeRoot(F, v).
2	**If** $B(v) = 0$:
3	MergeTree(F, u, v), U$(v) \leftarrow$ false.
4	**Else**:// must have $B(v) < 0$
5	ChangeRoot(F, u).
6	$b_T \leftarrow B(u) + B(v)$.
7	MergeTree(F, u, v), $B(u) \leftarrow b_T$, U$(v) \leftarrow$ false.
8	ZeroCut(F, u, b_T).

Figure 6.6. The UpdateForest subroutine.

Let F_0 be the forest with no edge, i.e., every vertex is a tree in F_0. Then F_0 is regular and $A(F_0) = \emptyset$. Our algorithmic idea in Section 6.2 suggests to start with the forest F being F_0 and to satisfy $P(F) = \emptyset$ eventually with additional active constraints. Note that $b(P(F)) \geq 0$ always holds and $P(F) = \emptyset$ is equivalent to $b(P(F)) = 0$. Intuitively, either we combine a positive tree with a negative tree to reduce $b(P(F))$, or we combine a positive tree with a zero tree to expand $P(F)$ in order to reduce $b(P(F))$ later. We propose to capture such progress by a potential tuple,

$$\Psi(F) \triangleq (b(P(F)), n - |V_{P(F)}|),$$

with the lexicographic ordering, i.e., for $\Psi(F) = (x, y)$ and $\Psi(F') = (x', y')$, $\Psi(F) \leq \Psi(F')$ iff $x < x'$ or $(x = x') \wedge (y \leq y')$. Assuming that the additional active constraint is (u, v) satisfying $u \in V_{P(F)}$ and $v \notin V_{P(F)}$, we design the UpdateForest subroutine as shown in Figure 6.6 that will decrease $\Psi(F)$ by adding (u, v) to $A(F)$ and removing active constraints from $A(F)$ if necessary. Note that such active constraint must exist eventually as we move FFs from the fanouts of $V_{P(F)}$ to their fanins; otherwise the min-area retiming problem is unbounded. In this subroutine, we first simplify the problem by the ChangeRoot subroutine on line 1 in order to make v a root in the regular forest. If v is the root of a zero tree, we assign v to be a child of u on line 3. Otherwise, we further simplify the problem by ChangeRoot on line 5 in order to make u a root. Then we assign v to be a child of u on line 7. Since after line 7, the tree rooted at u will not always be regular but will always be almost regular, we call the ZeroCut subroutine on line 8 to recover F as a regular forest without increasing the potential tuple. The ZeroCut subroutine is shown in Figure 6.7, which recursively cuts off the subtrees that

do not satisfy the conditions of a regular tree. The correctness of the ZeroCut and the UpdateForest subroutines are stated in the following lemmas.

Lemma 6.4. *Assume that every tree in F is regular expect T which is almost regular. Let u be the root of T. Then after we apply ZeroCut$(F, u, b(T))$, F becomes regular and $b(P(F))$ remains unchanged.*

Proof. The ZeroCut subroutine modifies T only by removing edges from it. It will terminate since every node in T is visited at most once. As the subtrees violating the definition of the regular trees are removed on line 3, it can be verified that all the trees in the forest are regular when ZeroCut terminates. □

Lemma 6.5. *Assume that F is a regular forest, $u \in V_{P(F)}$, and $v \notin V_{P(F)}$. Then after we apply UpdateForest(F, u, v), F remains a regular forest, $\Psi(F)$ is strictly decreased, and (u, v) is the only active constraint added to $A(F)$.*

Proof. It is obvious that the UpdateForest will terminate since both the ChangeRoot and the ZeroCut subroutines terminate. Moreover, (u, v) is the only active constraint added to $A(F)$ since both the ChangeRoot and the ZeroCut subroutines modify the forest only by removing edges. If $B(v) = 0$ on line 2, then it can be verified that F remains a regular forest. Since $b(P(F))$ will not change but $|V_{P(F)}|$ is increased by at least 1, $\Psi(F)$ is strictly decreased. On the other hand, if $B(v) \neq 0$ on line 2, then we must have $B(v) < 0$ according to Lemma 6.3 and that $v \notin V_{P(F)}$. After line 7, every tree in F is regular expect the tree rooted at u which is almost regular. Therefore, according to Lemma 6.4, after line 8, F is regular. Since $b(P(F))$ will be decreased in this case, $\Psi(F)$ is strictly decreased. □

Subroutine ZeroCut

Inputs

F :a forest.
u :a vertex in an almost regular tree T.
b_T:equals to $b(T)$.

Outputs

Updated forest F where the subtree of T rooted at u becomes a set of regular trees.

```
1  For each child v of u:
2    If ¬U(v) ∧ (b_T ≤ 0) ∧ (B(v) = 0)
       or U(v) ∧ (b_T ≥ 0) ∧ (B(v) = 0):
3      CreateTree(F, v).
4      ZeroCut(F, v, 0).
5    Else:
6      ZeroCut(F, v, b_T).
```

Figure 6.7. The ZeroCut subroutine.

6.3.2. The iMinArea Algorithm

Combining the algorithmic idea in Section 6.2 and the regular forest data structure, we design the iMinArea algorithm that solves the min-area retiming problem incrementally and optimally as shown in Figure 6.8. The invariants of the loop on line 3 are stated in the following lemma.

Lemma 6.6. *At the beginning of each iteration of the loop on line 3, r is a feasible retiming for ϕ, F is a regular forest, $A(F)$ is the set of the active constraints of r.*

Proof. We prove the lemma by induction. For the first time we enter the loop, it is obvious that r is a feasible retiming for ϕ, F is a regular forest, $A(F)$ is the set of the active constraints of r. Assume that the above 3 conditions hold when we enter the loop. It is straightforward that r is feasible when we leave the loop. The forest F is regular

according to Lemma 6.5. According to the definition of the active constraints, $A(F)$ is the set of the active constraints of r if r is not replaced by r_I. We now only need to prove that $A(F)$ is the set of the active constraints of r_I when r_I is feasible. Because $I = V_{P(F)}$, we have that

$$(u \in I) \wedge (v \in I), \text{ or } (u \notin I) \wedge (v \notin I), \forall (u, v) \in A(F),$$

which implies that

$$r(u) - r(v) = r_I(u) - r_I(v), \forall (u, v) \in A(F).$$

On the other hand, we have that

$$w(u, v) - r(u) + r(v) = 0, \text{ or } (D(u, v) > \phi) \wedge (r(v) + W(u, v) - r(u) = 1), \forall (u, v) \in A(F).$$

Therefore,

$$w(u, v) - r_I(u) + r_I(v) = 0, \text{ or } (D(u, v) > \phi) \wedge (r_I(v) + W(u, v) - r_I(u) = 1), \forall (u, v) \in A(F).$$

Thus $A(F)$ is the set of the active constraints of r_I. □

The preconditions for line 9 and 10 are established by the following lemma.

Lemma 6.7. *If $w_r(u, v) \neq 0$ for any fanout edge (u, v) of I, then r_I is valid. If $t(v) > \phi$ in r_I for some vertex v on line 9, then $q(v) \in V_{P(F)}$ and $v \notin V_{P(F)}$.*

Proof. Since r is valid and $w_r(u, v) \neq 0$ for any fanout edge (u, v) of I, we must have

$$w(u, v) - r_I(u) + r_I(v) \geq w(u, v) - r(u) + r(v) \geq 0, \forall (u, v) \in E.$$

Thus r_I is valid.

Assume $t(v) > \phi$ in r_I for some vertex v on line 9. Let $u = q(v)$. Then $D(u, v) > \phi$ and $W(u, v) - r_I(u) + r_I(v) = 0$. Since r is feasible, we must have $W(u, v) - r(u) + r(v) \geq 1$. Therefore,

$$r_I(u) - r(u) \geq r_I(v) - r(v) + 1.$$

As both $r_I(u) - r(u)$ and $r_I(v) - r(v)$ are either 0 or 1, we have $r_I(u) = r(u) + 1$ and $r_I(v) = r(v)$, which imply that $u \in V_{P(F)}$ and $v \notin V_{P(F)}$. □

Algorithm iMinArea

Inputs

ϕ:the clock period.

Outputs

The optimal feasible retiming r for ϕ.

```
 1  Initialize a feasible retiming r for φ.
 2  Initialize F to be a forest with no edge.
 3  Loop:
 4    I ← V_{P(F)}.
 5    If I = ∅:
 6      Stop, r is optimal.
 7    Else if w_r(u, v) = 0 for an edge (u, v) leaving I:
 8      UpdateForest(F, u, v).
 9    Else if t(v) > φ in r_I for some vertex v:
10      UpdateForest(F, q(v), v).
11    Else:
12      r ← r_I.
```

Figure 6.8. The iMinArea algorithm.

We have the following theorem,

Theorem 6.1. *The iMinArea algorithm will terminate and when it terminates, r is an optimal solution of the min-area retiming problem.*

Proof. When the optimality is not claimed on line 6, either the FF area of r will be strictly decreased by $b(I) > 0$ for some $I \subset V$ and $\Psi(F)$ remains the same on line 12, or $\Psi(F)$ will be strictly decreased and the FF area of r remains the same on line 8 and 10 according to Lemma 6.5. Since the problem is bounded, the number of the subsets of V is finite, and the number of the regular forests with vertices V is finite, we can claim the termination of the iMinArea algorithm. According to Lemma 6.2 and 6.6, when the iMinArea algorithm terminates, r is an optimal solution of the min-area retiming problem. □

The iMinArea algorithm requires $O(m)$ storage for the circuit graph and $O(n)$ storage for the auxiliary data structures. The time complexity of each iteration of the loop on line 3 is $O(m)$. The number of iterations can be bounded for reasonable practical VLSI circuits as stated in the following theorem,

Theorem 6.2. *The space complexity of the iMinArea algorithm is $O(m)$. If we assume that the labeling b is integer-valued, that the FF area in the initial feasible retiming is bounded by $O(m)$, and that $b(P(F_0)) = \sum_{(v \in V) \wedge (b(v) > 0)} b(v)$ is bounded by $O(n)$, then the time complexity of the iMinArea algorithm is $O(n^2 m)$.*

Proof. It is straightforward that the space complexity of the iMinArea algorithm is $O(m)$. Since the labeling b is integer-valued, in each iteration, we decrease $b(P(F))$ by at least 1, or increase $|V_{P(F)}|$ by at least 1, or decrease FF area by at least 1. Because the time complexity of each iteration is $O(m)$, the FF area in the initial feasible retiming is bounded by $O(m)$, and $b(P(F_0))$ is bounded by $O(n)$, the time complexity is $O\big(m(m + n^2)\big) = O(n^2 m)$. □

Such time complexity is comparable to that of the min-period retiming problem, which is much simpler than the min-area retiming problem – the best theoretical time complexity for min-period retiming is $O(nm \log n)$ [**69**], while the worse-case runtime for the most efficient practical algorithm is $O(n^2 m)$ [**73**].

6.3.3. Implementation Details

Many details of the iMinArea algorithm are not specified, e.g. the order to check the individual constraints on line 7 and 9. We extend the incremental idea to the implementation of the algorithm. All the techniques introduced in this section will not affect the theoretical complexity but will effectively improve the practical runtime of the algorithm.

First of all, it is not necessary to generate I on line 4 from scratch every time. It can be proved that the UpdateForest subroutine changes $V_{P(F)}$ by either inserting vertices or removing vertices but not both. We modify the UpdateForest subroutine to provide such information so that we can construct I to be $V_{P(F)}$ incrementally. Let the inserted vertices or the removed vertices be Q. They will be used later when the constraints on line 7 and 9 are checked incrementally.

Secondly, it is not efficient to check every fanout edge of I, to compute the labelings t and q in r_I, and to check every vertex every time when the algorithm reaches line 7 and 9. The constraints should be checked incrementally, i.e. the constraints that are known to hold should be excluded from being checked, and the labelings t and q should be updated incrementally. We maintain two vertex queues J and K for such purpose. For any vertex $u \notin J$, if $u \in I$, then for any edge (u, v), either $v \in I$ or $w_r(u, v) > 0$. For any vertex $u \notin K$ and any vertex v in the combinational fanin cone of u (including u) in r_I, $t(v)$ and

$q(v)$ are up-to-date, and $t(v) \leq \phi$. On line 7, we repeatedly remove a vertex u from J until a edge (u, v) leaving I satisfying $w_r(u, v) = 0$ is found or J is empty. On line 9, we repeatedly remove a vertex and its combinational fanin cone from K, compute $t(v)$ and $q(v)$ for any vertex v in the cone, until $t(v) > \phi$ for some vertex v or K is empty. The queues J and K are updated incrementally when I is changed. When I is changed by inserting the vertex set Q, it is sufficient to insert every vertex in Q to J and to insert every vertex in the combinational fanout cone of Q in r_I to K. Computing the cone could be time consuming when $|Q|$ is large. In such case we insert every vertex of G to K. When I is changed by removing the vertex set Q, it is sufficient to insert to J the vertices $u \in I$ that fanouts to a vertex $v \in Q$ satisfying $w_r(u, v) = 0$. Identifying such vertices could be inefficient when $|Q|$ is large. In such case we insert every vertex of G to J. For the queue K, we insert every vertex of G to it.

If the sharing of the FFs at the fanouts of gates is considered, we introduce redundant constraints to P0. Let u be any vertex with the dummy vertex u_m and let v be a fanout of u. In P0(r), we should have $w(u, v) + r(v) - r(u) \geq 0$ and $\mathrm{wmax}(u) - w(u, v) + r(u_m) - r(v) \geq 0$. Thus, $\mathrm{wmax}(u) + r(u_m) - r(u) \geq 0$. This redundant constraint is inserted to P0 and is checked first on line 7 after we remove u from the vertex queue J. The effect is that when both (u, v) and (v, u_m) are active constraints, we directly identify (u, u_m) as an active constraint and thus include u and u_m in one regular tree without requiring a detour to v. As $b(u) > 0$ and $b(u) + b(u_m) = 0$ for most u, $b(P(F))$ is reduced more frequently without the necessity to expand $P(F)$ first and the algorithm runs faster.

6.4. Experimental Results

We implement the iMinArea algorithm in C++. We obtain the code of Minaret [**71**] for comparison and obtain the code of the incremental min-period retiming algorithm [**73**] to compute the minimum clock period and the initial feasible retiming in the iMinArea algorithm. All the codes are compiled by GCC version 3.4 and run on a Linux workstation with dual 2.4GHz Intel Xeon processors and 2GB memory.

We perform experiments with three benchmark suites: the first one are the conventional ISCAS89 sequential circuits; the second one are the large circuits (myex1 through myex5) created by the authors of Minaret from combining ISCAS89 circuits; the third one are the ITC'99 sequential circuits [**77**], among which there are a few even larger circuits. We follow Minaret [**71**] to assume unit FF area and unit gate delay. Note that this is only for the purpose of comparison and our iMinArea algorithm can handle arbitrary FF areas and gate delays. For each circuit, we first apply Zhou's min-period retiming algorithm [**73**] to obtain the minimum clock period. Then we use both the iMinArea algorithm and Minaret to minimize the number of FFs, subject to the minimum clock period and considering the sharing of the FFs at the fanouts of gates.

The experimental results of the largest 26 circuits among all the benchmark suites are reported in Table 6.1. The iMinArea algorithm solved the problem for all the other circuits in less than 0.1 second and thus the results of them are excluded from being reported here. The statistics of the circuits are reported in the columns "$|V|$" and "$|E|$". For each circuit and each algorithm, we report the number of FFs in the columns "# FFs" and the runtime in seconds in the columns "t_{mp}(s)", "t_{mr}(s)", and "t_{ima}(s)" respectively. The number of FFs obtained by Minaret and that obtained by iMinArea are the same as expected, which

is significantly less than that obtained by min-period retiming ignoring the FF area. The minimum clock period is reported in the column "ϕ". The size of the flow network is tremendous in Minaret, as indicated by the number of arcs reported in the column "# arcs". The iMinArea algorithm can find the optimal feasible retiming after only a few incremental changes according to the column "# R", which shows the number of the feasible retimings found during optimization. The speed-up of the iMinArea algorithm in comparison to Minaret is reported in the column "$\frac{t_{\text{ima}}}{t_{\text{mr}}}$". Our iMinArea algorithm is much more efficient than Minaret with a speed-up of up to more than 100× and more than 60× in total for all the circuits. We do not report the detailed memory usage but mention that while Minaret used up all the 2GB virtual memory on our machine for the circuits "b19(std)" and "b19-1(std)", our iMinArea algorithm required at most 65MB memory as for the circuit "b19(std)".

We are also curious about iMinArea's performance for a degenerated special case: the *unconstrained* min-area retiming problem without clock period constraints. This problem is conventionally solved as a dual min-cost network flow problem on the circuit graph. We simply turn off line 1, 9, and 10 in iMinArea. Experimental results show that our algorithm is 4× faster than the conventional approach using the min-cost network flow solver CS2 version 4.3 [**50**]. We also compare our algorithm to Hurst et al. [**78**], which is an efficient algorithm specially designed only for the problem with unit FF area and implemented in ABC [**79**]. Surprisingly, even though iMinArea is designed for a much more general problem, it is only 3× slower than Hurst et al. on the special problem.

Table 6.1. Results comparison between Minaret and iMinArea.

	statistics		Zhou's Min-Period [**73**]			Minaret [**71**]			iMinArea			$\frac{t_{mr}}{t_{ima}}$
name	$\|V\|$	$\|E\|$	# FFs	ϕ	t_{mp}(s)	# FFs	# arcs	t_{mr}(s)	# FFs	# R	t_{ima}(s)	
s13207.1	8005	11295	629	51	0.01	446	38630	1.37	446	6	0.54	2.5
s15850.1	9844	13792	565	63	0.03	525	38318	6.10	525	4	0.96	6.4
s35932	16421	28944	1729	27	0.01	1729	53087	1.67	1729	1	0.58	2.9
s38584.1	19596	33402	1428	48	0.08	1427	97268	13.79	1427	2	2.32	5.9
s38417	21505	31261	1619	32	0.02	1370	1507162	21.52	1370	5	1.57	13.7
myex1	26114	42576	2578	42	0.11	2293	812872	25.76	2293	4	3.72	6.9
myex2	29135	46890	4545	45	0.17	2022	398697	38.91	2022	5	5.74	6.8
myex3	35712	62018	3912	35	0.10	3279	5693689	88.20	3279	2	12.32	7.2
myex4	40994	64527	3831	35	0.11	2803	2635127	83.62	2803	6	4.61	18.2
myex5	50533	77939	6292	47	0.27	3369	3563555	151.43	3369	7	13.38	11.3
b14(opt)	5331	11523	1021	27	0.02	402	896045	7.65	402	3	0.63	12.2
b14-1(opt)	4042	8750	810	24	0.01	423	511627	3.48	423	2	0.64	5.4
b15(opt)	5945	11599	736	35	0.03	segmentation fault			417	2	0.33	-
b15-1(opt)	6514	13303	1006	23	0.01	579	1468197	10.23	579	3	1.45	7.1
b17(opt)	17228	32503	1982	35	0.09	1249	10981826	110.89	1249	4	2.17	51.0
b17-1(opt)	18813	36819	3016	24	0.06	1640	6712930	62.59	1640	4	8.01	7.8
b18(opt)	58141	115232	5319	59	0.65	2800	12189334	514.82	2800	4	33.36	15.4
b18-1(opt)	57985	116401	6295	50	0.80	3055	8875292	331.63	3055	3	55.76	5.9
b19(std)	183558	328866	9537	107	1.80	-	122286738	>7200	5113	4	57.72	>120
b19-1(std)	173823	310840	9428	107	1.72	-	112595516	>7200	5108	4	49.36	>120
b20(opt)	9952	19257	1360	40	0.04	453	179307	18.75	453	2	0.91	20.6
b20-1(opt)	8392	15854	1103	39	0.03	444	108695	12.19	444	2	1.05	11.6
b21(opt)	9861	18148	1146	43	0.04	368	879762	28.78	368	2	0.56	51.2
b21-1(opt)	8007	14961	1066	38	0.03	424	121706	12.04	424	2	0.88	13.6
b22(opt)	14493	28009	1753	39	0.15	702	236192	23.26	702	3	3.13	7.4
b22-1(opt)	12955	25346	1711	42	0.12	707	612839	30.62	707	2	1.90	16.1
total			74417		6.48			>16K	43147		263.6	>60

6.5. Summary

In this chapter, we presented an efficient algorithm named iMinArea to solve the min-area retiming algorithm incrementally and optimally. Instead of attacking the problem by formulating and solving a min-cost network flow problem in a dense flow network, our iMinArea algorithm generates the critical timing constraints dynamically, maintains active constraints in a forest, and retimes a circuit incrementally. Experimental results confirmed that our algorithm is significantly faster and uses much less memory in comparison to the best existing approach.

Even though only backward retiming is discussed in iMinArea, forward retiming can be symmetrically handled and mixed with backward retiming. Therefore, initial state can be easily enforced in the retimed circuits [**80, 81**]. Moreover, similar to [**82**] for min-period retiming, if the hold conditions need to be satisfied, iMinArea can be extended to solve the problem optimally.

We should also note that, since it is incremental, iMinArea can be stopped any time when a designer is satisfied (with the area) or impatient (with the runtime). However, we have not (yet) found it necessary to do so.

CHAPTER 7

Risk Aversion Min-Period Retiming under Process Variations

Process variations have become a critical issue in VLSI fabrication that the designers must face with aggressive scaling down of VLSI feature sizes. Because of the increasing variations, chip characteristics, e.g. the clock period and the power consumption, fall into larger intervals instead of being single values or within narrow ranges. As statistical analysis approaches enable the designer to analysis the variations, statistical optimization algorithms will finally equip the designers with the necessary tools to control such stochastic effects in order to improve chip yield and system reliability.

A review of the recent advances in statistical timing analysis can be found in the paper [**83**]. In summary, state-of-the-art SSTA algorithms can achieve a good balance in terms of accuracy and efficiency to compute the arrival times and the clock period under variations by extending the sum and maximum operations to random variables, which represent the random gate delays and are usually in the canonical form of a linear combination of independent Gaussian random variables. How to further advance the current SSTA techniques to handle other aspects of the circuits and how to apply SSTA for efficient statistical optimization remain the challenge problems for SSTA researchers.

Conventional circuit optimizations have been extended to address the issue of variability through statistical optimization. For example, for the gate sizing problems, numerous approaches have been proposed, e.g.[**84, 85, 86**]. In the work [**84**], the Lagrangian relaxation based sizing technique is extended to consider variations by introducing a safety

margin to the circuit timing, which is dynamically changed according to path delay variations. In the work [**85**], the iterative improving gate sizing heuristic is extended to handle a yield-aware objective function for library-based gate sizing problems. In the work [**86**], a statistical gate sizing problem is formulated based on conventional two-stage stochastic programming problems with fixed recourse [**87**]. As the problem is shown to be convex, the authors apply Kelley's Cutting Plane algorithm for an optimal solution. With those successes, it is natural to ask if such approaches can provide insights into future SSTA researches and can be extended to other deterministic optimizations.

Among the many deterministic optimization techniques, retiming [**69**] is one of the most powerful sequential transformations that relocates the flip-flops (FFs) in a circuit without changing the circuit functionality. Intuitively, one can apply retiming to improve the timing yield of a circuit under process variations since relocating the FFs could balance the combinational paths. Such idea was previously explored by Wang and Zhou [**1**]. In that work, the authors proposed a heuristic algorithm by combining SSTA with a deterministic min-period retiming algorithm [**73**]. However, there is little theoretical guarantee that such heuristic would result in a good retiming solution. In this chapter, we study the statistical retiming problem with a more sound theoretical basis and propose a new heuristic algorithm to optimize the circuit for better clock period distribution under process variations. Our contributions in this chapter include:

(1) We formulate the risk aversion min-period retiming problem for statistical retiming optimization. Our formulation is based on conventional two-stage stochastic programming problems with fixed recourse. A coherent measure of risk called conditional value-at-risk [**88**] is used as the objective function to be minimized.

We prove that the proposed problem is an integer convex programming problem by presenting a continuous convex relaxation of it.

(2) We derive an analytical formula for the subgradient of the objective function based on the continuous relaxation. Compared to the previous works [**85, 86**] where the subgradients are computed through perturbing the circuits and evaluate the changes statistically for multiple times, our approach is much more efficient. We can use any random gate delay model through sampling from a black box representing the underlying variation model. Moreover, we show how current SSTA techniques can be improved to further speed-up the subgradient computation and thus the statistical retiming optimizations.

(3) We extend the concept of timing critical paths, which is essential for deterministic retiming optimizations [**69, 73, 9**], to a statistical sense. We propose a practical simplification of the concept such that it can be efficiently integrated into our algorithm for the risk aversion min-period retiming problem.

(4) We propose the Incremental Risk Aversion Retiming algorithm to solve the risk aversion min-period retiming problem heuristically. Guided by the subgradient and the statistical timing critical paths, our algorithm iteratively improves a retiming solution. The subproblem solved in each iteration is identified as an incremental min-area retiming problem and is solved through the incremental min-area retiming algorithm iMinArea presented in Chapter 6.

The rest of this chapter is organized as follows. The retiming problems, the two-stage stochastic programming problems, and the coherent measure of risk are introduced in Section 7.1. We formulate the risk aversion min-period retiming problem in Section 7.2

and show that it is a integer convex programming problem in Section 7.3. We propose our Incremental Risk Aversion Retiming algorithm in Section 7.4. After experimental results are given in Section 7.5, Section 7.6 concludes the chapter.

7.1. Preliminaries

7.1.1. Deterministic Retiming Problems

For retiming, a synchronous sequential circuit is modeled by a directed graph $G = (V, E)$ as in Leiserson and Saxe [**69**]. The vertices V represent combinational gates and the edges E represent signals between vertices. The gate delays are given as the nonnegative vertex weights $d : V \rightarrow \mathbb{R}^*$. The numbers of FFs on the signals are given as the nonnegative edge weights $w : E \rightarrow \mathbb{N}$.

To guarantee that the circuit functionality will be preserved after FF relocation, a retiming is given by a vertex labeling $r : V \rightarrow \mathbb{Z}$, which represents the number of FFs moved backward over each gate from its fanouts to its fanins. The FF number on the edge (u, v) after retiming is $w_r(u, v) = w(u, v) + r(v) - r(u)$. The retiming r is *valid* iff the FF number of every edge remains nonnegative,

$$w_r(u, v) \geq 0, \forall (u, v) \in E. \tag{7.1}$$

For a given valid retiming r, the retimed circuit works under a given clock period ϕ iff the maximum combinational path delay in the circuit is at most ϕ. In such case, the retiming r is called *feasible* for ϕ. To compute the maximum path delay, arrival times $t : V \rightarrow \mathbb{R}$ are introduced at the outputs of all the gates. The following constraints should

be satisfied.

$$w_r(u,v)=0 \Rightarrow t(v) \geq d(v)+t(u), \forall (u,v) \in E, \tag{7.2}$$

$$d(v) \leq t(v) \leq \phi, \forall v \in V.$$

Conventionally, two retiming problems can be formulated given the above definitions. The minimum period (min-period) retiming problem asks for a minimum clock period such that there exists a feasible retiming for it; the minimum area (min-area) retiming problem asks for a feasible retiming for a given clock period to minimize the total FF area. To solve both the min-period and the min-area retiming problems, it would be helpful to investigate the path-based critical timing constraints that can be integrated into a mathematical programming problem. A timing critical path is a directed path connecting two vertices satisfying the conditions that the total number of FFs along the path is the minimum among all the paths with the same endpoints, and the total delay along the path exceeds the desired clock period. For any pair of vertices, if there exists a timing critical path connecting them, a critical timing constraint will require at least 1 FF along the path after retiming. A valid retiming will be feasible for the desired clock period iff all the critical timing constraints are satisfied. However, because it is usually expensive and sometimes prohibitive to generate all the critical timing constraints, practically efficient algorithms [**73, 9**] are able to identify those critical timing constraints from Equation (7.2) only when they are required and to organize them into proper data structure in order to guide the optimization and to assert optimality with a low storage overhead.

7.1.2. Two-Stage Stochastic Programming Problem with Fixed Recourse and Coherent Measure of Risk

A decision problem whose output depends not only on the decision itself but also some uncertain parameters not available at the time of decision making is usually formulated as a two-stage stochastic programming problem with fixed recourse [**87, 89, 90, 91**]. In such programming problems, the uncertain parameters unknown at the time of decision are modeled as random variables. Note that the level of the accuracy that one could know about the distribution of the random variables would definitely limit one's ability to perform optimizations for such programming problems but that is out of the scope of this chapter. The programming problem is separated into two stages as suggested by its name. In the first stage, a decision is made and will incur an initial cost. In the second stage, the uncertain parameters are realized and a second stage cost is determined from both the decision and the realized uncertain parameters through a known deterministic programming problem, i.e. the fixed recourse. The objective of the programming problem is to make a decision in the first stage to minimize the "total cost" of the two stages – as the outcome is random, such cost may have many possible interpretations.

Let X be the random variable representing the outcome of the two-stage stochastic programming problem. An interpretation can be formalized by introducing a measure of risk $M[X]$ that maps X into a real number. As we are interested in minimization problems, usually a random variable mapped to a smaller value is better than the ones mapped to larger values. For example, a family of the most popular measures are the following ones involving the mean and standard deviations of the random variable X,

$$M_\gamma[X] \triangleq \mathrm{E}\big[X\big] + \gamma\sqrt{\mathrm{E}\big[(X - \mathrm{E}\big[X\big])^2\big]}.$$

On the other hand, if X represents the random clock period of a circuit under process variations, given a target clock period ϕ, one can measure X by the timing yield, which is the probability that X is no larger than ϕ, i.e.,

$$\mathrm{Yield}_\phi[X] \triangleq P(X \leq \phi).$$

However, as pointed out by Rockafellar [**88**], the above measures are not favorable objectives for optimizations because they are not *coherent*. A coherent measure of risk should satisfy a few conditions as follows.

Definition 7.1 (Coherent Measure of Risk [**88**])**.** *A measure of risk M is a coherent measure of risk in the basic sense if,*

1. $M[C] = C$ *for all constants* C.
2. $M[(1-\lambda)X + \lambda Y] \leq (1-\lambda)M[X] + \lambda M[Y]$ *for* $\lambda \in [0,1]$.
3. $M[X] \leq M[Y]$ *if* $P(X \leq Y) = 1$.
4. $M[X] \leq 0$ *when there exists a infinite sequence of random variables* X_k *such that* $\lim_{k\to+\infty} \mathrm{E}\big[(X_k - X)^2\big] = 0$ *and* $M[X_k] \leq 0$.
5. $M[\lambda X] = \lambda M[X]$ *for* $\lambda > 0$.

Intuitively, the first condition indicates that if a deterministic value is treated as a random variable taking a single value, the measure should interpret it by the deterministic value; the second condition requires that the measure to be convex with respect to the random variables; the third condition ensures the measure to be monotonic, i.e., if one

random variable is no smaller than the other with probability 1, the measure of the former should be no smaller than that of the latter; the fourth condition guarantees that if a random variable can be approximated by some other random variables, one will accept it when all the approximations are acceptable; and the fifth condition implies that the measure is insensitive to scaling.

Coherent measures of risk do exist. For example, the expectation $\mathrm{E}\big[X\big]$ of the random variable X is a coherent measure of risk, though it is feeble and cannot capture the risk associated with X. A more interesting coherent measure of risk, as proposed by Rockafellar [**88**], is the *conditional value-at-risk* that measures the risk in a random variable beyond a risk aversion level α. Generally speaking, the risk aversion level α is similar to the concept of yield in VLSI designs. For a risk aversion level α, a measure of risk $\mathrm{VaR}_\alpha[X]$ called *value-at-risk* is first defined as the value satisfying the following condition,

$$P(X \leq \mathrm{VaR}_\alpha[X]) = \alpha.$$

Intuitively, the value-at-risk measure can be treated as the inverse of the timing yield measure: while the timing yield measure computes the risk aversion level (the timing yield) from a given value (the target clock period), the value-at-risk measure computes the value at a given risk aversion level. The value-at-risk measure is not coherent, the conditional value-at-risk measure, which is coherent, is defined as follows based on value-at-risk.

$$\mathrm{CVaR}_\alpha[X] \triangleq \mathrm{E}\big[X|X > \mathrm{VaR}_\alpha[X]\big].$$

For any $x \in \mathbb{R}$, let x^+ denote $\max\{0, x\}$. One can prove that,

$$\mathrm{CVaR}_\alpha[X] = \mathrm{VaR}_\alpha[X] + \frac{1}{1-\alpha}\mathrm{E}\big[(X - \mathrm{VaR}_\alpha[X])^+\big]. \tag{7.3}$$

Therefore, if the conditional value-at-risk should be minimized, it has the advantage to optimize both the value-at-risk and the tail beyond the value-at-risk.

7.2. Problem Formulation

Under process variations, the delays of the gates in the circuit are no longer deterministic but random variables. Let Ω be the probabilistic space representing process variations. For a particular variation $\omega \in \Omega$, assume that the random gate delays are realized as the deterministic nonnegative vertex weights $d_\omega : V \to \mathbb{R}^*$. For a valid retiming r, let the minimum clock period for the retimed circuit under the variation ω be $\phi_\omega(r)$. According to Section 7.1.2, we formulate the following *risk aversion min-period retiming* problem as a two-stage stochastic programming problem with fixed recourse.

Problem 7.1 (Risk Aversion Min-Period Retiming)**.** *Given a risk aversion level of α, find an integer-valued vertex labeling r for the following programming problem:*

$$\begin{aligned} \textit{Minimize} \quad & \mathrm{CVaR}_\alpha[\phi_\omega(r)] \\ \textit{s.t.} \quad & w_r(u,v) \geq 0, \forall (u,v) \in E, \end{aligned}$$

where for every variation ω belonging to the probabilistic space Ω, $\phi_\omega(r)$ is the minimum objective of the following programming problem,

$$
\begin{aligned}
\textit{Minimize} \quad & \phi \\
\textit{s.t.} \quad & w_r(u,v) = 0 \Rightarrow t(v) \geq d_\omega(v) + t(u), \forall (u,v) \in E, \\
& d_\omega(v) \leq t(v) \leq \phi, \forall v \in V.
\end{aligned}
$$

It is clear that in the first stage of the risk aversion min-period retiming problem, a valid retiming will be chosen with an initial cost of 0. In the second stage, when the random gate delays are realized as d_ω, the minimum clock period is computed through the fixed recourse by solving the second stage programming problem. The second stage cost is the coherent risk aversion measure of the minimum clock period.

7.3. A Convex Relaxation

Note that the proposed risk aversion min-period retiming problem is difficult, not only because r should be integer-valued, but also because the second stage problem is not a mathematical programming problem. To overcome such difficulty, we propose to relax the problem before attempting to solve it.

7.3.1. Continuous Relaxation Formulation

The second stage of the risk aversion min-period retiming problem is not a mathematical programming problem. Consider an arbitrary simple path p from u to v, i.e. a path without cycles, in the circuit graph G. Let the total number of FFs along the path be $w(p)$. Let the total path delay be $d_\omega(p)$ for a particular $\omega \in \Omega$. We have the following lemma

that transforms the second stage problem into a mathematical programming problem by enumerating paths.

Lemma 7.1. *For a valid retiming r, the minimum clock period for a particular $\omega \in \Omega$ can be computed as that,*

$$\phi_\omega(r) = \max_{\textit{simple path } p \textit{ in } G} \frac{d_\omega(p)}{w_r(p) + 1}. \tag{7.4}$$

Proof. For a valid retiming r, for any simple path p from u to v, the minimum clock period $\phi_\omega(r)$ should satisfy that,

$$d_\omega(p) \leq \phi_\omega(r)(w_r(p) + 1),$$

where $w_r(p) = w(p) + r(v) - r(u)$ is the total number of FFs along the path p in the retimed circuit. Moreover, because r is valid, we should have that,

$$w_r(p) \geq 0.$$

On the other hand, there must exist a combinational path p^* in the retimed circuit with the maximum combinational path delay. For such path p^*, it must satisfy that,

$$w_r(p^*) = 0, \text{ and } d_\omega(p^*) = \phi_\omega(r).$$

Therefore, Equation (7.4) holds. □

Note that although the second stage problem as formulated in Lemma 7.1 is a mathematical programming problem, its size is exponential in terms of the size of the circuit

graph G, while the size of the second stage programming problem as formulated in Problem 7.1 is linear. As the mathematical programming problem formulation will be only applied to theoretical analysis, its size will not be a concern for practical implementations.

Based on Equation (7.4), we can relax the requirement that r should be integer-valued by extending $\phi_\omega(r)$ to real-valued r. First, we define a real-valued r to be *valid* iff $w_r(u,v) \geq 0$ holds for every edge $(u,v) \in E$. Then for any simple path p from u to v in G, it remains true that $w_r(p) \geq 0$. Therefore, for any valid real-valued r, we can define $\phi_\omega(r)$ using the same equation as Equation (7.4). In summary, we have the following continuous relaxation of the risk aversion min-period retiming problem.

Problem 7.2. *Given a risk aversion level of α, find a real-valued vertex labeling r for the following programming problem:*

$$\begin{aligned} \textit{Minimize} \quad & \mathrm{CVaR}_\alpha[\phi_\omega(r)] \\ s.t. \quad & w_r(u,v) \geq 0, \forall (u,v) \in E, \end{aligned}$$

where for every variation ω belonging to the probabilistic space Ω,

$$\phi_\omega(r) = \max_{\textit{simple path } p \textit{ in } G} \frac{d_\omega(p)}{w_r(p)+1}.$$

7.3.2. Convexity of Formulation

Let r be valid and real-valued. For a particular $\omega \in \Omega$, assume that for the simple path p_ω from u_ω to v_ω, we have that,

$$\phi_\omega(r) = \frac{d_\omega(p_\omega)}{w_r(p_\omega)+1}.$$

An vertex labeling $s_\omega : V \to \{-1, 0, 1\}$ can be introduced to identify u_ω and v_ω. Let $s_\omega(u_\omega) = 1$, $s_\omega(v_\omega) = -1$, and $s_\omega(x) = 0$ for any other $x \in V$. A very important property of Problem 7.2 is that it is a convex programming problem as stated in the following lemma.

Lemma 7.2. $\mathrm{CVaR}_\alpha[\phi_\omega(r)]$ *is a convex function of* r *for all real-valued valid* r*. For a particular valid* r*, define* $g_r : V \to \mathbb{R}$ *as that,*

$$g_r(u) \triangleq \frac{1}{1-\alpha}\mathrm{E}\big[I_\omega(r)\frac{\phi_\omega(r)s_\omega(u)}{w_r(p_\omega)+1}\big]. \tag{7.5}$$

Then g_r *is a subgradient of* $\mathrm{CVaR}_\alpha[\phi_\omega(r)]$*.*

Proof. Let r' be valid and real-valued. We should have,

$$\phi_\omega(r') \geq \frac{d_\omega(p_\omega)}{w_{r'}(p_\omega)+1}.$$

Therefore,

$$\begin{aligned}
\phi_\omega(r') - \phi_\omega(r) &\geq \frac{d_\omega(p_\omega)}{w_{r'}(p_\omega)+1} - \frac{d_\omega(p_\omega)}{w_r(p_\omega)+1} \\
&= \phi_\omega(r)\Big(\frac{w_r(p_\omega)+1}{w_{r'}(p_\omega)+1} - 1\Big) \\
&= \phi_\omega(r)\frac{w_r(p_\omega) - w_{r'}(p_\omega)}{w_{r'}(p_\omega)+1} \\
&= \phi_\omega(r)\Big(\frac{w_r(p_\omega) - w_{r'}(p_\omega)}{w_r(p_\omega)+1} + \frac{\big(w_r(p_\omega) - w_{r'}(p_\omega)\big)^2}{w_{r'}(p_\omega)+1}\Big) \\
&\geq \phi_\omega(r)\frac{w_r(p_\omega) - w_{r'}(p_\omega)}{w_r(p_\omega)+1} \\
&= \frac{\phi_\omega(r)}{w_r(p_\omega)+1}\Big(\big(r'(u_\omega) - r(u_\omega)\big) - \big(r'(v_\omega) - r(v_\omega)\big)\Big).
\end{aligned}$$

Intuitively, the above inequalities can be explained as follows: if the minimum clock period $\phi_\omega(r)$ should be decreased, then one should introduce more FFs onto the path p_ω by either incrementing $r(v_\omega)$ or decrementing $r(u_\omega)$; otherwise, if FFs are removed from the path p_ω by either decrementing $r(v_\omega)$ or incrementing $r(u_\omega)$, the minimum clock period would increase. By using the labeling s_ω, the above inequality can be rewritten as that,

$$\phi_\omega(r') - \phi_\omega(r) \geq \sum_{u \in V} \frac{\phi_\omega(r) s_\omega(u)}{w_r(p_\omega) + 1} \big(r'(u) - r(u)\big). \tag{7.6}$$

Given a risk aversion level α, let $I_\omega(r)$ be 1 if $\phi_\omega(r) \geq \mathrm{VaR}_\alpha[\phi_\omega(r)]$ and 0 otherwise. For ease of presentation, denote $\mathrm{VaR}_\alpha[\phi_\omega(r)]$ by A and $\mathrm{VaR}_\alpha[\phi_\omega(r')]$ by A$'$. Then, we have that,

$$\begin{aligned}
\mathrm{E}\big[I_\omega(r)\big] &= P(\phi_\omega(r) \geq \mathrm{A}) = 1 - \alpha, && (7.7)\\
(\phi_\omega(r) - \mathrm{A})^+ &= (\phi_\omega(r) - \mathrm{A}) I_\omega(r), && (7.8)\\
(\phi_\omega(r') - \mathrm{A}')^+ &\geq (\phi_\omega(r') - \mathrm{A}') I_\omega(r). && (7.9)
\end{aligned}$$

Thus,

$$\begin{aligned}
&\mathrm{CVaR}_\alpha[\phi_\omega(r')] - \mathrm{CVaR}_\alpha[\phi_\omega(r)] \\
&\qquad \big(\text{from (7.3)}\big) \\
&= \Big(\mathrm{A}' + \frac{\mathrm{E}\big[(\phi_\omega(r') - \mathrm{A}')^+\big]}{1 - \alpha}\Big) - \Big(\mathrm{A} + \frac{\mathrm{E}\big[(\phi_\omega(r) - \mathrm{A})^+\big]}{1 - \alpha}\Big) \\
&\qquad \big(\text{from (7.8) and (7.9)}\big) \\
&\geq (\mathrm{A}' - \mathrm{A}) + \frac{\mathrm{E}\big[\big((\phi_\omega(r') - \mathrm{A}') - (\phi_\omega(r) - \mathrm{A})\big) I_\omega(r)\big]}{1 - \alpha}
\end{aligned}$$

$$
\begin{aligned}
&= (\mathrm{A}' - \mathrm{A})\Big(1 - \frac{\mathrm{E}\big[I_\omega(r)\big]}{1-\alpha}\Big) + \frac{\mathrm{E}\big[\big(\phi_\omega(r') - \phi_\omega(r)\big)I_\omega(r)\big]}{1-\alpha} \\
&\qquad\qquad \big(\text{from (7.7)}\big) \\
&= \frac{\mathrm{E}\big[\big(\phi_\omega(r') - \phi_\omega(r)\big)I_\omega(r)\big]}{1-\alpha} \\
&\qquad\qquad \big(\text{from (7.6)}\big) \\
&\geq \frac{1}{1-\alpha}\mathrm{E}\big[I_\omega(r)\sum_{u\in V}\frac{\phi_\omega(r)s_\omega(u)}{w_r(p_\omega)+1}\big(r'(u) - r(u)\big)\big] \\
&= \sum_{u\in V}\frac{r'(u) - r(u)}{1-\alpha}\mathrm{E}\big[I_\omega(r)\frac{\phi_\omega(r)s_\omega(u)}{w_r(p_\omega)+1}\big] \\
&= \sum_{u\in V} g_r(u)\big(r'(u) - r(u)\big)
\end{aligned}
$$

Therefore, $\mathrm{CVaR}_\alpha[\phi_\omega(r)]$ is a convex function of r and g_r is a subgradient. □

It is straightforward that the set of all the valid real-valued r is convex. We have the following theorem according to Lemma 7.2.

Theorem 7.1. *Problem 7.2 is a convex programming problem. The optimal solution to the risk aversion min-period retiming problem is an integer optimal solution to Problem 7.2.*

Proof. It is implied by Lemma 7.2. □

7.4. Incremental Algorithm for Risk Aversion Min-Period Retiming

We have shown in Section 7.3 that the risk aversion min-period retiming is an integer convex programming problem and derived a subgradient of the objective function. Intuitively, such subgradient can be used to guide iterative heuristic searches. In this section,

we will first show the method to compute the subgradient in practice and then present a heuristic algorithm based on the idea of incremental retiming.

7.4.1. Computing Subgradient from Black Box Model

According to Lemma 7.2, the subgradient of the objective function can be computed as Equation (7.5). Obviously, how to compute such subgradient in practice depends on how the probability space Ω and the random gate delays are specified. There are two typical models: in the first model, the joint distribution of the gate delays is explicitly given; in the second model, one can only obtain knowledge of the distribution by drawing independent samples from a black box. In this chapter, we are interested in the latter black box model because the black box model is independent of the underlying distribution and thus our proposed algorithm can handle arbitrary variation models. Moreover, since the subgradient will be computed from each sample drawn from the block box model, established deterministic analysis frameworks can be reused. One may be concerned about the efficiency of the algorithms relying on the black box model because of the multiple samplings. However, since the subgradient is used to guide the optimization, absolute accuracy is not necessary and a limited number of samples will be suffice for effective optimizations. Note that in case of the former model where the distribution is explicitly given, it would be helpful if current SSTA techniques can be extended to compute the subgradient according to Equation (7.5) efficiently and accurately. Such extensions are out of the scope of this chapter and are left as one of the future directions of SSTA.

Since the risk aversion min-period retiming problem requires an integer solution of Problem 7.2, we maintain an integer solution through our algorithm. Therefore, only

Subroutine ComputeSubgrad

Inputs

G:the circuit graph.
r :a valid retiming.
α :the risk aversion level.
N:the number of samples.

Outputs

An approximation $\hat{g}_r$ of the subgradient g_r.

1 Draw the samples ω_i for $i = 1, 2, \ldots, N$.
2 **For** $i = 1$ to N:
3 Compute the minimum clock period ϕ_{ω_i} and the arrival times t by Equation (7.2); use $q(v)$ to record the source of the critical path to the vertex v.
4 Identify the sink v_i of a critical path whose delay is ϕ_{ω_i}. The source of the path $u_i \leftarrow q(v_i)$.
5 $A \leftarrow$ the maximum value such that $|\{i : \phi_{\omega_i} < A\}| \leq \alpha N$.
6 $\hat{g}_r(v) \leftarrow 0, \forall v \in V$.
7 **For** $i = 1$ to N:
8 **If** $\phi_{\omega_i} \geq A$:
9 $\hat{g}_r(u_i) \leftarrow \hat{g}_r(u_i) + \frac{\phi_{\omega_i}}{N(1-\alpha)}$. $\hat{g}_r(v_i) \leftarrow \hat{g}_r(v_i) - \frac{\phi_{\omega_i}}{N(1-\alpha)}$.

Figure 7.1. The ComputeSubgrad subroutine.

the subgradient for at integer solutions should be computed. Let r be a valid retiming. Equation (7.5) suggests that the subgradient can be approximated by taking the average of the corresponding values from the individual samples. Suppose that N samples, ω_i, $i = 1, 2, \ldots, N$, are independently drawn from the black box. We design the ComputeSubgrad subroutine as shown in Figure 7.1 to obtained an approximation $\hat{g}_r$ of g_r by averaging the samples. In this algorithm, after the samples are drawn, we perform timing analysis on line 3 for each sample to determine the minimum clock period and the arrival times according to Equation (7.2). As the same time, we maintain a vertex labeling $q(v)$ to record the source of the critical path to the vertex v. Then, the combinational path with the maximum path delay is identified implicitly on line 4. The endpoints of the path are

u_i and v_i. Note that the path delay should be ϕ_{ω_i} and there is no FF along the path in the retimed circuit. An approximation of $\mathrm{VaR}_\alpha[\phi_\omega]$ is obtained on line 5. Finally, in the loop on line 7, we compute an approximation $\hat{g}_r$ of the subgradient g_r by averaging $\frac{1}{1-\alpha} I_{\omega_i}(r)\phi_{\omega_i}(r)s_{\omega_i}(u)$ for each vertex u according to Equation (7.5).

Note that many previous statistical optimization works, e.g. [**85, 86**], employed a different approach to approximate such subgradient. For a decision variable, the previous approaches will first perturb the variable and then approximate the subgradient of this variable by the change of the objective function under such perturbation. Such approach incurs large runtime overhead because, first, although for some decision variables, perturbation will not change the objective function and thus they can be excluded from the above computation, the number of the decision variables that the above computation must be applied to will increase as the circuit size increases; second, evaluating the objective function usually requires expensive SSTA algorithms. On the other hand, our analytical formula for the subgradient, as in Equation (7.5), allows us to compute the subgradient comparably efficiently via sampling from a black box model. Moreover, as mentioned before, the efficiency of our approach can be further improved with future relevant SSTA researches.

7.4.2. Statistical Timing Critical Paths

As we can approximately compute the subgradient via the ComputeSubgrad subroutine, an intuitive idea for optimization is to iteratively improve a valid retiming following the subgradient. Suppose r is a valid retiming. One can improve r to another valid retiming

r' by solving the following problem.

$$\begin{aligned} \text{Minimize} \quad & \sum_{v \in V} \hat{g}_r(v)\big(r'(v) - r(v)\big) \\ \text{s.t.} \quad & w_{r'}(u,v) \geq 0, \forall (u,v) \in E, \\ & 0 \leq r'(v) - r(v) \leq 1. \end{aligned} \tag{7.10}$$

In this problem, the objective function is an first-order approximation of $\text{CVaR}_\alpha[\phi_\omega(r')]$ obtained from the subgradient approximation $\hat{g}_r$. As the first-order approximation would become inaccurate when r' is faraway from r, we require the difference between r' and r to be at most 1. Moreover, because the constraints are a system of difference inequalities, this problem can be solved by network-flow techniques and there always exists an integer-valued optimal solution. Therefore, it is not necessary to round a non-integer solution to an integer one for a valid retiming.

However, this intuitive idea does not perform well in practice. The reason is that even changing $r(v)$ by 1 for some vertex v will result in unexpected changes in the minimum clock period and thus will make the first-order approximation inaccurate. Cutting plane techniques, similar to the statistical gate sizing work [**86**], can be applied to form a more accurate approximation based on the subgradients computed in the previous iterations. However, such techniques no longer guarantee the existence of an integer-valued optimal solution and may require a heuristic to round a non-integer optimal solution. Therefore, they cannot be applied directly to our retiming problem.

We propose to overcome such difficulty by introducing the concept of *statistical timing critical path*s. These paths can be treated as a natural extension of the deterministic

timing critical paths as mentioned in Section 7.1.1 to the statistical sense. Let r be the current valid retiming. Consider a simple path p in G. For any variation $\omega \in \Omega$, let $d_\omega(p)$ be the path delay. We define the statistical timing critical paths and state their property in the following lemma.

Lemma 7.3. *Given a constant C, we define a simple path p to be a statistical timing critical path if* $\mathrm{CVaR}_\alpha[d_\omega(p)] > C$. *For any valid retiming r satisfying* $\mathrm{CVaR}_\alpha[\phi_\omega(r)] \leq C$, *we must have that $w_r(p) \geq 1$.*

Proof. We prove the lemma by contradiction. Assume $w_r(p) < 1$. Then since r is valid, we must have $w_r(p) = 0$. Then $\phi_\omega(r) \geq d_\omega(p)$ for any variation $\omega \in \Omega$ since p is a combinational path. Thus we should have

$$\mathrm{CVaR}_\alpha[\phi_\omega(r)] \geq \mathrm{CVaR}_\alpha[d_\omega(p)] > C,$$

which violates the assumption that $\mathrm{CVaR}_\alpha[\phi_\omega(r)] \leq C$. □

Based on Lemma 7.3, we can augment the formulation in Equation (7.10) by the following constraints without affecting the optimality.

$$w_{r'}(p) \geq 1, \forall \text{ simple path } p \text{ satisfying } \mathrm{CVaR}_\alpha[d_\omega(p)] > \mathrm{CVaR}_\alpha[\phi_\omega(r)]. \tag{7.11}$$

Note that the constraints in Equation (7.11) have the same structure as those in Equation (7.10), i.e., they are a system of difference inequalities. Therefore, the existence of an integer-valued optimal solution is still guaranteed.

7.4.3. Incremental Risk Aversion Retiming Algorithm

One difficulty of the constraints in Equation (7.11) is that since the risk measure should be computed for many simple paths, it could be inefficient in practice. We propose to simplify the computation in our implementation by identifying similar paths through deterministic timing analysis. Let $\overline{d}(v) = \mathrm{E}\big[d_\omega(v)\big]$ be the nominal delay for each gate v. For a simple path p, let $\overline{d}(p)$ be the nominal path delay with respect to the nominal gate delays $\overline{d}$. For a given valid retiming r, let $\overline{\phi}(r)$ be the nominal minimum clock period, i.e., the minimum clock period with respect to $\overline{d}$. Then, we assume a simple path to be a statistical timing critical path if $\overline{d}(p) > \beta\overline{\phi}(r)$, where $\beta \geq 1$ is a parameter specified by the designer. In summary, given a valid retiming r, we propose to solve the following incremental retiming problem to obtain another valid retiming r' in order to improve the conditional value-at-risk measure of risk.

Problem 7.3.

$$
\begin{aligned}
\textit{Minimize} \quad & \sum_{v \in V} \hat{g}_r(v)\big(r'(v) - r(v)\big) \\
\textit{s.t.} \quad & w_{r'}(u,v) \geq 0, \forall (u,v) \in E, \\
& w_{r'}(p) \geq 1, \forall \textit{ simple path } p \textit{ satisfying } \overline{d}(p) > \beta\overline{\phi}(r), \\
& 0 \leq r'(v) - r(v) \leq 1.
\end{aligned}
$$

In Problem 7.3, since path enumeration is required to construct the constraints, the number of the constraints can be quadratic in terms of the number of the vertices, i.e. $\Theta(|V|^2)$. This may impose huge storage and runtime overhead if we are going to solve

Subroutine IncreRetime

Inputs

G:the circuit graph.
$\overline{d}$:the nominal gate delay.
$\overline{r}$:a valid retiming.
$\hat{g}_r$:the approximation of the subgradient.
β :a designer specified parameter.

Outputs

The optimal solution r' of Problem 7.3.

```
1  Compute φ̄(r) as the nominal minimum clock period of r. φ ← βφ̄(r).
2  Initialize F to be a regular forest with no edge with respect to −ĝ_r.
3  Loop:
4    I ← vertices of the positive trees in F.
5    If I = ∅:
6      r' ← r. Return.
7    If w_r(u, v) = 0 for an edge (u, v) leaving I:
8      Update F with (u, v). Continue the loop.
9    Construct a retiming r_I by moving 1 FF from the fanouts of I to
     their fanins in r.
10   Compute the arrival times t and the sources of the critical path q
     for each vertex v in r_I.
11   If t(v) > φ in r_I for some vertex v:
12     Update F with (q(v), v). Continue the loop.
13   r' ← r_I. Return.
```

Figure 7.2. The IncreRetime subroutine.

Problem 7.3 directly. However, we can treat Problem 7.3 as a special min-area retiming problem and apply the incremental min-area retiming algorithm iMinArea presented in Chapter 6 to solve it. The iMinArea requires only $O(|V|)$ storage on top of the circuit graph G and is efficient in practice. Let $\hat{g}_r(v)$ represents the increase of FF area when 1 FF is moved from the fanouts of v to its fanins. It is straightforward that the given valid retiming r is feasible for the clock period $\beta\overline{\phi}(r)$ with respect to the nominal gate delays. Then, Problem 7.3 actually asks for a set of vertices I such that the retiming r', which is obtained by moving 1 FF from the fanouts of I to its fanins, is a feasible

retiming for the clock period $\beta\overline{\phi}(r)$ with the minimum FF area. Because only 1 FF is allowed to move, it is not necessary to run the iMinArea algorithm until it finishes. We adapt the iMinArea algorithm presented in Chapter 6 in our IncreRetime subroutine as shown in Figure 7.2 to solve Problem 7.3. The following lemma states the correctness of the IncreRetime subroutine.

Lemma 7.4. *The IncreRetime subroutine terminates and when it terminate, it returns an optimal solution of Problem 7.3.*

Proof. According to Chapter 6, Theorem 6.1 implies that the IncreRetime subroutine will terminate and Lemma 6.1 implies that it will return an optimal solution of Problem 7.3. □

Based on the above discussions, we design the *Incremental Risk Aversion Retiming* algorithm as shown in Figure 7.3 to solve the risk aversion min-period retiming problem. In this algorithm, from a given initial valid retiming, we iteratively improve the current solution by first computing a subgradient on line 6 and then moving to the next solution on line 7 via solving Problem 7.3. The iteration will stop when a current retiming cannot be improved as found on line 9, or a maximum number R of iterations have been performed. The retiming solution with the best conditional value-at-risk measure of risk will be picked at the end of the algorithm.

7.5. Experiments

We obtain the code of the deterministic incremental min-period retiming algorithm [**73**] and build a risk-aware deterministic approach for comparison with our Incremental Risk

Algorithm Incremental Risk Aversion Retiming

Inputs

G:the circuit graph.
r :an initial valid retiming.
α :the risk aversion level.
R:the maximum number of iterations.

Outputs

The retiming r^* with the best $\text{CVaR}_\alpha[\phi_\omega(r^*)]$.

```
1  r* ← r.
2  For k = 1 to R:
3    Compute CVaR_α[φ_ω(r)] by sampling.
4    If CVaR_α[φ_ω(r)] < CVaR_α[φ_ω(r*)]:
5      r* ← r.
6    Compute ĝ_r by ComputeSubgrad.
7    Solve Problem 7.3 for r' by IncreRetime.
8    If r' = r:
9      Stop.
10   Else:
11     r ← r'.
```

Figure 7.3. The Incremental Risk Aversion Retiming algorithm.

Aversion Retiming algorithm. In this approach, we first assign each gate a deterministic delay derived from the gate delay distribution and a parameter γ specified by the designer. Then we run Zhou's algorithm [**73**] for a min-period retiming to obtain a solution. For a gate v, the deterministic gate delay is that

$$\mathrm{E}\big[d_\omega(v)\big] + \gamma\sqrt{\mathrm{E}\big[(d_\omega(v) - \mathrm{E}\big[d_\omega(v)\big])^2\big]},$$

i.e., a weighted summation of the nominal delay and the standard deviation. Note that this deterministic approach is similar to the "Alternative Algorithm" as proposed in the work [**1**].

We implement our Incremental Risk Aversion Retiming algorithm in C++. All the codes are compiled by GCC version 3.4 and run on a Linux workstation with dual 927MHz Intel Pentium III processors and 512MB memory.

We derive our experimental benchmarks from the conventional ISCAS89 sequential circuits. To establish a gate delay model for process variations, we assume a joint Gaussian distribution of the gate delay. The parameters of the distribution are determined as follows. First, we assign each gate a nominal delay proportional to the number of its fanouts and a standard deviation that is within 20% to 30% of the nominal value. Then, assuming that each gate has a dimension of 1×1, we perform a wire-length driven placement of the circuits using the placement tool mPL6 [**92**]. After placement, the chip area is divided into a 4×4 grid. Two gate delays are assumed to be perfectly correlated if they are within a same grid block, i.e., the covariance is 1. Otherwise, the covariance of two gate delays is assigned to be inversely proportional to the distance of the centers of the grid blocks that the two gates belong to.

We assume a risk aversion level of $\alpha = 0.9$. For each benchmark, we first perform three deterministic optimizations with the parameter $\gamma = 0$, 1, and 3 and obtain three solutions. Then we run our Incremental Risk Aversion Retiming algorithm with the initial retiming being the solution obtained from the above deterministic optimizations with $\gamma = 1$. Our algorithm is allowed to run for at most 50 iterations before one solution is obtained. The other parameters are $N = 500$ and $\beta = 1.01$. The conditional value-at-risk measure of the clock period for each solution is evaluated by performing Monte Carlo analysis for 10000 samples to ensure accuracy. The results are reported in Table 7.1 as follows. The statistics of the circuits are reported in the columns "$|V|$" and "$|E|$". Under the column

"Deterministic Approach", we report the conditional value-at-risk measure of the clock period for the original circuit before retiming in column "init" and report those of the three solutions obtained by the deterministic optimizations in the columns "$\gamma = 0$", "$\gamma = 1$", and "$\gamma = 3$". The column "best" shows the best one from the previous 4 columns, which is the best solution that one can get through the deterministic approach. The runtimes of the deterministic approach are all within 1 seconds and are thus excluded from being reported here. The results from our algorithm is reported under the column "Ours". The conditional value-at-risk measure of the clock period is reported in the column "CVaR". The improvement in percentage compared to the one in the column "best" is reported in the column "impr.". The number of the iterations performed is reported in the column "# R" and the runtime in seconds is reported in the column "t(s)". Note that for most of the benchmark, computing the subgradient uses more than 90% of the runtime. It can be seen from the table that our algorithm improves the solution quality for almost every benchmark circuit for up to 8% within fair amount of runtimes.

In addition, we compare the solutions in terms of the timing yield and report the results in Table 7.2. The target clock periods are determined such that the solution obtained by our algorithm will have a timing yield of 90%. This table shows that the timing yield can be effectively improved by optimizing the conditional value-at-risk measure.

Table 7.1. Results comparison between deterministic approach and our algorithm.

	Statistics		Deterministic Approach					Ours			
name	$\|V\|$	$\|E\|$	init	$\gamma = 0$	$\gamma = 1$	$\gamma = 3$	best	CVaR	impr.	# R	t(s)
s27	11	19	13.29	13.29	13.12	13.12	13.12	13.12	0.00%	1	0.0
s208.1	105	182	26.78	23.22	23.22	23.22	23.22	23.15	0.28%	2	0.1
s298	120	250	35.03	28.31	28.31	28.50	28.31	28.16	0.55%	50	1.2
s382	159	312	48.97	32.27	32.27	32.27	32.27	31.85	1.32%	5	0.2
s386	160	354	57.08	53.59	53.59	53.59	53.59	53.59	0.00%	50	1.6
s344	161	280	47.15	33.28	32.69	32.69	32.69	32.24	1.38%	50	1.5
s349	162	284	46.97	33.08	32.62	32.62	32.62	32.40	0.68%	50	1.5
s400	165	326	50.23	34.23	34.23	34.01	34.01	33.24	2.26%	7	0.2
s420.1	219	384	38.69	27.17	27.17	27.17	27.17	27.03	0.53%	3	0.2
s444	182	358	51.87	34.34	34.02	34.73	34.02	33.55	1.38%	10	0.4
s510	212	431	50.54	49.33	49.53	49.53	49.33	49.33	0.00%	50	2.1
s526	194	451	49.61	35.15	34.87	34.87	34.87	33.48	3.99%	50	2.0
s641	380	563	154.35	154.35	154.35	154.35	154.35	153.40	0.62%	8	0.6
s713	394	614	167.47	167.47	167.47	167.47	167.47	167.32	0.09%	5	0.4
s820	290	776	133.84	133.08	133.08	133.84	133.08	132.94	0.10%	50	3.1
s832	288	788	134.14	130.73	130.73	130.73	130.73	130.73	0.00%	50	3.1
s838.1	447	788	63.28	41.79	41.39	42.45	41.39	41.16	0.56%	50	4.1
s953	396	766	61.12	52.17	51.98	51.98	51.98	51.98	0.00%	50	3.8
s1196	530	1023	66.75	65.95	65.95	65.95	65.95	65.88	0.11%	2	0.3
s1238	509	1055	70.67	70.67	70.67	70.67	70.67	70.66	0.01%	2	0.3
s1423	658	1169	211.37	160.50	160.50	160.50	160.50	158.91	0.99%	12	1.5
s1488	654	1406	190.86	161.60	161.60	161.60	161.60	161.60	0.00%	50	6.7
s1494	648	1412	196.48	174.32	174.32	174.32	174.32	174.32	0.00%	50	6.6
s5378	2780	4261	66.64	66.64	66.64	66.64	66.64	64.05	3.89%	50	29.4
s9234.1	5598	4604	114.28	114.28	115.76	115.76	114.28	113.63	0.57%	50	50.6
s13207.1	7952	11082	193.40	123.08	122.62	123.10	122.62	118.85	3.08%	50	120.8
s15850.1	9773	13566	243.37	120.56	119.73	119.51	119.51	109.88	8.06%	50	156.0
s35932	16066	28589	187.44	170.62	172.38	177.34	170.62	171.84	-0.71%	50	265.3
s38417	22180	31127	173.83	97.46	96.62	97.18	96.62	93.69	3.03%	50	457.3
s38584.1	19254	33060	267.32	245.73	247.31	246.22	245.73	242.41	1.35%	50	445.3

Table 7.2. Results comparison in terms of timing yield.

	Deterministic Approach				Ours
name	init	$\gamma = 0$	$\gamma = 1$	$\gamma = 3$	
s27	88.3%	88.3%	90.0%	90.0%	90.0%
s208.1	62.9%	89.6%	89.6%	89.6%	90.0%
s298	48.5%	89.2%	89.2%	88.7%	90.0%
s382	9.6%	88.0%	88.0%	88.0%	90.0%
s386	80.6%	90.0%	90.0%	90.0%	90.0%
s344	17.6%	84.8%	87.8%	87.8%	90.0%
s349	17.7%	86.2%	88.6%	88.6%	90.0%
s400	10.9%	84.8%	84.8%	85.8%	90.0%
s420.1	13.1%	89.4%	89.4%	89.4%	90.0%
s444	5.8%	86.3%	87.5%	84.9%	90.0%
s510	86.8%	90.0%	89.5%	89.5%	90.0%
s526	17.6%	82.3%	83.9%	83.9%	90.0%
s641	89.1%	89.1%	89.1%	89.1%	90.0%
s713	89.9%	89.9%	89.9%	89.9%	90.0%
s820	88.7%	89.8%	89.8%	88.7%	90.0%
s832	86.8%	90.0%	90.0%	90.0%	90.0%
s838.1	5.7%	87.9%	89.3%	84.5%	90.0%
s953	66.6%	89.5%	90.0%	90.0%	90.0%
s1196	88.2%	89.8%	89.8%	89.8%	90.0%
s1238	90.0%	90.0%	90.0%	90.0%	90.0%
s1423	22.3%	88.4%	88.4%	88.4%	90.0%
s1488	63.0%	90.0%	90.0%	90.0%	90.0%
s1494	71.2%	90.0%	90.0%	90.0%	90.0%
s5378	82.5%	82.5%	82.5%	82.5%	90.0%
s9234.1	90.6%	90.6%	90.0%	90.0%	90.0%
s13207.1	82.8%	88.9%	88.9%	88.9%	90.0%
s15850.1	46.5%	87.2%	87.5%	87.5%	90.0%
s35932	74.2%	91.5%	89.5%	84.7%	90.0%
s38417	0.2%	83.1%	84.3%	82.2%	90.0%
s38584.1	85.3%	90.0%	90.0%	90.0%	90.0%

7.6. Summary

In this chapter, we formulated the risk aversion min-period retiming problem to optimize the clock period of a circuit under process variations. The formulation is based on conventional two-stage stochastic programming problem with fixed recourse with a risk aversion objective. We proved that the proposed problem is an integer convex programming problem. We gave an analytical formula for the subgradient of the objective function and proposed to compute an approximation of the subgradient by sampling from a black box. We presented a heuristic incremental algorithm to solve the proposed problem. and the effectiveness of our proposed approach is confirmed by the experimental results.

CHAPTER 8

Optimal Jumper Insertion for Antenna Avoidance Considering Antenna Charge Sharing

Antenna effect is a phenomenon in VLSI fabrication where current caused by plasma process flows through gate oxides and damages them. It reduces both manufacturing yield and product reliability, which are among the most important issues with the rapid scaling-down of VLSI feature sizes. The relationship between the amount of damage and the antenna ratio has been studied for a long time [**93, 94, 95**]. Generally speaking, a wire segment acting as antenna may collect charging current when exposed to plasma. If the segment only connects to gate oxides but not diffusions, a voltage potential may build up and a discharging current tunneling through the gate oxides will form under the potential and damage the gate oxides. The damage can be observed via the drift of threshold voltage; the spatial variations of plasma stress across the wafer may cause variations in threshold voltages with spatial correlations. Besides the processing parameters, which are constant, the antenna ratio of total exposed antenna area to total gate oxide area determines the amount of voltage potential and thus the damage: smaller antenna ratio results in less damage. Antenna rules are commonly enforced as upper-bounds on the antenna ratio in design rules [**96**].

Correcting antenna problem after placement and routing stage is feasible and effective [**97, 98, 99**]: if a wire segment violates the antenna rule, either jumpers are inserted to

break the wire segment or a protection diode is added to the unused area of the chip and connected to the wire segment to form a low impedance discharging path. However, as indicated by the works [**100, 101**], correction after placement and routing may not be effective or even possible beyond the $0.13\mu m$ technology. The main reason is that with the scaling-down of the VLSI feature sizes, antenna rules become more stringent. The number of wire segments violating the rules and thus the numbers of the jumpers and the protection diodes increase dramatically, which is not practical considering the limitation of the unused chip area and the routing resource. So, considering antenna effect in an earlier stage and planning ahead is a must to avoid the problems.

Two recent works [**102, 103**] considered the antenna effect during the routing stage by combining jumper insertion to layer assignment. In these approaches, jumper insertion guides the layer assignment algorithms by predicting the jumper positions. Assigning a wire segment to a layer has the same effect of inserting a jumper if a discharging path to the source is formed. If for some reason, e.g. limited routing resource, such layer assignment is not feasible, a jumper must be inserted. Two more works [**104, 105**] focused on the jumper insertion problem itself. The latter one provided an exact algorithm to solve it in general routing trees with obstacles. In the works [**102, 104, 105**], a random discharging model was used and the upper-bound on the total exposed antenna area connected to gate oxides was controlled. In [**103**], although the same bound was controlled, the author pointed out that when multiple sinks are presented, the algorithm will be conservative and not optimal. More specifically, since there may be multiple gate oxides connected to one wire segment, the discharging current is "shared" among those gates. Using a small upper-bound on the total exposed antenna area would over-constrain the routing,

while using a large upper-bound on the total exposed antenna area would result in a huge number of wire segments violating the antenna ratio rule. In addition, when gates are commonly sized for performance and power consumption during design, determining a proper upper-bound on the total exposed antenna area could be more difficult. Therefore, the upper-bound on antenna ratio, which is the most important rule for the antenna effect, should be directly addressed to improve the accuracy in both antenna planning in routing and antenna fixing in post-layout stages.

In this chapter, we present an optimal algorithm to solve the jumper insertion problem under the upper-bound of the antenna ratio. Our algorithm handles general routing trees, i.e. Steiner trees, as well as obstacles. Since we directly consider the ratio upper-bound, we get better result for antenna avoidance. We first formulate the RatioJI problem that models jumper insertion under ratio upper-bound. Then we solve it via dynamic programming by the RatioPart algorithm. The time complexity is $O(\alpha|V|^2)$ and the space complexity is $O(|V|^2)$, where $|V|$ is the number of the nodes in the routing tree and α is a factor depending on how to find a non-blocked position on a wire for a jumper. In our experiments with and without obstacles, we have an α equal to 1. Moreover, our dynamic programming algorithm works on free trees. It is different from the classical dynamic programming approaches in the VLSI CAD area [**106, 107**], which work on rooted trees. We believe that the general framework is valuable for other problems on free trees and that it is possible to exploit this characteristic to reduce the practical running time via heuristics that determine the order for performing operations dynamically.

The rest of this chapter is organized as follows. In Section 8.1, backgrounds on antenna effect are introduced. The RatioJI problem is formulated in Section 8.2. Algorithm

to solve the problem is presented in Section 8.3. After experimental results given in Section 8.4, Section 8.5 concludes the chapter.

8.1. Antenna Effect

A detailed overview of the antenna effect can be found in the work [**93**]. We briefly introduce the backgrounds here.

VLSI chips are commonly fabricated layer by layer. Since interconnect networks consist of wires and vias on and between different layers, wire segments are formed during the fabrication. Some of them connect to gate inputs, which are gate oxides, and others connect to gate outputs, which are diffusions. During radio frequency (RF) plasma processes, exposed wire segments act as antennas. They collect ion and electron currents from the plasma. Since a wire segment also acts as a capacitor, if the two currents do not cancel each other through every RF cycle and the wire segment does not connect to a diffusion, charging on the capacitor happens and an antenna voltage V_g will build up. When the wire segment only connects to gate oxides, V_g reaches a steady state when the Fowler–Nordheim (FN) tunneling current through the gate oxides balances the plasma caused charging current. In such steady state, V_g are also the gate voltages on all the gate oxides connected to the wire segment. The charging current density J_p is a function of the antenna voltage and the tunneling current density J_{FN} is a functions of the gate voltage when the technology parameters are given as constants. Since both the antenna voltage and the gate voltage are V_g, the following equation shows the relationship between the antenna ratio and the current densities:

$$\text{antenna ratio} \triangleq \frac{\text{total exposed antenna area}}{\text{total gate oxide area}} = \frac{J_{FN}(V_g)}{J_p(V_g)}.$$

As plotted in Figure 12 of the work [**93**], the charging current decreases with the increase of V_g; but the tunnelings current, which damages the gate oxide, increases with the increasing of V_g. A higher antenna ratio means a larger V_g and thus a larger J_{FN} and more damage.

The methods to compute the exposed antenna area are different for different plasma-based manufacturing processes. According to the work [**97**], there are three types. The first are the conductor layer pattern etching processes where the perimeters of the wires are exposed. So the exposed area is computed as the perimeter length of conductor layer patterns. The second are the ashing processes where the area of the wires are exposed. Thus the exposed area is the area of the conductor layer patterns. The third are the contact etching processes where the area of the contacts on the lower conductor layer are exposed. Therefore the exposed area is the total area of the contacts. Our problem formulation is general enough to handle all these types.

8.2. Problem Formulation

Considering jumper insertion on a general routing tree for an upper-bound R on the antenna ratio, let $T = (V, E)$ be the routing tree where V is the set of nodes representing the gates as well as the Steiner points and E is the set of tree edges representing wires connecting those nodes. A function g is defined on every node $v \in V$: if v represents a gate, $g(v) > 0$ gives the gate oxide area; if v does not represent a gate but a Steiner

point, let $g(v) = 0$. A function l is defined on an edge segment s: $l(s) \geq 0$ is the exposed antenna area of s.

Jumpers will be inserted on edges to form cuts. For an edge $e = (u, v) \in E$, a *cut* c_e divides e into edge segments. The size of the cut c_e, written as $|c_e|$, is defined as the number of the jumpers inserted. Since there is no gate within the edge, it is obvious that at most 2 jumpers are enough to satisfy the antenna rule. Thus c_e is written as

$$c_e = (p_0 = u, p_1, \ldots, p_{k+1} = v), k = 0, 1, \text{ or } 2.$$

where $(p_{i-1}, p_i), i = 1, 2, \ldots, k+1$ are the edge segments and $|c_e| = k$. It is not always possible to insert a jumper at any position. Various reasons introduce obstacles along the wire. For example, the top layers may be occupied by other nets. To model the obstacles on a particular wire, which are the positions that jumper cannot be inserted, we use $\mathcal{C}_e$ to denote the set of allowed cuts on an edge e.

By assigning one cut to each edge in the tree, T is partitioned into connected components. The partitioning is given by the set of the cuts: $C = \{c_e : e \in E\}$. A partitioning C is called *feasible* if $c_e \in \mathcal{C}_e$ for every $e \in E$. Obviously the number of all the jumpers inserted is $\sum_{e \in E} |c_e|$. For a connected component S, let $g(S)$ be the total gate oxide area and $l(S)$ be the total exposed antenna area. A feasible partitioning is *valid* if for every connected component S, either $g(S) = 0$ or $\frac{l(S)}{g(S)} \leq R$. The reason is that, according to Section 8.1, for all the gate oxides in a specific connected component, the tunneling current densities are the same since the gate voltages are all equal to the antenna voltage. Intuitively, the plasma charging current is "shared" among those gates. A connected

component can survive larger total exposed antenna area if the total gate oxide area is larger. Note that it is possible to have no valid partitioning because of the obstacles.

Since inserting jumpers will degrade performance and manufacturing yield, it is preferred to find the minimal number of jumpers to meet a specific ratio upper-bound. The corresponding partitionings are called the *optimal* partitionings. The optimal jumper insertion under ratio upper-bound is formulated as the following RatioJI problem.

Problem 8.1 (RatioJI)**.** *Suppose $T = (V, E)$ is a tree, functions g and l model the gate oxide area and the exposed antenna area respectively, and set $\mathcal{C}_e$ models the obstacles on a wire e where jumpers cannot be inserted. For a ratio upper-bound R, find an optimal partitioning, which is a partitioning C with $c_e \in \mathcal{C}_e, \forall e \in E$ and a minimal number of jumpers such that for a connected component S, either the total gate oxide area $g(S)$ is 0 or the ratio of total exposed antenna area to total gate oxide area $\frac{l(S)}{g(S)} \leq R$.*

The RatioJI problem models both antenna fixing and antenna planning under various antenna ratio models. For antenna fixing after layer assignment, the problem can be solved at each layer to meet the antenna ratio rule on that layer with proper parameters. For antenna planning in routing, the layer assignment can be obtained in an algorithm similar to those in [**102, 103**] by predicting jumper positions via solving the RatioJI problem. Although the predictions may be inaccurate, the previous works [**102, 103**] showed that the number of jumpers required for antenna fixing later could be reduced.

Two commonly used models of antenna ratio are the partial antenna ratio (PAR) and cumulative antenna ratio (CAR) [**103**]. In PAR, the antenna ratio bound is applied to the antenna ratio computed at a specific layer for a specific wire segment at a time. Different

layer may have different bound. The RatioJI problem models PAR by constructing T to represent the part of routing tree including that wire segment and all the interconnects and gates connected to it in lower layers which are already fabricated. Other parameters are chosen accordingly. In CAR, the antenna ratio is defined as the ratio of the cumulative antenna area to its connected gate area. The RatioJI problem models CAR by constructing T to represent the cumulative antenna area and its connected gates and l to represent the antenna area contributing to the cumulative antenna area.

8.3. Optimal Jumper Insertion

8.3.1. Algorithm Overview

We solve the RatioJI problem by dynamic programming. Classical dynamic programming algorithms usually work on a rooted tree [**106, 107**]. However, a routing tree T is a free tree without a root. Our algorithm will process the edges one by one in some order. We briefly describe our algorithm here while the definitions of terminologies and the details of algorithm will be presented in later sections.

Our algorithm consists of two stages. In the first stage, *dominant solutions* are generated bottom-up. At any step, the edges are divided into two groups: processed or unprocessed. *Dominant partial solutions* are maintained for every tree in the forest formed by connecting all nodes via the processed edges. At the beginning, no edge is processed. Thus every node is a tree by itself and has only one *partial solution*. The algorithm processes one edge at a time by combining two trees with an unprocessed edge and computing the dominant partial solutions of the new tree based on those of the two old trees. An edge can be processed only if it is the only unprocessed edge incident to one tree. An

example is shown in Figure 8.1. Starting from Figure 8.1 (a), there are three possible edges to be processed next as shown in (b), (c), and (d). Note that the edge (e, h) cannot be processed at the current step. When every edge is processed at the end, the forest contains one tree, which is T itself, and all the dominant solutions are generated.

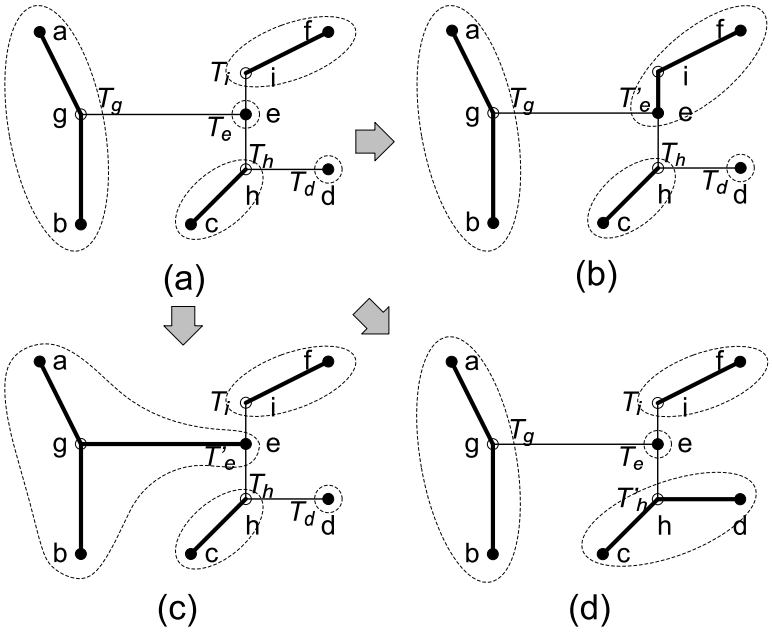

Figure 8.1. Process one edge by combining two trees. Bold edges are processed; others are unprocessed. Solid nodes are gates; others are Steiner points. In (a), the edges (a, g), (b, g), (f, i), and (c, h) are processed while the dominant partial solutions are maintained on trees T_g, T_d, T_h, and T_i. One of the three edges, (e, i), (e, g), and (d, h), could be processed next: in (b), (e, i) is processed by combining T_e and T_i into T'_e; in (c), (e, g) is processed by combining T_e and T_g into T'_e; in (d), (d, h) is processed by combining T_h and T_d into T'_h.

In the second stage, since there is one optimal partitioning among the dominant solutions if there exists a valid partitioning, we can either find an optimal partitioning or report that there is no valid partitioning. A top-down back-tracking will construct the optimal partitioning if there is one.

8.3.2. Dominant Solutions

Denote the forest formed by connecting all nodes via the processed edges by F. Suppose there are m trees in F, written as $T_1, T_2, \ldots, T_m$. There are $m-1$ unprocessed edges currently. When $m > 1$, every T_i has at least one unprocessed edge connects to it because T is connected. Since an edge can be processed only if it is the only unprocessed edge to one tree, each tree T_i can only have one node u_i incident on the unprocessed edges.

For a tree T_i, *partial solution*s are the partitionings of T_i with the following properties. For every component S not containing u_i, the ratio upper-bound must be satisfied, i.e., either $g(S) = 0$ or $\frac{l(S)}{g(S)} \leq R$. Denote the component containing u_i by S_i. If there is at least one unprocessed edge incident on u_i, the upper-bound can be violated in S_i, which may be corrected when future combinations are executed. For a partition of T_i, let n be the number of the jumpers inserted. Define $x = l(S_i) - g(S_i)R$. Let I be 0 if $g(S_i) = 0$ and 1 if $g(S_i) > 0$. The partial solution is represented by the triple (n, x, I). To ease the representation, we refer to the partial solution by the triple when there is no ambiguity.

Dominant relationship is defined among the partial solutions such that only necessary partial solutions are maintained for each tree T_i. For two different partial solutions $P_1 = (n_1, x_1, I_1)$ and $P_2 = (n_2, x_2, I_2)$ of one tree, we say that P_1 *dominates* P_2 if $(n_1 \leq n_2) \wedge (x_1 \leq x_2) \wedge (I_1 = I_2)$. The reason is that a partial solution with smaller n and x can always substitute another one of the same I in any optimal partitioning of T.

If there is no other partial solutions dominating (n, x, I), we call (n, x, I) a *dominant partial solution*. By denoting the number of nodes in T_i as N_i, the number of dominant partial solutions of T_i is upper-bounded by a linear function of N_i.

Lemma 8.1. *There are at most $2N_i - 2$ jumpers on a tree T_i of N_i nodes and thus there are at most $4N_i - 2$ dominant partial solutions to maintain.*

Proof. A tree T_i of N_i nodes has $N_i - 1$ edges. Since at most 2 jumpers are inserted per edge, there are at most $2N_i - 2$ totally.

For each $n \in \{0, 1, \ldots, 2N_i - 2\}$ and $I \in \{0, 1\}$, only one dominant partial solution (n, x, I) is maintained. Thus there are at most $(2N_i - 1) \times 2 = 4N_i - 2$ dominant partial solutions to maintain.

□

When there is only one tree in the forest, i.e. $m = 1$, we call the dominant partial solutions of the tree *dominant solutions*. Note that the set of dominant solutions could be different with different orders to process the edges.

8.3.3. Generation of Dominant Solutions

In this section, we consider how to generate solutions for a new tree by assimilating one tree into another. Assume that there are more than one tree in the forest, i.e, $m > 1$. Consider an unprocessed edge $e = (u, v)$ connecting two trees T_u and T_v in the forest F. We process the edge e and combine T_u and T_v only when e is the sole unprocessed edge incident on u. Denote the new tree made up of T_u, T_v, and e by T'_v. The dominant partial solutions of tree T'_v are generated from those of the trees T_u and T_v, as stated in Lemma 8.2.

Lemma 8.2. *A dominant partial solution (n, x, I) of T'_v consists of a dominant partial solution (n_v, x_v, I_v) of T_v, a dominant partial solution (n_u, x_u, I_u) of T_u, and a cut c_e on e. The c_e and (n, x, I) must be one of the following cases.*

1. *If $c_e = (u, v)$, then $n = n_v + n_u$ and $x = x_v + x_u + l(u, v)$. The I is 0 if $I_v = I_u = 0$; otherwise I is 1.*

2. *If $c_e = (u, p_1, v)$, then $n = n_v + n_u + 1$, $x = x_v + l(p_1, v)$, and $I = I_v$. The I_v and I_u must not be 1 at the same time. If $I_u = 1$, then c_e is the element in $\mathcal{C}_e$ satisfying $x_u + l(u, p_1) \leq 0$ with the minimal $l(p_1, v)$. If $I_u = 0$, then c_e is the element in $\mathcal{C}_e$ with the minimal $l(p_1, v)$.*

3. *If $c_e = (u, p_1, p_2, v)$, then $n = n_v + n_u + 2$, $x = x_v + l(p_2, v)$, and $I = I_v$. If $I_u = 1$, then c_e is the element in $\mathcal{C}_e$ satisfying $x_u + l(u, p_1) \leq 0$ with the minimal $l(p_2, v)$. If $I_u = 0$, then c_e is the element in $\mathcal{C}_e$ with the minimal $l(p_2, v)$.*

Proof. We proof the lemma by examining each case.

1. When $c_e = (u, v)$, it is straightforward that $n = n_v + n_u$, $x = x_v + x_u + l(u, v)$, and I is 0 if $I_v = I_u = 0$ or 1 otherwise.
2. When $c_e = (u, p_1, v)$, it is straightforward that $n = n_v + n_u + 1$, $x = x_v + l(p_1, v)$, and $I = I_v$. Denote the component containing u in the partial solution (n, x, I) by S.

   If $I_u = 1$, we have $g(S) > 0$. The following condition should be satisfied by p_1:

   $$0 \geq l(S) - g(S)R = x_u + l(u, p_1).$$

   We proof $I_v = 0$ by contradiction. If $I_v \neq 0$, we have $I_v = 1$ and that,

$$\begin{aligned} x &= x_v + l(p_1, v) \\ &\geq x_v + l(p_1, v) + (x_u + l(u, p_1)) \\ &= x_v + x_u + l(u, v). \end{aligned}$$

The (n, x, I) is dominated by the partial solution in case *1.*, which violates our assumption. So I_v should be 0 and c_e should be the element in $\mathcal{C}_e$ satisfying $x_u + l(u, p_1) \leq 0$ with the minimal $l(p_1, v)$.

If $I_u = 0$, we have $g(S) = 0$. So c_e should be the element in $\mathcal{C}_e$ with the minimal $l(p_1, v)$.

3. When $c_e = (u, p_1, p_2, v)$, it is straightforward that $n = n_v + n_u + 2$, $x = x_v + l(p_2, v)$, and $I = I_v$. Denote the component containing u in the dominant partial solution (n, x, I) by S.

   If $I_u = 1$, we have $g(S) > 0$. The following condition should be satisfied by p_1:

   $$0 \geq l(S) - g(S)R = x_u + l(u, p_1).$$

   So c_e should be the element in $\mathcal{C}_e$ satisfying $x_u + l(u, p_1) \leq 0$ with the minimal $l(p_2, v)$.

   If $I_u = 0$, we have $g(S) = 0$. So c_e should be the element in $\mathcal{C}_e$ with the minimal $l(p_2, v)$.

□

Suppose D_v and D_u are the sets of the dominant partial solutions of T_v and T_u respectively. We design the Combine subroutine as shown in Figure 8.2 based on Lemma 8.2. The Combine subroutine updates D_v to be the set of the dominant partial solutions of T'_v. A set B is stored for every dominant partial solution to enable the back-tracking which is used to construct the optimal partitioning later.

Subroutine Combine

Inputs

$e = (u, v)$, D_v, and D_u.

Outputs

Updated D_v.

```
1  D* ← ∅.
2  For each (d_v, d_u) in the set D_v × D_u:
3    (n_v, x_v, I_v, B_v) ← d_v; (n_u, x_u, I_u, B_u) ← d_u.
4    For each case in Lemma 8.2:
5      Compute (n, x, I) and c_e.
6      B ← B_v ∪ {(n_u, x_u, I_u, u, e, c_e)}.
7      Add (n, x, I, B) to D*.
8  Remove non-dominant partial solutions from D*.
9  D_v ← D*.
```

Figure 8.2. The Combine subroutine.

In Figure 8.2, D^* holds the candidates of the dominant partial solutions. Suppose T_u has N_u nodes and T_v has N_v nodes. The set D^* is implemented as an array of $4(N_u+N_v)-2$ elements since there are at most $4(N_u + N_v) - 2$ combinations of n and I according to Lemma 8.1. Since D_u will not be modified by any following Combine calls, the position of element (n_u, x_u, I_u, B_u) in the array D_u is stored instead of the triple (n_u, x_u, I_u) on line 6. For the partial solutions generated on line 7, only the ones with different n's and I's are stored in D^*; if there are partial solutions with the same n and I, only the one

with the least x is stored. Remaining non-dominant partial solutions in D^* are removed on line 8. Then line 7 takes $O(1)$ time and both lines 1 and 8 take $O(N_u + N_v)$ time.

8.3.4. The RatioPart Algorithm

ALGORITHM RatioPart

Inputs

T, g, l, $\mathcal{C}_e$'s, and R.

Outputs

Report an optimal partitioning if there is one.

```
1  Mark all the edges as unprocessed.
2  For each u in V:
3    If g(u) = 0:
4      D_u ← {(0, 0, 0, ∅)}.
     Else:
5      D_u ← {(0, −g(u)R, 1, ∅)}.
6  While there is at least one unprocessed edge:
7    Pick an edge e = (u, v) such that it is the only unprocessed edge on
     u.
8    Update D_v by the Combine subroutine.
9    Mark e as processed.
10 w ← the last v in the While loop.
11 d_1 ← the element with the least n in D_w whose I = 0.
12 d_2 ← the element with the least n in D_w whose I = 1 and x ≤ 0.
13 If none of d_1 and d_2 exists:
14   Report there is no valid partitioning.
     Else:
15   (n, x, I, B) ← the one with the smaller n.
16   ReportPart(B).
```

Figure 8.3. The RatioPart algorithm.

Figure 8.3 gives the RatioPart algorithm that solves the RatioJI problem. At the beginning, no edge is processed. The forest contains $|V|$ trees where every tree contains one node. There is one dominant partial solution on each tree since there is only one possible partitioning. According to this, the D_u's are built on line 2 to 5. The **While**

loop on line 6 to 9 processes the edges and updates the dominant partial solutions. The invariant of the loop is stated in Lemma 8.3.

Lemma 8.3. *At line 7 of the RatioPart algorithm, for every $1 \leq i \leq m$, all unprocessed edges connecting to T_i incident on one node u_i and D_{u_i} has all the dominant partial solutions of T_i.*

Proof. We prove the above invariant by induction.

Before the **While** loop on line 6 to 9, every node v is a tree T_v by itself. All unprocessed edges connecting to T_v incident on v. The **For** loop on line 2 to 5 initializes all the D_v's correctly.

Assume the invariant holds on line 7. Denote the tree containing u by T_u and v by T_v respectively. Denote the tree after processing e and combining T_u and T_v by T'_v. We only need to prove that the invariant holds for T'_v when the current loop finishes since all the other trees and their dominant partial solutions remain unchanged.

Since all unprocessed edges connecting to T_u incident on u, the edge e is the only unprocessed edge connecting to T_u. After e is processed, there is no unprocessed edge connecting to T_u. Thus all unprocessed edges connecting to T'_v incident on v.

We prove the D^* in the Combine subroutine contains all the dominant partial solutions of T'_v by contradiction. Assume this is not the case. Suppose the dominant partial solution $(n', x', I') \notin D^*$ and it consists of some partial solution (n'_v, x'_v, I'_v) of T_v, some partial solution (n'_u, x'_u, I'_u) of T_u, and some cut c_e on e. Because D_v has all the dominant partial solutions of T_v, there must exist $(n_v, x_v, I_v) \in D_v$ such that $n_v \leq n'_v$, $x_v \leq x'_v$ and $I_v = I'_v$. Similarly there is $(n_u, x_u, I_u) \in D_u$ such that $n_u \leq n'_u$, $x_u \leq x'_u$ and $I_u = I'_u$. So,

denoting the partial solution consisting of (n_v, x_v, I_v), (n_u, x_u, I_u), and c_e by (n, x, I), we have $n \leq n'$, $x \leq x'$, and $I = I'$. Since (n', x', I') is a dominant partial solution, it must be true that $n = n'$, $x = x'$, and $I = I'$. On the other hand, according to Lemma 8.2, either $(n, x, I) \in D^*$ or there exists some partial solution in D^* dominating (n, x, I). This contradicts that $(n', x', I') \notin D^*$ is a dominant partial solution.

Therefore the invariant holds for T'_v after the current loop.

□

The code on line 11 to 16 searches for an optimal partitioning in the set D_w. It is guaranteed to find one if there is one, which is stated in Lemma 8.4.

Lemma 8.4. *If there is a valid partitioning of T, then an optimal partitioning is presented in the dominant solutions D_w.*

Proof. If there is a valid partitioning, there must be a optimal partitioning. Suppose the partial solution (n, x, I) of T_w is the optimal partitioning with the minimal x. We prove $(n, x, I) \in D_w$ by contradiction. According to Lemma 8.3, D_w has all the dominant partial solutions of T_w. If $(n, x, I) \notin D_w$, then there exists $(n', x', I') \in D_w$ dominating (n, x, I). So $n' \leq n$, $x' \leq x$, and $I' = I$. Thus (n', x', I') is also valid. Since (n, x, I) is the optimal partitioning with the minimal x, we should have $n' \geq n$ and $x' \geq x$. So $n = n'$, $x = x'$, and $I = I'$, which contradicts that (n', x', I') dominating (n, x, I).

□

On line 15, (n, x, I) is the triple of the optimal partitioning and the set B contains information to reconstruct the optimal partitioning. The subroutine ReportPart, as shown in Figure 8.4, takes B as the parameter and recursively reports the optimal partitioning.

The search on line 3 of the ReportPart subroutine takes constant time since we implement D_u as an array and the position of element with the triple (n_u, x_u, I_u) in the array is stored.

Subroutine ReportPart
Inputs The set B.
Outputs Report the cuts corresponding to B.
1 **For** each (n, x, I, u, e, c_e) in B: 2 Report the cut c_e on edge e. 3 Find the element (n_u, x_u, I_u, B_u) in D_u satisfying $n_u = n$, $x_u = x$, and $I_u = I$. 4 ReportPart(B_u).

Figure 8.4. The ReportPart subroutine.

The correctness of the algorithm is given by the following theorem.

Theorem 8.1. *The RatioPart algorithm terminates in finite time and gives an optimal partitioning if there is a valid one.*

Proof. The **While** loop on line 6 to 9 terminates because the number of unprocessed edges is limited and decreases by one each time. The ReportPart subroutine terminates because the number of edges is limited and the cut on each edge is reported exactly once. It is straightforward that the other parts of the RatioPart algorithm terminate so the whole RatioPart algorithm terminates.

A valid partitioning is a partial solution (n, x, I) of T_w such that either $I = 0$ or $(I = 1) \wedge (x \leq 0)$. By searching all such partial solution in D_w, the RatioPart algorithm gives an optimal partitioning if there is a valid one according to Lemma 8.4.

□

In the current implementation, the order of the edges to be processed is determined statically in the RatioPart algorithm for simplicity. From an arbitrary node, a depth-first search (DFS) [**29**] on the routing tree is performed. Define the finishing time of edge (u, v) to be the finishing time of v where we assume that u is discovered prior to v. The edges are ordered according to their finishing times. For example, by a DFS of the tree in Figure 8.1 starting from the node e, the nodes are discovered in the order e, g, a, b, h, c, d, i, and f and finished in the order a, b, g, c, d, h, f, i, and e. Thus the order of the edges are (a, g), (b, g), (g, e), (c, h), (d, h), (h, e), (f, i), and (i, e). Note that it is possible to dynamically determine which edge to be processed next when there are multiple choices like the situation in Figure 8.1. Although the set of the dominant solutions may be different for different choices, Theorem 8.1 ensures that an optimal partitioning will always be found if there is one. A good heuristic may benefit both the running time and the storage requirement. We leave this as a direction of future research.

8.3.5. Complexity of the RatioPart Algorithm

The space complexity of the RatioPart algorithm is stated in Theorem 8.2.

Theorem 8.2. *The space complexity of the RatioPart algorithm is* $O(|V|^2)$.

Proof. In the RatioPart algorithm, D_u is stored for every u. Each D_u contains at most $4|V| - 2$ elements according to Lemma 8.1. Suppose there are m_u edges incident on u in T. Then for every element in D_u, it contains a set B with at most m_u elements. So D_u requires at most $O(m_u|V|)$ storage and the total storage required by all the D_u's is that,

$$\sum_{u \in V} O(m_u|V|) = O((\sum_{u \in V} m_u)|V|) = O(|V|^2).$$

For the Combine subroutine, D^* is an array of at most $O(|V|)$ elements and every element requires at most $O(|V|)$ storage. So it requires at most $O(|V|^2)$ storage. For the ReportPart subroutine, since the depth of the recursion is at most $|V|$, it requires $O(|V|)$ storage. Therefore, the space complexity for the RatioPart algorithm is $O(|V|^2)$.

□

To evaluate the time complexity, we assume that both functions g and l take constant time to compute. The time complexity is stated in Theorem 8.3.

Theorem 8.3. *The time complexity of the RatioPart algorithm is $O(\alpha|V|^2)$ where α is a factor depending on how to find a non-blocked position on a wire for a jumper.*

Proof. We use the potential method [**29**] to evaluate the total running time of the Combine subroutine when it is used in the RatioPart algorithm. In the Combine subroutine, suppose the line 5 takes $O(\alpha)$ time, where α is a factor depending on how to find the element in $\mathcal{C}_e$ according to Lemma 8.2, i.e., how to find a non-blocked position on a wire for a jumper. Recall that N_v and N_u are the number of nodes of the trees T_v and T_u respectively. Then the loop from line 2 to 7 takes $O(\alpha N_u N_v)$ time. As discussed in Section 8.3.3, line 1 and 8 take $O(N_u + N_v)$ time. So the total running time of the Combine algorithm is $O(\alpha N_u N_v)$. Assume that the upper-bound on the running time is $A\alpha N_u N_v$ where A is a constant.

For a forest F, define its potential to be

$$\Phi(F) = \frac{A}{2}\alpha \sum_{i=1}^{m} N_i^2$$

where N_i is the number of the nodes in the tree T_i. Suppose a forest F' is obtained after applying the Combine subroutine to process edge (u, v) and combine T_v and T_u. Denote the running time for this Combine run by t, we have

$$\Phi(F') - \Phi(F) = A\alpha N_u N_v \geq t.$$

Since initially the potential is $\frac{A}{2}\alpha|V|$ and finally the potential is $\frac{A}{2}\alpha|V|^2$, the Combine subroutine takes at most

$$\frac{A}{2}\alpha|V|^2 - \frac{A}{2}\alpha|V| = O(\alpha|V|^2)$$

time when it is used in the RatioPart algorithm.

In the ReportPart subroutine, every D_u is searched at most once in the recursion. Therefore, line 16 in the RatioPart algorithm takes $O(|V|)$ time.

All the other parts in the RatioPart algorithm take at most $O(|V|^2)$ time. So the time complexity is $O(\alpha|V|^2)$.

□

8.4. Experiments

The experiments are conducted on a Linux workstation with dual 933MHz Pentium III processors and 512MB memory where the RatioPart algorithm is implemented in C++ and compiled with GCC 3.4. Since there is no previous work considering the upper-bound of the antenna ratio with multiple sinks, we do not compare our results to other's.

Two sets of benchmarks are used in the experiments. The benchmarks in the first set are 4 randomly generated ones containing 100, 1000, 10000, and 20000 gates respectively, where the gates are randomly located in a 10000 by 10000 grid. The benchmarks in the second set are the 4 largest industrial benchmarks in the work [**108**]. For each benchmark,a Steiner tree is constructed as the routing tree using the algorithm in the work [**109**]. All the gate sizes are assumed to be 1 and the exposed antenna area of a wire is computed as the wire length, which is the Manhattan distance between the two nodes it connects. The statistics of the benchmarks are shown in Table 8.1. The column "name" shows the name of each benchmark where the benchmarks with names starting with "a" are the random generated ones and the benchmarks with names starting with "n" are the industrial ones. The column "# gates" shows the number of the gates in each benchmark. The column "# nodes" shows the number of the nodes on the routing tree including the gates and the Steiner points. The column "SMT" shows the total wire length of the Steiner tree. The column "ratio" shows the antenna ratio of the routing tree when all the wires are exposed. For any antenna ratio upper-bound larger than the number in this column, no jumper is needed.

Table 8.1. Statistics of the benchmarks for jumper insertion.

name	# gates	# nodes	SMT	ratio
a100	100	154	79189	792
a1000	1000	1439	229604	230
a10000	10000	14542	721452	73
a20000	20000	28989	1019236	51
n2676	2676	3733	81180	31
n12052	12052	16447	243886	21
n22373	22373	30497	1264194	57
n34728	34728	47954	904740	27

We first run the RatioPart algorithm without obstacles. For each benchmark, 6 upper-bounds of the antenna ratio are chosen representatively according to the "ratio" column in Table 8.1. The results are reported in Table 8.3. The "ratio" columns show the upper-bounds used. The "# cuts" columns show the number of the jumpers inserted. The "time(s)" columns show the running time in seconds. It can be seen that the practical running times depend not only on the number of the nodes but also on the bound since different bounds will result in different numbers of partial dominant solutions to maintain on each tree. In Figure 8.5, the trend of the decrease of the running time with the increase of the antenna ratio is shown by plotting the number of jumpers inserted and the running time in seconds vs. different bounds for the benchmark a10000. On the other hand, the quadratic theoretical time complexity can be verified in Figure 8.6. There is one sample for each benchmark in the figure. Each sample is the one with the smallest bound among the 6 bounds for one benchmark. The slope of the best-fitting line in the log–log plot shows a practical running time of $\Theta(|V|^{1.85})$.

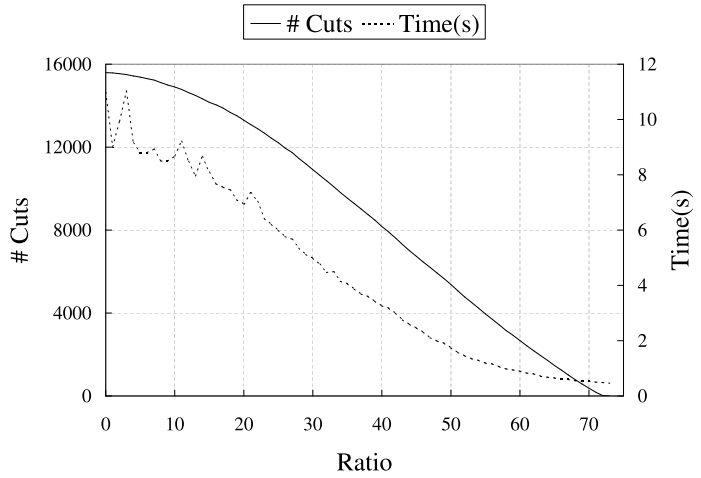

Figure 8.5. The number of the cuts and the running time vs. the antenna ratio bound for a10000 without obstacles.

Table 8.2. Results of jumper insertion with obstacles.

	a100 ratio=700		a1000 ratio=220		a10000 ratio=70		a20000 ratio=50	
% ob	# cuts	time(s)	# cuts	time(s)	# cuts	time(s)	# cuts	time(s)
0%	18	0.002	54	0.016	376	0.514	444	1.969
10%	22	0.002	56	0.014	394	0.473	464	1.850
20%	–	0.002	56	0.014	421	0.461	483	1.756
30%	–	0.002	60	0.016	438	0.455	508	1.617
40%	–	0.002	63	0.014	485	0.371	531	1.420
50%	–	<1ms	69	0.016	532	0.320	574	1.223
60%	–	<1ms	87	0.012	625	0.299	618	1.004
70%	–	<1ms	–	0.010	979	0.236	717	0.775
80%	–	<1ms	–	0.008	–	0.156	976	0.615
90%	–	<1ms	–	0.008	–	0.105	–	0.285
	n2676 ratio=30		n12052 ratio=20		n22373 ratio=55		n34728 ratio=26	
% ob	# cuts	time(s)	# cuts	time(s)	# cuts	time(s)	# cuts	time(s)
0%	6	0.051	23	0.799	359	2.086	6	4.914
10%	6	0.055	25	0.707	408	1.885	10	4.633
20%	6	0.051	65	0.635	460	1.672	13	4.068
30%	6	0.047	69	0.576	518	1.455	13	3.598
40%	15	0.043	118	0.523	623	1.318	20	2.955
50%	15	0.041	159	0.426	751	1.053	20	2.406
60%	19	0.037	183	0.371	881	0.834	20	2.025
70%	29	0.029	226	0.293	1207	0.650	24	1.529
80%	44	0.029	266	0.225	–	0.400	32	1.029
90%	–	0.018	–	0.139	–	0.236	50	0.590

We then run the RatioPart algorithm with obstacles. We randomly choose edges to be *forbidden* which means that no jumper insertion is allowed on them. For each benchmark, we choose one representative bound, which is the second largest one among the 6 bounds in Table 8.3. Scenarios with different percentage of forbidden edges are tested. The results are reported in Table 8.2. The column "% ob" shows the percentage of the forbidden edges. The cells with "–" denote that no valid partitioning under the upper-bound can be found with the obstacles presented. Note that the obstacle setting here is only for

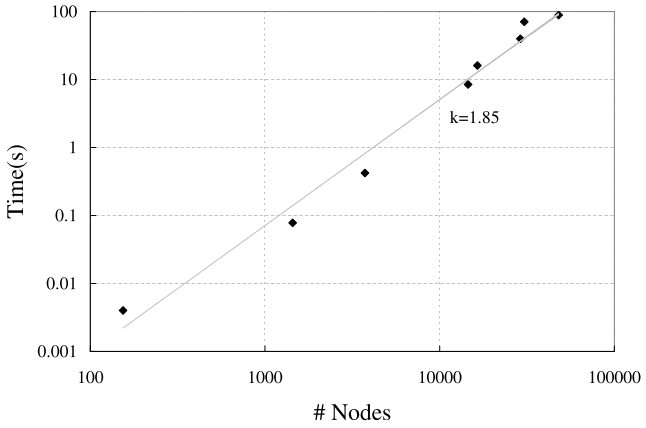

Figure 8.6. The running time vs. the number of the nodes for small antenna ratio bounds.

simplicity while the RatioPart algorithm can handle more general settings, e.g. the one in the work [**105**] where the obstacles can forbid inserting jumpers on parts of an edge.

8.5. Summary

In this chapter, we presented an optimal algorithm for antenna avoidance via jumper insertion under the upper-bound of the antenna ratio. The algorithm is based on dynamic programming on free trees. The experimental results confirmed the effectiveness of our approach. Future works include routing with antenna planning and heuristics for practical running time reduction.

Table 8.3. Results of jumper insertion without obstacles.

a100			a1000			a10000			a20000		
ratio	# cuts	time(s)	ratio	# cuts	time(s)	ratio	# cuts	time(s)	ratio	# cuts	time(s)
100	141	0.004	50	1406	0.078	10	14902	8.48	10	28807	39.8
200	138	0.004	150	607	0.037	30	10912	4.93	20	22873	26.9
400	92	0.002	200	197	0.018	50	5374	1.74	30	15020	12.8
600	41	0.002	210	121	0.016	60	2675	0.90	40	7091	4.78
700	18	0.002	220	54	0.016	70	376	0.51	50	444	1.97
800	0	<1ms	230	0	0.016	80	0	0.38	60	0	1.13
n2676			n12052			n22373			n34728		
ratio	# cuts	time(s)	ratio	# cuts	time(s)	ratio	# cuts	time(s)	ratio	# cuts	time(s)
10	3363	0.422	4	18048	16.1	10	36693	70.6	10	41853	88.7
15	2394	0.258	8	14403	9.83	30	18770	14.0	15	25652	37.7
20	1383	0.150	12	8593	4.48	40	10020	5.65	20	10804	13.4
25	508	0.074	16	3352	1.29	50	2833	2.74	23	3797	7.55
30	6	0.051	20	23	0.80	55	359	2.09	26	6	4.91
35	0	0.047	22	0	0.57	60	0	1.47	30	0	2.71

References

[1] J. Wang and H. Zhou. Minimal period retiming under process variations. In *Proc. of Great Lake Symposium on VLSI*, pages 131–135, 2004.

[2] J. Wang and H. Zhou. Interconnect estimation without packing via ACG floorplans. In *Proc. Asian and South Pacific Design Automation Conference*, pages 1152–1155, 2005.

[3] J. Wang and H. Zhou. Exploring adjacency in floorplanning. Unpublished, 2005.

[4] J. Wang, P. Wu, and H. Zhou. Processing rate optimization by sequential system floorplanning. In *Proc. International Symposium Quality Electronic Design*, pages 340–345, 2006.

[5] J. Wang and H. Zhou. Optimal jumper insertion for antenna avoidance under ratio upper–bound. In *Proc. of the Design Automation Conf.*, pages 1445–1453, 2006.

[6] J. Wang, M. Y. Kao, and H. Zhou. Address generation for nanowire decoders. In *Proc. of Great Lake Symposium on VLSI*, pages 525–528, 2007.

[7] J. Wang and H. Zhou. Optimal jumper insertion for antenna avoidance considering antenna charge sharing. *IEEE Transactions on Computer Aided Design*, 26(8):1445–1453, August 2007.

[8] J. Wang, D. Das, and H. Zhou. Gate sizing by lagrangian relaxation revisited. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 111–118, 2007.

[9] J. Wang and H. Zhou. An efficient incremental algorithm for min-area retiming. In *Proc. of the Design Automation Conf.*, 2008.

[10] H. Zhou and J. Wang. ACG-adjacent constraint graph for general floorplans. In *Proc. Intl. Conf. on Computer Design*, pages 572–575, 2004.

[11] Z. Gu, J. Wang, R. P. Dick, and H. Zhou. Incremental exploration of the combined physical and behavioral design space. In *Proc. of the Design Automation Conf.*, pages 208–213, 2005.

[12] C. Lin, J. Wang, and H. Zhou. Clustering for processing rate optimization. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 189–195, 2005.

[13] Z. P. Gu, Y. Yang, J. Wang, R. P. Dick, and L. Shang. TAPHS: Thermal-aware unified physical-level and high-level synthesis. In *Proc. Asian and South Pacific Design Automation Conference*, pages 879–885, 2006.

[14] C. Lin, J. Wang, and H. Zhou. Clustering for processing rate optimization. *IEEE Transactions on Very Large-Scale Integrated Systems*, 14(11):1264–1275, November 2006.

[15] N. Liveris, C. Lin, J. Wang, H. Zhou, and P. Banerjee. Retiming for synchronous data flow graphs. In *Proc. Asian and South Pacific Design Automation Conference*, pages 480–485, 2007.

[16] Z. Gu, J. Wang, R. Dick, and H. Zhou. Unified incremental physical-level and high-level synthesis. *IEEE Transactions on Computer Aided Design*, 26(9):1576–1588, September 2007.

[17] M. R. Casu and L. Macchiarulo. Floorplanning for throughput. In *Proc. International Symposium on Physical Design*, pages 62–69, 2004.

[18] C. E. Leiserson and J. B. Saxe. Optimization synchronous systems. *Journal of VLSI and Computer Systems*, 1:41–67, 1983.

[19] J. P. Fishburn. Clock skew optimization. *IEEE Transactions on Computers*, 39(7):945–951, July 1990.

[20] A. P. Hurst, P. Chong, , and A. Kuehlmann. Physical placement driven by sequential timing analysis. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 379–386, 2004.

[21] C. Lin and H. Zhou. Retiming for wire pipelining in system-on-chip. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 215–220, 2003.

[22] L. P. Carloni, K. L. McMillan, A. Saldanha, , and A. L. Sangiovanni-Vincentelli. A methodology for correct-by-construction latency insensitive design. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 309–315, 1999.

[23] L. P. Carloni and A. L. Sangiovanni-Vincentelli. Performance analysis and optimization of latency insensitive systems. In *Proc. of the Design Automation Conf.*, pages 361–367, 2000.

[24] V. Nookala and S. S. Sapatnekar. A method for correcting the functionality of a wire-pipelined circuit. In *Proc. of the Design Automation Conf.*, pages 570–575, 2004.

[25] S. N. Adya and I. L. Markov. Fixed-outline floorplanning: Enabling hierarchical design. *IEEE Transactions on Very Large-Scale Integrated Systems*, 11(6):1120–1135, December 2003.

[26] J. Cochet-Terrasson, G. Cohen, S. Gaubert, M. Mc Gettrick, , and J-P Quadrat. Numerical computation of spectral elements in max-plus algebra. In *Proc. IFAC Conf. on Syst. Structure and Control*, 1998.

[27] A. Dasdan, S. S. Irani, , and R. K. Gupta. Efficient algorithms for optimum cycle mean and optimum cost to time ratio problems. In *Proc. of the Design Automation Conf.*, pages 37–42, 1999.

[28] R. H.J.M. Otten and R. Brayton. Planning for performance. In *Proc. of the Design Automation Conf.*, pages 122–127, 1998.

[29] T. H. Cormen, C. E. Leiserson, R. H. Rivest, , and C. Stein. *Introduction to Algorithms*. MIT Press, 2 edition, 2001.

[30] R. H.J.M. Otten. What is floorplan? In *Proc. International Symposium on Physical Design*, pages 201–206, 2000.

[31] J. Grason. *A Dual Linear Graph Representation for Space-Filling Location Problems of the Floor-planning Type*. MIT Press, Cambridge, MA, 1970.

[32] K. Kozminski and E. Kinnen. An algorithm for finding a rectangular dual of a planar graph for use in area planning for vlsi integrated circuits. In *Proc. of the Design Automation Conf.*, pages 655–656, 1984.

[33] J. Bhasker and S. Sahni. A linear algorithm to find a rectangular dual of a planar triangulated graph. *Algorithmica*, 3:247–278, 1988.

[34] Y. T. Lai and S. M. Leinwand. Algorithms for floorplan design via rectangular dualization. *IEEE Transactions on Computer Aided Design*, 7(12):1278–1289, December 1988.

[35] A. B. Kahng. Classical floorplanning harmful? In *Proc. International Symposium on Physical Design*, pages 207–213, 2000.

[36] H. H. Chan, S. N. Adya, , and I. L. Markov. Are floorplan representations important in digital design? In *Proc. International Symposium on Physical Design*, pages 129–136, 2005.

[37] P. N. Guo, C. K. Cheng, , and T. Yoshimura. An O-tree representation of non-slicing floorplan and its applications. In *Proc. of the Design Automation Conf.*, pages 268–273, 1999.

[38] N. Viswanathan and C. C. Chu. Fastplace: Efficient analytical placement using cell shifting, iterative local refinement and a hybrid net model. In *Proc. International Symposium on Physical Design*, pages 26–33, 2004.

[39] F. Y. Young, C. C. N. Chu, W. S. Luk, and Y. C. Wong. Handling soft modules in general non-slicing floorplan using Lagrangian relaxation. *IEEE Transactions on Computer Aided Design*, 20(5):687–692, May 2001.

[40] E. F. Y. Young, C. C. N. Chu, and M. L. Ho. Placement constraints in floorplan design. *IEEE Transactions on Very Large-Scale Integrated Systems*, 12(7):735–745, July 2004.

[41] C. Lin, H. Zhou, and C. Chu. A revisit to floorplan optimization by Lagrangian relaxation. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 164–171, 2006.

[42] X. Tang, R. Tian, and M. D. F. Wong. Minimizing wire length in floorplanning. *IEEE Transactions on Computer Aided Design*, 25(9):1744–1753, September 2006.

[43] H.-C. Lee, Y.-W. Chang, and H. H. Yang. MB*-Tree: A multilevel floorplanner for large-scale building-module design. *IEEE Transactions on Computer Aided Design*, 26(8):1430–1444, August 2007.

[44] T. Ohtsuki, N. Sugiyama, and H. Kawanishi. An optimization technique for integrated circuit layout design. In *Proc. ICCST*, pages 67–68, Kyoto, Japan, 1970.

[45] R. H.J.M. Otten. What is floorplan? In *Proc. International Symposium on Physical Design*, pages 201–206, 2000.

[46] F. Y. Young, C. C. N. Chu, and Z. C. Shen. Twin Binary Sequences: A non-redundant representation for general non-slicing floorplan. *IEEE Transactions on Computer Aided Design*, 22(4):457–469, April 2003.

[47] J.-M. Lin and Y.-W. Chang. TCG: A transitive closure graph-based representation for non-slicing floorplans. In *Proc. of the Design Automation Conf.*, pages 764–769, 2001.

[48] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani. VLSI module placement based on rectangle-packing by the sequence-pair. *IEEE Transactions on Computer Aided Design*, 15(12):1518–1524, December 1996.

[49] C. Lin. *Incremental Mixed-Signal Layout Generation Concepts*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2002.

[50] CS2 version 4.3. Andrew Goldberg's network optimization library. `http://www.avglab.com/andrew/soft.html`.

[51] J. Fishburn and A. Dunlop. TILOS: A posynomial programming approach to transistor sizing. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 326–328, 1985.

[52] J. M. Shyu, A. Sangiovanni-Vincentelli, J. P. Fishburn, and A. E. Dunlop. Optimization-based transistor sizing. *IEEE Journal of Solid-State Circuits*, 23(2):400–409, April 1988.

[53] S. S. Sapatnekar, V. B. Rao, P. M. Vaidya, and S. M. Kang. An exact solution to the transistor sizing problem for CMOS circuits using convex optimization. *IEEE Transactions on Computer Aided Design*, 12:1621–1634, November 1993.

[54] H. Sathyamurthy, S. S. Sapatnekar, and J. P. Fishburn. Speeding-up pipelined circuits through a combination of gate sizing and clock skew optimization. *IEEE Transactions on Computer Aided Design*, 17(2):173–182, February 1998.

[55] C.-P. Chen, C. C. N. Chu, and D. F. Wong. Fast and exact simultaneous gate and wire sizing by lagrangian relaxation. *IEEE Transactions on Computer Aided Design*, 18(7):1014–1025, July 1999.

[56] V. Sundararajan, S. S. Sapatnekar, and K. K. Parhi. Fast and exact transistor sizing based on iterative relaxation. *IEEE Transactions on Computer Aided Design*, 21(5):568–581, May 2002.

[57] H. Tennakoon and C. Sechen. Gate sizing using lagrangian relaxation combined with a fast gradient-based pre-processing step. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 395–402, 2002.

[58] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[59] M. S. Bazaraa, H. D. Sherali, and C. M. Shetty. *Nonlinear Programming*. Wiley-Interscience, 3rd edition, 2006.

[60] W. C. Elmore. The transient response of damped linear networks with particular regard to wide-band amplifiers. *Journal of Applied Physics*, 19(1):55–63, January 1948.

[61] K. Kasamsetty, M. Ketkar, and S. S. Sapatnekar. A new class of convex functions for delay modeling and their application to the transistor sizing problem. *IEEE Transactions on Computer Aided Design*, 19(7):779–788, July 2000.

[62] W. Chuang, S. S. Sapatnekar, and I. N. Hajj. Timing and area optimization for standard-cell VLSI circuit design. *IEEE Transactions on Computer Aided Design*, 14(3):308–320, March 1995.

[63] O. Coudert. Gate sizing for constrained delay/power/area optimization. *IEEE Transactions on Very Large-Scale Integrated Systems*, 5(4):465–472, December 1997.

[64] S. Hu, M. Ketkar, and J. Hu. Gate sizing for cell library-based designs. In *Proc. of the Design Automation Conf.*, pages 847–852, 2007.

[65] C. Lin and H. Zhou. Clock skew scheduling with delay padding for prescribed skew domains. In *Proc. Asian and South Pacific Design Automation Conference*, 2007.

[66] R. T. Rockafellar. Ordinary convex programs without a duality gap. *Journal of Optimization Theory and Applications*, 7(3):143–148, March 1971.

[67] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Application*. Prentice Hall, 1993.

[68] A. V. Goldberg and T. Radzik. A heuristic improvement of the bellman-ford algorithm. *Applided Math. Let.*, 6:3–6, 1993.

[69] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[70] N. Shenoy and R. Rudell. Efficient implementation of retiming. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 226–233, 1994.

[71] N. Maheshwari and S. S. Sapatnekar. Efficient retiming of large circuits. *IEEE Transactions on Very Large-Scale Integrated Systems*, 6(1):74–83, March 1998.

[72] S. S. Sapatnekar and R. B Deokar. Utilizing the retiming-skew equivalence in a practical algorithm for retiming large circuits. *IEEE Transactions on Computer Aided Design*, 15(10):1237–1248, October 1996.

[73] H. Zhou. Deriving a new efficient algorithm for min-period retiming. In *Asia and South Pacific Design Automation Conference*, Shanghai, China, January 2005.

[74] D. P. Singh, V. Manohararajah, and S. D. Brown. Incremental retiming for FPGA physical synthesis. In *Proc. of the Design Automation Conf.*, pages 433–438, Anaheim, CA, June 2005.

[75] H. Lerchs and I.F. Grossmann. Optimum design of open-pit mines. *Trans C.I.M.*, 68:17–24, 1965.

[76] D. S. Hochbaum. A new–old algorithm for minimum-cut and maximum-flow in closure graphs. *Networks*, 37(4):171–193, July 2001.

[77] CAD Group at Politecnico di Torino. ITC'99 benchmarks (2nd release). `http://www.cad.polito.it/tools/itc99.html`.

[78] A. Hurst, A. Mishchenko, and R. Brayton. Fast minimum-register retiming via binary maximum-flow. In *Proc. FMCAD*, 2007.

[79] Berkeley Logic Synthesis and Verification Group. ABC–a system for sequential synthesis and verification. `http://www.eecs.berkeley.edu/~alanmi/abc/`.

[80] H. J. Touati and R. K. Brayton. Computing the Initial States of Retimed Circuits. *IEEE Transactions on Computer Aided Design*, 12(1):157–162, January 1993.

[81] G. Even, I. Y. Spillinger, and L. Stok. Retiming Revisited and Reversed. *IEEE Transactions on Computer Aided Design*, 15(3):348–357, March 1996.

[82] C. Lin and H. Zhou. An efficient retiming algorithm under setup and hold constraints. In *Proc. of the Design Automation Conf.*, San Francisco, CA, 2006.

[83] D. Blaauw, K. Chopra, A. Srivastava, and L. Scheffer. Statistical timing analysis: From basic principles to state of the art. *IEEE Transactions on Computer Aided Design*, 27(4):589–607, April 2008.

[84] S. H. Choi, B. C. Paul, and K. Roy. Novel sizing algorithm for yield improvement under process variation in nanometer technology. In *Proc. of the Design Automation Conf.*, pages 454–459, 2004.

[85] D. Sinha, N. Shenoy, and H. Zhou. Statistical timing yield optimization by gate sizing. *IEEE Transactions on Very Large-Scale Integrated Systems*, 14(10), October 2006.

[86] A. Davoodi and A. Srivastava. Variability driven gate sizing for binning yield optimization. In *Proc. of the Design Automation Conf.*, pages 959–964, 2006.

[87] R. J. B. Wets. Stochastic programs with fixed recourse: The equivalent deterministic program. *SIAM Review*, 16(3):309–339, July 1974.

[88] R. T. Rockafellar. Coherent approaches to risk in optimization under uncertainty. *INFORMS TutORials in Operations Research*, pages 38–61, 2007.

[89] D. B. Shmoys and C. Swamy. Stochastic optimization is (almost) as easy as deterministic optimization. In *Proc. IEEE Symposium on the Foundations of Computer Science*, pages 228–237, 2004.

[90] N. Immorlica, D. Karger, M. Minkoff, and V. S. Mirrokni. On the costs and benefits of procrastination: Approximation algorithms for stochastic combinatorial optimization problems. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 691–700, 2004.

[91] A. M. C. So, J. Zhang, and Y. Ye. Stochastic combinatorial optimization with controllable risk aversion level. *Lecture Notes in Computer Science*, 4110:224–235, August 2006.

[92] T. F. Chan, J. Cong, J. R. Shinnerl, K. Sze, and M. Xie. mPL6: Enhanced multilevel mixed-size placement. In *Proc. International Symposium on Physical Design*, pages 212–214, 2006.

[93] H C Shin and C. Hu. Thin gate oxide damage due to plasma processing. *Semicond. Sci. Technol.*, 11:463–473, 1996.

[94] R. Rakkhit, F. P. Heiler, P. Fang, , and C. Sander. Process induced oxide damage and its implications to device reliability of submicron transistors. In *IEEE 38th Annu. Int. Reliability Phys. Symp.*, pages 293–296, 1993.

[95] H. Watanabe, J. Komori, K. Higashitani, M. Sekine, , and H. Koyama. A wafer level monitoring method for plasma-charging damage using antenna pmosfet test structure. *IEEE Transactions on Semiconductor Manufacturing*, 10(2):228–232, May 1997.

[96] The MOSIS Service. Process–induced damage rules (otherwise known as "antenna rules")–general requirements. `http://www.mosis.org/Technical/Designrules/guidelines.html#antenna`.

[97] H. Shirota, T. Sadakane, M. Terai, , and K. Okazaki. A new router for reducing "antenna effect" in asic design. In *IEEE Custom Integrated Circuits Conf.*, pages 601–604, 1998.

[98] P. H. Chen, S. Malkani, C. Peng, , and J. Lin. Fixing antenna problem by dynamic diode dropping and jumper insertion. In *Proc. International Symposium Quality Electronic Design*, pages 275–282, 2000.

[99] L. Huang, X. Tang, H. Xiang, D. Wong, and I. Liu. A polynomial time optimal diode insertion/routing algorithm for fixing antenna problem. *IEEE Transactions on Computer Aided Design*, 23(1):141–147, January 2004.

[100] R. H.J.M. Otten, R. Camposano, , and P. R. Groeneveld. Design automation for deepsubmicron: present and future. In *Proc. DATE: Design Automation and Test in Europe*, pages 650–657, 2002.

[101] H. K.-S. Leung. Advanced routing in changing technology landscape. In *Proc. International Symposium on Physical Design*, pages 118–121, 2003.

[102] T. Y. Ho, Y. W. Chang, , and S. J. Chen. Multilevel routing with antenna avoidance. In *Proc. International Symposium on Physical Design*, pages 34–40, 2004.

[103] D. Wu, J. Hu, , and R. Mahapatra. Coupling aware timing optimization and antenna avoidance in layer assignment. In *Proc. International Symposium on Physical Design*, pages 20–27, 2005.

[104] B. Y. Su and Y. W. Chang. An exact jumper insertion algorithm for antenna effect avoidance/fixing. In *Proc. of the Design Automation Conf.*, pages 325–328, 2005.

[105] B. Y. Su, Y. W. Chang, and J. Hu. An optimal jumper insertion algorithm for antenna avoidance/fixing on general routing trees with obstacles. In *Proc. International Symposium on Physical Design*, pages 56–63, 2006.

[106] L. J. Stockmeyer. Optimal orientations of cells in slicing floorplan designs. *Information and Control*, 57(2-3):91–101, May/June 1983.

[107] L. P.P.P. van Ginneken. Buffer placement in distributed rc-tree network for minimal elmore delay. In *Proc. Intl. Symposium on Circuits and Systems*, pages 865–868, 1990.

[108] A. B. Kahng, I. I. Mandoiu, and A. Zelikovsky. Highly scalable algorithms for rectilinear and octilinear steiner trees. In *Proc. Asia and South Pacific Design Automation Conference*, pages 827–833, 2003.

[109] H. Zhou. Efficient steiner tree construction based on spanning graphs. *IEEE Transactions on Computer Aided Design*, 23(5):704–710, May 2004.