

© 2015 Pei-Ci Wu

NEW METHODS FOR ELECTRONIC DESIGN AUTOMATION

BY

PEI-CI WU

DISSERTATION

Submitted in partial fulfillment of the requirements  
for the degree of Doctor of Philosophy in Electrical and Computer Engineering  
in the Graduate College of the  
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Doctoral Committee:

Professor Martin D. F. Wong, Chair and Director of Research  
Associate Professor Deming Chen  
Professor Wen-Mei W. Hwu  
Professor Robin A. Rutenbar

# ABSTRACT

As the semiconductor technology marches towards the 14nm node and beyond, EDA (electronic design automation) has rapidly increased in importance with ever more complicated modern integration circuit (IC) designs. This presents many new issues for EDA including design, manufacturing, and packaging. Challenging EDA problems in these three domains are studied in this dissertation.

Timing closure, which aims to satisfy the timing constraints, is always a key problem in the physical design flow. The challenges of timing closure for IC designs keep increasing as the technology advances. During the timing optimization process, buffers can be used to speed up the circuit or serve as delay elements. In this dissertation, we study the hold-violation removal problem for a circuit-level design. Considering the challenges of industrial designs, discrete buffer sizes, accurate timing models/analysis and complex timing constraints make the problem difficult and time-consuming to solve. In this dissertation, a linear programming-based methodology is presented. In the experiment, our approach is tested on industrial designs, and is incorporated into the state-of-the-art industrial optimization flow.

While buffers can help fix hold-time violations, they also increase the difficulty of routability and the utilization of a design. And the larger area of cells contributes larger leakage power, while power is an increasing challenge as the technology advances. Therefore, in Chapter 3, we study the buffer insertion problem that is to find which buffers to be inserted in order to meet the timing constraints, meanwhile minimizing the total area of inserted buffers. Several approaches are presented. We test the proposed approaches on the industrial designs, and the machine learning based approach shows better results in terms of quality and runtime.

Aerial image simulation is a fundamental problem in the regular lithography-related process. Since it requires a huge amount of mathematical computa-

tion, an efficient yet accurate implementation becomes a necessity. In the literature, GPU or FPGA has successfully demonstrated its potential with detailed tuning for accelerating aerial image simulation. However, the advantages of GPU or FPGA to CPU are not solid enough, given that the careful tuning for the CPU-based method is missing in the previous works, while the recent CPU architectures have significant modifications towards high performance computing capabilities. In this dissertation, we present and discuss several algorithms for the aerial image simulation on multi-core SIMD CPU. Our experimental results show that the performance on the multi-core SIMD CPU is promising, and careful CPU tuning is necessary in order to exploit its computing capabilities.

Since the constantly evolving technology continues to push the complexity of package and printed circuit board (PCB) design to a higher level, nowadays a modern package can contain thousands of pins. On the other hand, the size of a package is still kept to a minimum. This makes the footprint of such a package on a PCB a very dense pin grid, such that staggered pin arrays have been introduced for modern designs with high pin density. Although some studies have been done on escape routing for hexagonal arrays, the hexagonal array is only a special kind of staggered pin array. There exist other kinds of staggered pin arrays in current industrial designs, and the existing works cannot be extended to solve them. In this dissertation, we study the escape routing problem on staggered pin arrays. Network flow models are proposed to correctly model staggered pin arrays, and our proposed algorithm is proved optimal.

The high complexity of PCB design makes the manual design of PCBs an extremely time-consuming and error-prone task. An auto-router for PCBs would improve design productivity tremendously since each board takes about 2 months to route manually. This dissertation focuses on a major step in PCB routing called bus planning. In the bus planning problem, we need to simultaneously solve the bus decomposition, escape routing, layer assignment and global bus routing. This problem was only partially addressed by Kong et al. (2009). In this dissertation, we present an ILP-based solution to the entire bus planning problem. We apply our bus planner to an industrial PCB (with over 7000 nets and 12 signal layers) which was previously successfully routed manually, and compare with a state-of-the-art industrial internal tool where the layer assignment and global bus routing are based on the algo-

rithm proposed by Kong et al. (2009). Experimental results show that our bus planner successfully achieves better routability.

*To my parents and my family, for their love and support.*

# ACKNOWLEDGMENTS

I owe my deepest gratitude to my adviser, Prof. Martin D. F. Wong. His encouragement, guidance and support have constantly led me forward from the initial to the final level of my thesis subjects.

Besides my adviser, I thank the rest of my doctoral committee, Prof. Deming Chen, Prof. Robin A. Rutenbar, and Prof. Wen-Mei W. Hwu, for their insightful comments and constructive suggestions. Their invaluable opinions have significantly improved the quality of this dissertation.

I also want to express my grateful thanks to Mentor Graphics Inc. for funding and supporting my research on hold optimization. In particular, I thank Mr. Ivailo Nedelchev and Dr. Sarvesh Bhardwaj for sharing their precious experience and knowledge for my research and experiments.

All the members of Prof. Wong's research team who have overlapped with me made my life in UIUC very enjoyable. I thank Dr. Hui Kong, Dr. Tan Yan, Dr. Lijuan Luo, Dr. Hongbo Zhang, Dr. Qiang Ma, Dr. Yuelin Du, Dr. Ting Yu, Ms. Leslie Hwang, Mr. Zigang Xiao, Mr. Haitong Tian, and Mr. Tsung-Wei Huang for all the stimulating discussions and the seamless collaborations we had, as well as their help in my study and life.

Last but not least, I thank my dear friends for whatever help they kindly provided throughout my Ph.D. study. I thank my sisters and brother for their endless love and support. I thank my parents for raising me up, and providing me whatever they have to let me pursue my goal.

# TABLE OF CONTENTS

LIST OF TABLES . . . . .	ix
LIST OF FIGURES . . . . .	x
CHAPTER 1 INTRODUCTION . . . . .	1
CHAPTER 2 TIMING CLOSURE: HOLD OPTIMIZATION . . . . .	4
2.1 Introduction . . . . .	4
2.2 Background . . . . .	5
2.3 Linear Programming Based Optimization . . . . .	7
2.3.1 Basic Linear Programming Formulation . . . . .	7
2.3.2 Complex Timing Constraints . . . . .	12
2.3.3 Graph Reduction . . . . .	13
2.4 Bottom-Up Buffer Insertion . . . . .	14
2.4.1 Dynamic Programming Based Algorithm . . . . .	14
2.4.2 Bottom-up Methodology . . . . .	15
2.5 Our Optimization Flow . . . . .	16
2.6 Experimental Results . . . . .	17
2.7 Conclusion . . . . .	20
CHAPTER 3 AN AREA-AWARE BUFFER INSERTION FOR HOLD VIOLATIONS . . . . .	21
3.1 Introduction . . . . .	21
3.2 Background . . . . .	22
3.2.1 Problem Formulation . . . . .	22
3.2.2 The Optimization Flow . . . . .	24
3.3 An Optimal Algorithm . . . . .	24
3.4 A Heuristic Algorithm . . . . .	25
3.5 A Machine Learning Based Approach . . . . .	27
3.5.1 Data Preparation . . . . .	28
3.5.2 Learning: Gradient Descent and Logistic Regression . . . . .	30
3.5.3 The Algorithm for the Buffer Insertion . . . . .	31
3.6 Experimental Results . . . . .	32
3.7 Conclusion . . . . .	33



CHAPTER 4	AERIAL IMAGE SIMULATION ON MULTI-CORE	
	SIMD CPU . . . . .	36
4.1	Introduction . . . . .	36
4.2	Background . . . . .	38
4.2.1	Aerial Image Simulation . . . . .	38
4.2.2	Multi-Core SIMD CPU Programming . . . . .	41
4.3	Implementation Methods on Multi-Core SIMD CPU . . . . .	42
4.3.1	An Improved Approach . . . . .	43
4.3.2	Four Lookup Tables per SSE Approach . . . . .	46
4.3.3	Four Pixels per SSE Approach . . . . .	48
4.3.4	Multi-Threading . . . . .	51
4.4	Experimental Results . . . . .	52
4.5	Conclusion . . . . .	55
CHAPTER 5	ESCAPE ROUTING ON STAGGERED PIN ARRAYS	58
5.1	Introduction . . . . .	58
5.2	Problem Formulation . . . . .	62
5.3	Our Network Flow Models . . . . .	63
5.3.1	Modeling Equation (5.5) . . . . .	63
5.3.2	Proof of Theorem 5.1 . . . . .	65
5.3.3	Modeling Equation (5.4) . . . . .	69
5.3.4	Proof of Theorem 5.2 . . . . .	71
5.3.5	Escape Routing Algorithm . . . . .	72
5.4	Experimental Results . . . . .	73
5.5	Conclusion . . . . .	74
CHAPTER 6	AN ILP-BASED AUTOMATIC BUS PLANNER	
	FOR DENSE PCBS . . . . .	76
6.1	Introduction . . . . .	76
6.2	Problem Formulation . . . . .	80
6.3	Methodology . . . . .	81
6.4	Candidate Routes Generation . . . . .	83
6.4.1	Escape Routing and Bus Decomposition . . . . .	83
6.4.2	Global Routing . . . . .	84
6.5	Layer Assignment . . . . .	86
6.5.1	ILP Formulation . . . . .	87
6.5.2	Algorithm . . . . .	88
6.6	Resolving Congestion . . . . .	89
6.7	Postprocessing . . . . .	90
6.8	Experimental Results . . . . .	91
6.9	Conclusion . . . . .	93
CHAPTER 7	CONCLUSIONS AND FUTURE WORK . . . . .	94
REFERENCES	. . . . .	96

# LIST OF TABLES

2.1	Summary and the initial timing statistics of the industrial designs . . . . .	18
2.2	Results of our proposed approach . . . . .	18
2.3	Results of the industrial hold optimization flow . . . . .	19
2.4	Results of our proposed approach together with the industrial hold optimization flow . . . . .	19
3.1	Summary and the initial timing statistics of the industrial designs . . . . .	33
3.2	Experimental results (normalized) . . . . .	34
4.1	Data set for our experiments . . . . .	53
4.2	Configuration setups and the options . . . . .	53
4.3	Runtime comparison. Unit is second. . . . .	56
4.4	Speedup comparison. Speedup = runtime of base / runtime of the method. . . . .	57
5.1	Experimental results . . . . .	73
6.1	An example bus decomposition process for a bus $b$ . . . . .	83
6.2	Each stage of our planner and other methods, and the completion rate on a state-of-the-art industrial PCB . . . . .	92

# LIST OF FIGURES

2.1	Buffer insertion: (a) insert buffer at the output of cell $X$ ; (b) insert buffer at one of the inputs of cell $Z$ . . . . .	8
2.2	Example of missing setup constraints. Pin $a$ in (a) and pin $d$ in (b) both have positive setup slacks and positive hold slacks, so they are excluded in the graph that is formed by solid lines, but their corresponding setup timing constraints still have to be included into the linear program. . . . .	11
2.3	The graph is reduced to four paths and the edges between these paths. . . . .	13
2.4	The flow of our approach. . . . .	17
4.1	Rectangle decomposition. The impact of $R((x_1, y_1) - (x_2, y_2))$ on the central pixel (the red square) can be calculated by looking up the impacts of the four shaped rectangles together[11].	39
4.2	Rectangle (blue color) extending outside the lookup table in (a) is truncated at the boundary of the lookup table in (b).	40
4.3	Example of a rectangle (blue rectangle) and its correspond- ing impact region (green zone). . . . .	40
4.4	An illustration of using SSE so that all four elements inside the register are computed in parallel. . . . .	42
4.5	An example of computing the relative coordinates for the consecutive pixels. (a) The relative positions of the left- bottom corner of the blue rectangle for the two consecutive pixels $a$ and $b$ are shown in (b) and (c), respectively. . . . .	43
4.6	Impact of the blue rectangle on the pixel (the red dot): (a) $T(x_2, y_2)$ ; (b) $T(x_2, y_2) - T(x_1, y_2)$ ; (c) $T(x_2, y_2) - T(x_2, y_1)$ . . .	44
4.7	The impact region is decomposed into four regions: $Z_1$ (the green zone), $Z_2$ (the yellow zone), $Z_3$ (the gray zone), and $Z_4$ (the brown zone). . . . .	44
4.8	Each SSE instruction updates four images at pixel $(x, y)$ for four corresponding lookup tables. . . . .	47
4.9	Rearrange the data layouts of the images and lookup ta- bles, so one SSE instruction can load/store the register from/to memory. . . . .	47

4.10	Each SSE instruction updates four continuous pixels that start from $(x, y)$ for the $k$ -th lookup table. . . . .	48
4.11	An example shows the memory address of the consecutive pixels are not aligned with the 16-byte boundary, where green squares represent the memory address for a row of image pixels in the impact region. . . . .	49
4.12	The address of the image is not aligned with the address of the lookup table, while their memory accesses are both aligned with 16-byte boundary. . . . .	50
4.13	An example that shows the four possible behaviors regarding the alignment between an image and a lookup table. . . .	50
5.1	An example of a traditional square pin array (left) and the enlarged view of a tile (right). <i>O-cap</i> and <i>D-cap</i> are two capacity constraints within a tile. . . . .	59
5.2	An example of a $4 \times 8$ staggered pin array (left) and the enlarged view of a tile (right). $\Delta x$ is the distance in the x-dimension between two adjacent columns of pins, and $\Delta y$ is the distance in the y-dimension between two adjacent rows of pins. . . . .	59
5.3	Examples of three types of staggered pin array that are available in industrial PCB designs. (a) A hexagonal array where the dashed polygon is a hexagon; (b) a rotated square pin array where the dashed polygon is a rotated square; (c) a staggered pin array that is neither a hexagonal array nor a rotated square pin array. In this example, <i>H-cap</i> = 1 and <i>V-cap</i> = 3. . . . .	60
5.4	Our network model for Equation (5.5). (a) The capacity of each edges; (b) a tile; (c) a 1/2-tile along the bottom boundary of the array; (d) a 1/2-tile along the right boundary of the tile; (e) a 1/4-tile in the northeast corner of the array. <i>t</i> is the super sink. . . . .	64
5.5	The properties of a directed routing <i>R</i> with the minimum crossings with the tile boundaries. (a) Property 1; (b) Property 2. . . . .	66
5.6	Reconnect the wires to solve a routing that violates Property 1. The number of crossings is further reduced while the legality of the routing is still maintained. . . . .	66
5.7	Shift a wire to its neighboring tile to solve a routing that violates Property 2. The number of crossings is further reduced without affecting the legality of the routing. . . . .	67
5.8	Two wires are reconnected to solve a routing that violates Property 2. The number of crossings is further reduced without affecting the legality of the routing. . . . .	67

5.9	(a)-(e) Analysis of the possible flow configurations on the inside-tile network for Equation (5.5). The dotted lines represent a min-cut in the inside-tile network. . . . .	68
5.10	Our network model for Equation (5.4). (a) The capacity of each edges; (b) a tile; (c) a 1/2-tile along the bottom boundary of the array; (d) a 1/2-tile along the right boundary of the tile; (e) a 1/4-tile in the northeast corner of the array. $t$ is the super sink. . . . .	70
5.11	(a)-(e) Analysis of the possible flow configurations on the inside-tile network for Equation (5.4). The dotted lines represent a min-cut in the inside-tile network. $h = \lfloor H\text{-cap}/2 \rfloor$ and $v = \lfloor V\text{-cap}/2 \rfloor$ . . . . .	71
5.12	A flow solution (left) is transformed into an octilinear routing(right). The zigzag wires obtained from the octilinear routing style effectively utilize the routing area. . . . .	72
5.13	Routing solution of <i>case_10</i> . The dashed square is drawn on top of the result to show that routing uses up almost all routing resources. . . . .	74
6.1	One layer of an example bus planning solution. . . . .	77
6.2	If the escape directions of the buses in Figure 6.1 are predetermined, and bus 3 in the lower left component is decided to escape to the left boundary instead of the bottom boundary, the set of buses can no longer be routed on one layer. . . . .	78
6.3	An example bus planning problem, where bus 5 is decomposed into $5_a$ and $5_b$ , and buses in (a) are assigned into two layers as shown in (b) and (c) with no conflicts. . . . .	82
6.4	(a) The Hanan grid and (b) the routing graph constructed for an example PCB with two components $C_1$ and $C_2$ . . . . .	84
6.5	The dynamic routing graphs in a Hanan grid cell. (a) and (b) give the routing graphs before and after a bus is routed. After the edge $(a, c)$ is occupied by the bus, four new vertices $(a1, c1, a2$ and $c2)$ and their corresponding new edges are added. The weight of the new edges $(a1, c1)$ , $(a1, c2)$ , $(a2, c1)$ , and $(a2, c2)$ is equal to the weight of the old edge $(a, c)$ . . . . .	85
6.6	By traversing the boundaries of a Hanan grid cell, (a) the routes passing through this cell can be transformed to (b) a set of one-directional intervals. The two segments $(a, f)$ and $(c, g)$ in (b) overlap with each other, indicating that the two corresponding routes conflict with each other in (a). . . . .	86

- 6.7 An example of the critical cuts considering empty components on one layer. Shaded rectangles denote the buses that are assigned to this layer. Since  $C_2$  has no buses being assigned,  $C_2$  is an empty component on this layer. Thick lines denote the critical cuts starting from the corner  $s$ , which ignore/cross  $C_2$  in order to capture the routing space within  $C_2$ . 90

# CHAPTER 1

## INTRODUCTION

As the semiconductor technology marches towards the 14nm node and beyond, EDA (electronic design automation) has rapidly increased in importance with ever more complicated modern integration circuit (IC) designs. This presents many new issues for EDA including design, manufacturing, and packaging. My dissertation covers challenging EDA problems in these domains.

The challenges of timing, power, and area for IC designs keep increasing as the IC technology advances. Timing closure, which is to satisfy the timing constraints, is always a key problem in the physical design flow. Transforms such as gate sizing and buffer insertion etc. ([1]) are known to be useful for fixing timing violations. These transforms not only can help timing, but they also impact power and area of IC designs. So, intelligently adopting these transforms for timing closure is an important task. There are two kinds of timing violations: setup violations and hold violations. The setup violation removal problem is studied extensively ([2, 3, 4, 5, 6, 7, 8]). However, few works discuss the hold violation fixing problem. Typically, hold violations are addressed after setup optimization has been performed. Thus, hold optimization has to consider setup constraints as well. Hold violations can be fixed by inserting delays into wires. As mentioned in [9], the main challenges in modern industrial designs such as discrete cell sizes (i.e. discrete buffer sizes for hold optimization), accurate cell timing models, complex timing constraints, etc., have to be considered during optimization, which makes the existing works not applicable on modern designs due to the lack of consideration of the challenges on modern designs. In Chapter 2, we study the problem of hold-violation removal for a circuit level design. We present an optimization flow to solve hold violations by inserting buffers as delay elements. We propose a linear programming-based approach that captures the different delays introduced between setup constraints and hold-time constraints

due to different cell libraries specified for the constraints. To the best of our knowledge, this is the first work that identifies this issue and models it into the linear programming model. In the experiment, we test our proposed approach on industrial circuit-level designs, and then run with a state-of-the-art industrial hold optimization flow, and the results show that our algorithms perform better in terms of hold violations and runtime. Then, in Chapter 3, we study the min-cost buffer insertion problem for hold violation, that is to minimize the area of the inserted buffers while the timing constraints are met. Several approaches are presented in the chapter, and to the best of our knowledge, this is the first work that adopts machine learning for buffer insertion. We test the proposed approaches on the industrial designs, and show better results than the state-of-the-art industrial hold optimization.

As semiconductor devices shrink, the lithography technology becomes increasingly complicated. To print correctly on a mask from the wafer is no longer a trivial task. Aerial image simulation, that is to simulate the light density on top of the wafer for given illumination conditions, is considered an essential step in design for manufacturability (DFM) process. Aerial image simulation contains a huge number of numerical computations that makes it a timing consuming task. Therefore, an efficient yet accurate aerial image simulation becomes a necessity. Recently, the FPGA-based approach [10] and GPU-based approach [11] are both proposed to accelerate this polygon-based method with considerable speedup reported. However, compared to the FPGA-based or GPU-based approach, not many efforts were made to improve the performance of the CPU baseline programming in the previous works. Furthermore, careful tuning for multi-core SIMD CPU architecture for the CPU-based approach is missing in the previous works. In Chapter 4, we first present a more efficient approach by further optimizing the basic polygon-based approach in terms of the total amount of computation. Then, several implementations are proposed to accelerate the approach with multi-core SIMD CPU. In our experiments, with a hex-core SIMD CPU, our fastest method achieves up to 73X speedup over the baseline serial approach, while the GPU-based approach in [11] achieves up to 34X speedup. Our results reveal that with explicit tunings, the multi-core CPU-based approach can achieve a considerable speedup.

As IC technology advances, the package size keeps shrinking while the pin count of a package keeps increasing. Nowadays, a dense PCB contains



thousands of pins [12]. These complicated designs lead to very dense designs with high pin counts, such that staggered pin arrays have been introduced to enable such designs with high pin density. Escape routing is an important problem in package and printed circuit board (PCB) design. Its purpose is to route from specific pins inside a pin array to the boundary of the array. Currently, the escape routing problem for staggered pin arrays is solved manually in industry. The problem of escape routing on staggered pin array is studied in Chapter 5. In the chapter, we first introduce different types of staggered pin arrays and show that previous work [13] can only solve one type of staggered pin arrays. Based on different configurations of staggered pin arrays, network flow models are presented to correctly model the capacities of different types of staggered pin arrays. Theorems and proofs for the network flow model are also provided in the chapter.

The high complexity of PCB design makes manual design of PCBs an extremely time-consuming and error-prone task. An auto-router for PCBs would improve design productivity tremendously since each board takes about 2 months to route manually. Typically, in the manually routed designs, the nets are grouped as buses, and the nets within the same bus are expected to be routed together [14, 15, 16, 17, 18]. So, the bus planning, which is to simultaneously solve the bus decomposition, escape routing, layer assignment and global bus routing, is an important yet difficult step. The bus planning problem of PCBs has been studied by a number of previous works ([16, 19, 14, 18, 17, 15]); however, none of them can provide a complete bus planner. In Chapter 6, we study the problem of automatic bus planning. In the chapter, we present an ILP-based boards level bus planner, which considers bus decomposition, escape routing and global routing simultaneously during the layer assignment stage. We test our bus planner on a dense industrial board, and compare with the state-of-the-art industrial internal bus planner. The results show that our bus planner outperforms the industrial one in terms of both solution quality and runtime.

# CHAPTER 2

## TIMING CLOSURE: HOLD OPTIMIZATION

### 2.1 Introduction

Timing closure, which is to satisfy the timing constraints, is a key problem in the physical design domain. Timing constraints generally consist of setup and hold-time constraints. Setup (long-path) constraints ensure that the signal transitions do not arrive too late, while hold-time (short-path) constraints ensure that the signal transitions do not arrive too early. There are several techniques for timing optimization such as buffer insertion, gate sizing, and netlist restructuring [1]. Buffers can be used to speed up the circuit or serve as delay elements. In this chapter, we focus on the problem of fixing hold violations by inserting buffers as delay elements.

Typically, hold violations are addressed after setup optimization has been performed. Thus, hold optimization has to consider setup constraints as well. Hold violations can be fixed by inserting delays into wires. The problem of minimizing the inserted delays for removing hold violations has been studied extensively, and the delay insertion technique can also be used in clock period minimization [20, 21, 22, 23, 24]. The delay insertion technique assumes that the inserted delay can be realized by unit-delay elements. Huang et al. [21] solve hold violation by minimizing the number of buffer insertions, and recently, Tu et al. [24] fixed hold violations under different power modes for ultra-low voltage designs; both works assume that any inserted delay can be realized by unit-delay elements. However, it is known that in practical designs, unit-delay elements do not always exist, i.e., discrete types of buffers/inverters have to be considered during delay insertion.

However, it is difficult to apply these existing approaches on modern industrial designs. As mentioned in [9], the main challenges in modern industrial designs such as discrete cell sizes (i.e. discrete buffer sizes for hold optimiza-

tion), accurate cell timing models, complex timing constraints, etc., have to be considered. Moreover, cell libraries specified for the setup constraints and the hold-time constraints are usually different in modern industrial designs, meaning that delay of a buffer caused by hold-time constraints could be different from that for a buffer caused by setup constraints. Therefore, hold optimization that can consider these challenges is a necessity.

In this chapter, we present an optimization flow to solve hold violations by inserting buffers as delay elements. We first propose a linear programming-based approach that captures the different delays introduced between setup constraints and hold-time constraints due to different cell libraries specified for the constraints. To the best of our knowledge, this is the first work that identifies this issue and models it into the linear programming model. We also model complex timing constraints. Then, we use graph reduction to reduce the size of the linear programming, thereby reducing the running time. Finally, a bottom-up buffer insertion algorithm is proposed to realize the solution of the linear programming by inserting buffers. In the experiment, we test our proposed approach on industrial circuit-level designs, and then run with a state-of-the-art industrial hold optimization flow, and the results show that our algorithms perform better in terms of hold violations and runtime.

The rest of the chapter is organized as follows: Section 2.2 introduces some background information. Our linear programming based approach and the buffer insertion algorithm are presented in Sections 2.3 and 2.4. Section 2.5 gives the overall optimization flow based on our approaches. Experimental results are then presented in Section 2.6. Finally, Section 2.7 gives the concluding remarks.

## 2.2 Background

Consider a design  $D$  that contains a set of combinational circuits  $C$ , a set of pins  $P$  on the combinational circuits, and a set of nets  $N$  that define the connectivity between the pins. Let  $PI$  denote the primary inputs and the outputs to the sequential cells (e.g., flip-flops and latches), and  $PO$  denote the primary outputs and the inputs to the sequential cells. Furthermore, a set of buffer cells  $B$  is defined in the standard cell library, each of which has

different area and different technology parameters.

The design will then be enforced with a collection of timing constraints, e.g., setup constraints and hold-time constraints. The setup constraints and the hold-time constraints are also called the long-path constraints and the short-path constraints, respectively. Furthermore, the standard cell library used by the setup constraints could be different from the one used by the hold-time constraints. Thus, the timing information for these two constraints has to be considered individually. For the setup constraints, each pin  $p$  in  $P$  has a required arrival time  $setup\_req_p$ , and an actual arrival time  $setup\_aat_p$ . The slack w.r.t. the setup constraints at pin  $p$  is then defined as

$$setup\_slack_p = setup\_req_p - setup\_aat_p.$$

Similarly, for the hold-time constraints, each pin  $p$  in  $P$  has a required arrival time  $hold\_req_p$  and an actual arrival time  $hold\_aat_p$ . Then, the hold slack is defined as

$$hold\_slack_p = hold\_aat_p - hold\_req_p.$$

Note that the required time and actual arrival time are computed and given by the timer engine. Negative setup slacks and negative hold slacks indicate setup violations and hold violations, respectively. For timing closure, the design must achieve no timing violations. Let  $TNS$  denote the absolute value of the total negative setup slacks of all the pins in  $PO$  and  $THS$  denote the absolute value of the total negative hold slacks of all the pins in  $PO$ .

Hold violations are typically addressed after setup optimization has been performed.  $TNS$  must not worsen during hold-violation removal, otherwise another pass of setup optimization has to be applied, thereby causing long design cycle. While buffers are inserted as delay elements to fix hold violations, the inserted buffers also increase the area and the power consumption of the design. Therefore, the problem of buffer insertion for hold-violation removal can be defined as follows: *Given a design and a buffer library, find a buffering solution such that  $THS$  and the cost of buffering (i.e. area and power consumption) are both minimized while  $TNS$  does not worsen.*

Here, there are two things we would like to point out. First, as stated before, buffers can also be used to reduce wire delay, so setup slacks can be improved which would help the subsequent hold optimization. However,

doing so is beyond the discussion of this chapter, which focuses on using buffers as delay elements. Second, buffering is costly in terms of area and power consumption, so over-buffering should be avoided, and along with buffering, some other techniques such as gate sizing and netlist restructuring can also help resolve hold violations. Thus, it is not practical to fix all hold violations only by buffer insertion. In this chapter, we will demonstrate the effectiveness of our proposed approach by running our approach together with other hold optimization techniques in the experiment.

An industrial timer is used as an underlying timing engine to provide timing information such as the cell delays, the required times, and the actual arrival times of the pins w.r.t. the setup constraints and the hold-time constraints, respectively. The cell delay model we used is based on the lookup table where the two inputs are slew and load capacitance, respectively, where the slew is also based on the slew lookup table.

## 2.3 Linear Programming Based Optimization

In this section, we first tackle the hold-violation removal problem as a problem of inserting delay into wires to remove hold violations. Then, an approach to use buffers to realize the required delays will be presented in the later subsection. For this delay insertion problem, a linear programming formulation that captures both the setup constraints and the hold-time constraints is presented first. Then, we extend such formulation for the complex timing constraints. Finally, a graph-reduction approach is proposed to reduce the size of the linear programming formulation while the reduced linear programming remains optimal.

### 2.3.1 Basic Linear Programming Formulation

The input to our linear programming is a combinational circuit  $C^*$  s.t. for any pin  $p$  of  $C^*$ ,  $hold\_slack_p < 0$  and  $setup\_slack_p > 0$ . Obviously, for the hold-violation removal problem, only the pins with negative hold slacks have to be considered. Furthermore, as  $TNS$  has to be maintained, it is natural that only those pins with positive setup slacks are allowed to insert delays. Therefore, we are only interested in the pins with negative hold slacks

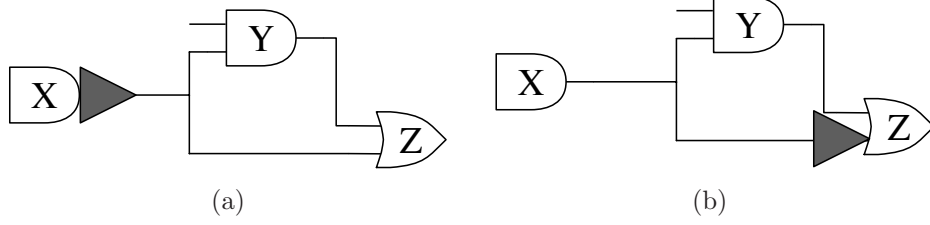


Figure 2.1: Buffer insertion: (a) insert buffer at the output of cell  $X$ ; (b) insert buffer at one of the inputs of cell  $Z$ .

and positive setup slacks, and a depth-first-search can easily get  $C^*$  from a combinational circuit.

Given a combinational circuit  $C^*$ ,  $C^*$  can then be represented as a directed acyclic graph  $G(V, E)$ , where  $V$  is the pins of  $C^*$  and  $(i, j) \in E$  represents an edge from pin  $i$  to pin  $j$ . Let  $I$  denote the zero in-degree pins, i.e. sources in  $G$ , and  $O$  denote the zero out-degree pins, i.e. sinks in  $G$ . Then, for each pin  $i$  in  $V$ , three real-value variables,  $x_i$ ,  $ha_i$ , and  $sa_i$ , are introduced in our linear programming model, respectively.  $x_i$  represents delays inserted at pin  $i$  for hold-time constraints.  $ha_i$  and  $sa_i$  represent the arrival time at pin  $i$  for hold-time constraints and setup constraints, respectively. Hold-time constraints can then be formulated as follows:

$$ha_i = hold\_aat_i + x_i \quad \forall i \in I \quad (2.1)$$

$$ha_j \leq ha_i + hd_{i \rightarrow j} + x_j \quad \forall (i, j) \in E \quad (2.2)$$

$$ha_i \geq hold\_req_i \quad \forall i \in O \quad (2.3)$$

where  $hold\_aat_i$ ,  $hold\_req_i$ , and  $hd_{i \rightarrow j}$  are all constant and given from the timer.  $hd_{i \rightarrow j}$  is set to the delay of edge  $(i, j)$ .

Setup constraints can be formulated in a similar way. However, we know that the inserted delay has to be realized by buffers and the delay caused by a buffer under setup constraints could be different from one caused by a buffer under hold-time constraints. Thus, it is not reasonable that  $x_i$  also represents inserted delays for setup constraints. To resolve this, a buffer library characterization is necessary in order to get an empirical ratio  $\alpha$  such that

$$\alpha_i = \frac{\hat{x}_i}{x_i}$$

where  $\hat{x}_i$  represents the corresponding delay introduced by the setup con-

straints while delay  $x_i$  is inserted at pin  $i$  for hold-time constraints. Therefore, we can use  $\alpha \cdot x$  for the delay caused by setup constraints in the linear programming model.

For the buffer library characterization, the delay of a buffer at a pin is calculated as if the buffer was inserted right next to the pin, and we assume that the buffer only affects the driver cell and the sink cells of the buffer. Suppose a buffer  $b \in B$ . Let  $d_{i,b}$  represent the delay of buffer  $b$  when inserting buffer  $b$  at pin  $i$ . For any driver/sink cell  $C$  of pin  $i$ ,  $\Delta d_C^b$  denotes the difference of the delays of the cell  $C$  between before and after inserting buffer  $b$  at pin  $i$ . Figure 2.1(a) shows a buffer  $b$  inserted at the output pin  $i$  of cell  $X$ , so the delay introduced by inserting this buffer is  $d_{i,b} + \Delta d_X^b + \Delta d_Y^b + \Delta d_Z^b$ . In Figure 2.1(b), a buffer  $b$  is inserted at the inputs  $i$  of cell  $Z$ , so the delay introduced by this buffer is  $d_{i,b} + \Delta d_X^b + \Delta d_Z^b$ . It should be noted that inserting a buffer at the input of cell  $Z$  as shown in Figure 2.1(b) impacts the slew on the output of  $X$ , and this change in slew can lead to some change in the delay of cell  $Y$ . For simplicity, we do not model this effect. Also, note that this introduced delay calculation has to be done for hold-time constraints and setup constraints, respectively. Thus, we use  $hd$  for hold-time constraints and  $sd$  for setup constraints in the following.  $HID_{i,b}$  denotes the introduced hold delay of inserting buffer  $b$  at pin  $i$ , and it is defined as follows:

$$HID_{i,b} = hd_{i,b} + \Delta hd_{driver\_cell(i)}^b + \sum_{c:sink\_cells(i)} \Delta hd_c^b$$

Similarly,  $SID_{i,b}$  denotes the introduced setup delay of inserting buffer  $b$  at pin  $i$ , and is defined as follows:

$$SID_{i,b} = sd_{i,b} + \Delta sd_{driver\_cell(i)}^b + \sum_{c:sink\_cells(i)} \Delta sd_c^b$$

Note that the delay is calculated based on the lookup table and the output slews of those affected cells are updated accordingly based on the slew table.

Since there are  $|B|$  types of buffers in the buffer library, it is difficult to get an accurate ratio of setup delay over hold delay and costly to enumerate all combinations of buffers for all the pins. Therefore, our buffer characterization enumerates only one type of buffer at a time, and sets  $\alpha$  as the maximum ratio during the enumeration. The empirical ratio  $\alpha_i$  at a pin  $i$  is then set as

follows:

$$\alpha_i = \max_{b \in B} \left\{ \frac{SID_{i,b}}{HID_{i,b}} \right\} \quad (2.4)$$

Then, we can have the following setup constraints:

$$sa_i = setup\_aat_i + \alpha_i \cdot x_i \quad \forall i \in I \quad (2.5)$$

$$sa_j \geq sa_i + sd_{i \rightarrow j} + \alpha_j \cdot x_j \quad \forall (i, j) \in E \quad (2.6)$$

$$sa_i \leq setup\_req_i \quad \forall i \in O \quad (2.7)$$

Setting  $\alpha$  as Equation (2.4) guarantees that if there is a feasible buffering solution w.r.t the hold-time constraints, the buffering solution is always feasible w.r.t. the setup constraints, under the assumption that only a buffer is allowed to be inserted at a pin. Although we in fact allow more than one buffer inserted at a pin in our implementation, the empirical ratio still helps to reduce the possibility that the corresponding setup delays of a buffering solution are underestimated such that a feasible solution to the linear programming actually worsens *TNS* after inserting buffers.

Our objective is set to minimize the total number of inserted delays, since the number of inserted delays closely corresponds to the amount of area and power consumption. Therefore, the linear program is formed by combining Equation (2.1)-(2.3) and Equation (2.5)-(2.7) and the following objective:

$$\text{Minimize} \quad \sum_{i \in P} x_i \quad (2.8)$$

However, the setup constraints limit the delays that can be inserted; it is likely that the allowed inserted delays cannot satisfy the hold-time constraints, which makes the above linear program infeasible. To avoid this, a relax variable  $r_i$  is created for each  $i \in O$ . Thus, Equation (2.3) has to be rewritten as follows:

$$ha_i + r_i \geq hold\_req_i \quad \forall i \in O \quad (2.9)$$

$r_i$  is only necessary when there is no feasible solution, so in the objective function, we assign a relatively large cost to the relax variables. Equation



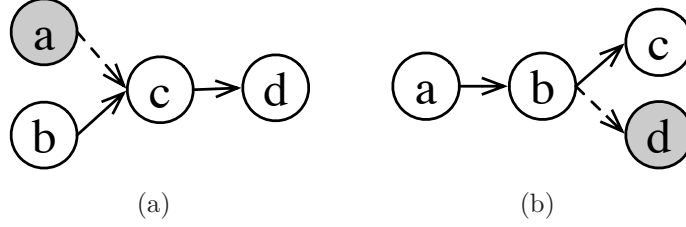


Figure 2.2: Example of missing setup constraints. Pin  $a$  in (a) and pin  $d$  in (b) both have positive setup slacks and positive hold slacks, so they are excluded in the graph that is formed by solid lines, but their corresponding setup timing constraints still have to be included into the linear program.

(2.8) is then changed as follows:

$$\text{Minimize} \quad \sum_{i \in V} x_i + \sum_{i \in O} (|V| + 1) \times r_i \quad (2.10)$$

The above objective function guarantees that  $r_i > 0$  if and only if some corresponding  $x_i$  is maximum w.r.t the setup constraints. This can be easily proved by contradiction.

Recall that  $G$  only contains pins with positive setup slacks and negative hold slacks, which means that there are some pins in the combinational circuit with positive setup slacks and positive hold slacks that are not included. The corresponding setup constraints with those pins have to be formulated into the linear programming model, otherwise the solution to our linear program could violate the setup constraints w.r.t the whole circuit, i.e.  $TNS$  becomes worse. Take Figure 2.2 as an example. In Figure 2.2(a), pin  $a$  has a positive setup slack and a positive hold slack, so pin  $a$  and edge  $(a, c)$  are not included in the graph  $G$ . However, while delays are inserted at pin  $c$  and  $d$ , the actual arrival time from the path  $(a \rightarrow c \rightarrow d)$  is missing in our model. The arrival time from  $a$  has to be carried into the linear program. Let  $SI$  be a set of pins  $\in V$  that contains at least one fan-in with positive setup slacks and positive hold slacks. Pin  $c$  is in  $SI$  in Figure 2.2(a). Then, the following setup constraints are added:

$$sa_i \geq setup\_aat_i + \alpha_i \cdot x_i \quad \forall i \in SI \quad (2.11)$$

where  $setup\_aat_i$  is constant and given from the timer, which carries the arrival time from the fan-ins of pin  $i$ . In Figure 2.2(b), pin  $d$  has a positive

setup slack and a positive hold slack, so it is not included in  $G$ . While delays are inserted at pin  $a$  and  $b$ , the setup required time constraint at pin  $d$  has to be modeled. Let  $SO$  be a set of pins  $\in V$  that has fan-outs with positive setup slacks and positive hold slacks. Pin  $b$  in Figure 2.2(b) is then in  $SO$ . Then, Equation (2.7) can be rewritten as follows:

$$sa_i \leq setup\_req_i \quad \forall i \in O \cup SO \quad (2.12)$$

where  $setup\_req_i$  is constant and given from the timer.

To summarize, our linear programming model can minimize the amount of inserted hold delays while the corresponding setup delays are captured, so the setup constraints can be more accurately maintained. In addition, the partial graph  $G$  correctly models the whole combinational circuit by using the existing timing information from the timer, so there is no need to model the whole combinational circuit, thereby reducing the size of the linear program.

### 2.3.2 Complex Timing Constraints

In order to handle the industrial high-performance designs, complex timing constraints have been modeled into the linear programming formulation. We consider multiple clock domains, multiple cycles, and different clock phases within one clock domain in our formulation. Essentially, these three timing constraints are set for the paths from pins in  $I$  to pins in  $O$ . Thus, additional arrival variables have to be created from those pins to capture those timing constraints. Denote  $I^c$  and  $O^c$  as a set of pins in  $I$  and a set of pins  $O$  that are defined with a complex timing constraint  $c$  as defined above. The timing constraints are formulated as follows:

$$\begin{aligned} ha_i^c &= hold\_aat_i^c + x_i & \forall i \in I^c \\ ha_j^c &\leq ha_i^c + hd_{i \rightarrow j} + x_j & \forall (i, j) \in (I^c \rightarrow O^c) \\ ha_i^c + r_i^c &\geq hold\_req_i^c & \forall i \in O^c \\ sa_i^c &= setup\_aat_i^c + \alpha_i \cdot x_i & \forall i \in I^c \\ sa_i^c &\geq setup\_aat_i^c + \alpha_i \cdot x_i & \forall i \in SI^c \\ sa_j^c &\geq sa_i^c + sd_{i \rightarrow j} + \alpha_j \cdot x_j & \forall (i, j) \in (I^c \rightarrow O^c) \\ sa_i^c &\leq setup\_req_i^c & \forall i \in O^c \cup SO^c \end{aligned}$$

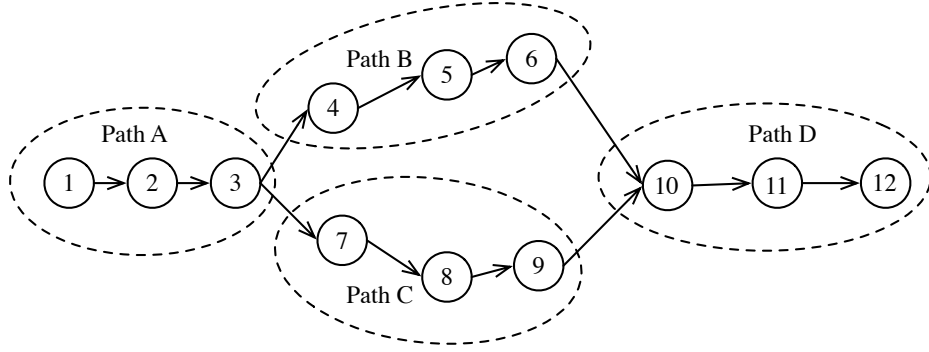


Figure 2.3: The graph is reduced to four paths and the edges between these paths.

Note that only the arrival variables are duplicated for the complex timing constraints. The delay variables are identical for different timing constraints.

### 2.3.3 Graph Reduction

In previous sections, we always assume a delay variable and two arrival variables for every pin. In this section, we demonstrate that some variables are not necessary, so the graph can be reduced. We perform a breadth-first-search on  $G$  to find a set of paths such that for each path, its start pin is in  $I \cup SI$  or has multiple fan-ins and its end pin is in  $O \cup SO$  or has multiple fan-outs. Then,  $G$  is reduced to these paths and the edges connecting different paths. In Figure 2.3, the graph is reduced to Path A, Path B, Path C, Path D, edge  $(3 \rightarrow 4)$ , edge  $(3 \rightarrow 7)$ , edge  $(6 \rightarrow 10)$ , and edge  $(9 \rightarrow 10)$ . Obviously, any pin in the middle of the paths is redundant and its arrival variables can be removed. The arrival variables for pin 2, 5, 8, 11 are redundant in Figure 2.3. Furthermore, we let one path have only one delay variable. Given such path  $P$ , the delay variable is assigned to pin  $x$  s.t.

$$\alpha_x = \max_{i \in P} \{\alpha_i\}.$$

By doing so, the size of the linear program can be reduced. For Figure 2.3, the linear program becomes  $8 \times 2$  arrival variables and 4 delay variables, compared to the original program with  $12 \times 2$  arrival variables and 12 delay variables. Especially since the hold-violation removal is always per-

formed after the setup optimization, the design containing a certain number of buffers/inverters inserted for the setup optimization can be reduced.

As for the optimality of the reduced linear program, it is easy to see that the arrival times can be propagated accurately, since the branching points always have arrival variables. For the inserted delays, consider a path containing two variables  $x_1$  and  $x_2$ , and the original timing constraint as follows:

$$\begin{aligned} x_1 + x_2 &\geq \textit{hold\_req} \\ 5x_1 + 10x_2 &\leq \textit{setup\_req} \end{aligned}$$

where  $\alpha_1 = 5$  and  $\alpha_2 = 10$ . After reduction, the path has only one delay variable  $X$  with a  $\alpha$  of 10, so the hold-time constraint becomes

$$X = x_1 + x_2 \geq \textit{hold\_req}$$

and the setup constraint becomes

$$5x_1 + 10x_2 \leq 10X = 10(x_1 + x_2) \leq \textit{setup\_req}$$

So the optimal solution to the reduced linear program is still optimal to the original linear program. Therefore, by graph reduction, the size of our linear program is reduced while the optimality is still maintained.

## 2.4 Bottom-Up Buffer Insertion

We will realize the solution of the linear programming model by inserting buffers in this section. First, a dynamic programming algorithm is proposed to insert buffers to realize the required delay at a pin. Next, a bottom-up methodology is presented to process all the pins.

### 2.4.1 Dynamic Programming Based Algorithm

We first consider the following problem: Given a pin  $p$ , hold delay  $D_H$  and setup delay  $D_S$ , find a buffering solution at pin  $p$  from a buffer library  $B$  such that (1) the hold delays introduced by the chosen buffers are as close

to  $D_H$  as possible, (2) the setup delays introduced by the chosen buffers are not larger than  $D_S$ , (3) the area of the chosen buffers is minimized.

To solve this problem, a dynamic programming (DP) based algorithm is proposed. A set of buffering candidates  $C(L, d_h, d_s, A)$  is kept during the process, where  $L$  represents a list of the chosen buffers,  $d_h$  is the hold delay introduced by  $L$ ,  $d_s$  is the setup delay introduced by  $L$ , and  $A$  is the area of  $L$ . For each buffer in  $B$ , we insert it to any of the existing candidates and then make up new buffering candidates. Suppose we have a new candidate  $C'(I', d'_h, d'_s, A')$  by inserting a buffer  $b$  into an existing candidate  $C(I, d_h, d_s, A)$ . First, if  $d'_s > D_S$ ,  $C'$  is removed immediately. Second, if  $d'_h \leq d_h$ ,  $C'$  is removed as well. The reason is that we see buffers as delay elements, so we want hold delays to be increased whenever inserting a buffer. In some cases, it is possible that the hold delays can become larger than  $d_h$  after inserting more buffers; however, more buffers mean larger area and more power consumption, so we simply remove this kind of candidate. Similarly, we do not want to overshoot  $D_H$  to cause over-buffering, so if  $d'_h > D_H + \text{margin}$  where *margin* is a parameter, then  $C'$  is removed too. In our experiment, *margin* is set to 20ps. Next,  $C'$  is dominated by any existing candidate  $C^*(I^*, d_h^*, d_s^*, A^*)$  if  $d'_h < d_h^*$  and  $A' > A^*$ . Such candidates that are dominated by other candidates are pruned. The process stops until there are no new candidates. Since  $D_H$  and  $D_S$  bound the possible candidates and decreasing  $d_h$  is not allowed, the process usually stops very quickly in our experiment. Once the process stops, we choose the candidate that has the largest ratio of  $d_h/A$  as the buffering solution.

## 2.4.2 Bottom-up Methodology

A bottom-up methodology based on the above DP algorithm is presented to realize all the delay values obtained from the linear programming. We process the pins by the bottom-up topological ordering (i.e. from  $PO$  to  $PI$ ). For each pin, we have a hold delay and a setup delay obtained from the linear programming, so we can apply the above DP algorithm to insert buffers. However, we can see that the DP algorithm cannot realize the exact amount of hold delays/setup delays by inserting buffers. All those delays that failed to be realized are extra delays that can help buffer insertion at

other pins. Thus, such delays have to be collected during the bottom-up process.

Suppose now we are processing pin  $p$ . Then buffer insertion was already done for those pins in the downstream of pin  $p$ , so we can have an updated required setup time denoted as  $cur\_setup\_req_p$  which is equal to

$$setup\_req_p - ds\_delay$$

where  $setup\_req_p$  is the original required time from the timer and  $ds\_delay$  can be easily calculated by propagating the inserted delays as we process the pins by the topological ordering. Then,  $cur\_setup\_req_p - sa_p$  is the extra setup delay at pin  $p$ . Note that  $sa_p$  is known after solving the linear programming model, so no additional calculation is necessary. So, the setup delay  $D_H$  is set to

$$x_p + cur\_setup\_req_p - sa_p$$

We can then collect the extra hold delay similarly for pin  $p$  to get the hold delay  $D_S$ . Finally, the DP algorithm can be applied on pin  $p$  with  $D_H$  and  $D_S$ . Therefore, by the bottom-up topological ordering, we can perform buffer insertion on each pin while collecting extra delays during the process.

## 2.5 Our Optimization Flow

Our optimization flow for a circuit-level design is illustrated in Figure 2.4. After launching the timer, we start from the pin with the worst hold slack to get a combinational circuit that only contains pins with the negative hold slacks and positive setup slacks. Then, the linear programming model is used to solve the combinational circuit. The bottom-up buffer insertion is then applied to realize the delay values obtained from the linear programming into buffers. After this combinational circuit is finished, the timer is launched again to get the accurate timing information for the next pass of the optimization. The flow stops until all the pins of the design are processed.

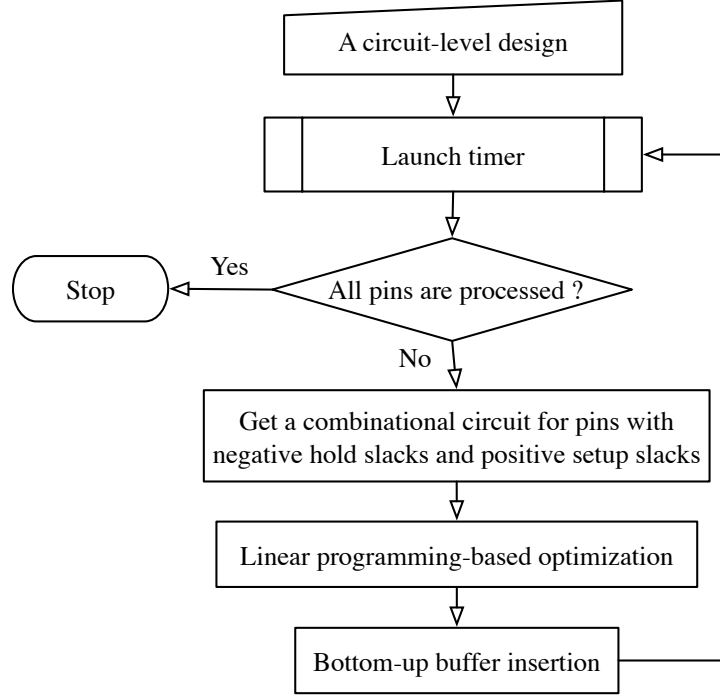


Figure 2.4: The flow of our approach.

## 2.6 Experimental Results

Our approach is written in C++ and the experiments are performed on a machine with an eight-core 2.8GHz Intel Xeon processor and 24GB of memory.

The test cases are four industrial designs, which are all at the pre-detailed routing stage. The design statistics and the initial timing statistics of these designs are shown in Table 2.1. Then, we make three sets of experiments: (1) run our proposed approach; (2) run the state-of-the-art industrial hold optimization flow; (3) run our proposed approach first and then perform the state-of-the-art industrial hold optimization flow (called OURS+IND for short). The industrial hold optimization flow applies a set of transformations such as buffering, sizing, composition, decomposition, local restructuring, and area recovery. The results of the three sets of experiments are listed in Table 2.2, Table 2.3, and Table 2.4, respectively. And the results reported in Table 2.2-2.4 are all obtained after legalization and detailed routing.

In Table 2.2, we observe that our proposed approach obtains 74% and 40% average reduction in the total negative hold slacks (THS) and the worst negative hold slack (WHS), respectively (in the best case, 88% and 61%

Table 2.1: Summary and the initial timing statistics of the industrial designs

	#cells	TNS (ps)	WNS (ps)	THS (ps)	WHS (ps)	area (10 <sup>6</sup> )	leak- age
d1	372K	32K	194	996K	186	4.09	544
d2	456K	1.47·10 <sup>6</sup>	384	654K	196	6.52	1035
d3	269K	270K	850	437K	213	2.23	2.63
d4	1170K	4.17·10 <sup>6</sup>	3340	3.6·10 <sup>6</sup>	412	9.07	34.85
avg		1.49·10 <sup>6</sup>	1192	1.4·10 <sup>6</sup>	252	5.48	404

Table 2.2: Results of our proposed approach

	TNS (ps)	WNS (ps)	THS (ps)	WHS (ps)	area (10 <sup>6</sup> )	leak- age	CPU (m)
d1	32K	194	116K	85	4.25	556	5.6
d2	1.47·10 <sup>6</sup>	384	137K	77	6.63	1047	4.3
d3	269K	850	123K	145	2.28	2.75	5.6
d4	4.19·10 <sup>6</sup>	3340	1.1·10 <sup>6</sup>	302	9.32	34.9	20
avg	1.49·10 <sup>6</sup>	1192	3.7·10 <sup>5</sup>	152	5.62	410	8.88
c <sup>a</sup>	1	1	74%	40%	1.03	1.01	

---

<sup>a</sup>Comparison with Table 2.1. TNS, WNS, area, and leakage: ratio of the average. THS and WHS: average reduction.

reduction in THS and WHS, respectively), while the total negative setup slacks (TNS) and the worst setup slack (WNS) both remain almost the same on average, and there is a very slight increment of area and leakage power (on average, 3% in area and 1% in leakage power) compared with the initial results in Table 2.1.

Next, let us see the results of running our proposed approach together with the industrial hold optimization flow. Two sets of comparisons are listed in Table 2.4: (1) comparison with the initial results (see row c1); (2) comparison with the results of running the industrial hold optimization flow only (see row c2). Compared with the initial results, OURS+IND achieves 99.5% and 70% average reduction in THS and WHS, respectively, and there are only 4% and 2% average increment of area and leakage power, respectively, while WNS remains the same. We also observe that OURS+IND obtains better TNS for all the designs, and so does the hold optimization flow (see TNS in Table 2.3). The reason is that the transformations in the industrial hold optimization flow also help to reduce setup violations, but our approach only



Table 2.3: Results of the industrial hold optimization flow

	TNS (ps)	WNS (ps)	THS (ps)	WHS (ps)	area (10 <sup>6</sup> )	leakage	CPU (m)
d1	31K	194	792	54	4.34	571	9.67
d2	1.46·10 <sup>6</sup>	382	777	72	6.71	1045	9.67
d3	257K	850	15K	122	2.32	3.25	21.1
d4	4.03·10 <sup>6</sup>	3340	80K	281	9.42	35.26	94
avg	1.44·10 <sup>6</sup>	1192	24142	132	5.7	413.6	33.61
c <sup>a</sup>	0.97	1	98%	47%	1.04	1.02	

---

<sup>a</sup>Comparison with Table 2.1.

Table 2.4: Results of our proposed approach together with the industrial hold optimization flow

	TNS (ps)	WNS (ps)	THS (ps)	WHS (ps)	area (10 <sup>6</sup> )	leakage	CPU (m)
d1	31.8K	194	258	17	4.33	566	9.2
d2	1.46·10 <sup>6</sup>	384	157	48	6.72	1052	9.2
d3	260K	850	2523	64	2.31	3.02	14.5
d4	4.18·10 <sup>6</sup>	3340	26K	171	9.44	35.13	78
avg	1.48·10 <sup>6</sup>	1192	7235	75	5.7	414	27.73
c1 <sup>a</sup>	1	1	99.5%	70%	1.04	1.02	
c2 <sup>b</sup>	1.03	1	70%	43%	1	1	1.21X

---

<sup>a</sup>Comparison with Table 2.1.

<sup>b</sup>Comparison with Table 2.3.

focuses on hold violations. Compared with the results of running the hold optimization flow only, OURS+IND gives 70% and 43% reduction in THS and WHS on average, respectively, and WNS, area, and leakage power all still remain the same on average. There is 3% average increment of TNS, since the industrial hold optimization flow performs better on setup violations removal. Furthermore, OURS+IND runs faster than the industrial hold optimization flow on all the designs (1.21X speedup on average and 1.46X speedup in the best case).

## 2.7 Conclusion

In this chapter, we first propose a linear programming based approach that minimizes the number of inserted delays. A bottom-up buffer insertion is then presented to realize the delay into buffers. The flow of optimizing a circuit-level design is also presented. By running our approach together with the state-of-the-art industrial hold optimization flow on the industrial designs, better results in terms of hold violations and runtime are reported.

# CHAPTER 3

## AN AREA-AWARE BUFFER INSERTION FOR HOLD VIOLATIONS

### 3.1 Introduction

Buffer insertion is an important transform in the timing optimization process. Typically, for the setup optimization, buffers are used to speed up the circuit, while for the hold optimization, buffers are inserted as delay elements. The objective of the buffer insertion problem is to insert a number of buffers to meet timing constraints. In this chapter, we focus on the problem of buffer insertion for hold violations.

The buffer insertion problem for setup constraints has been studied extensively. For a given routing tree, Van Ginneken [2] in 1990 proposed a dynamic programming approach that can optimally find where to insert buffers in the routing tree to meet the timing constraints. Then, the dynamic programming approach has been extended based on different problem formulations by a number of research [3, 4, 5, 6, 7, 8]. More than one size of buffer is considered in [3], and [4] further speeds up the Van Ginneken algorithm. Some researches consider routing and buffer insertion simultaneously to minimize the interconnect delay ([6, 7, 8, 25]). Simultaneous gate sizing and buffer insertion are proposed in [5, 26]. Buffer insertion under accurate delay models is presented in [25, 27], and buffer insertion with minimum cost is proposed in [28, 29].

Compared to researches on buffer insertion for setup violations, there are only a few works studying the problem of buffer insertion for hold violations. As discussed in Chapter 2, the problem of buffer insertion for hold violations is typically seen as a problem of minimizing the amount of inserted delay ([20, 21, 22, 23, 24]) under an assumption that the inserted delays can be realized by unit-delay elements. However, these existing works do not consider discrete buffer sizes, accurate timing models, and complex tim-

ing constraints, etc., making them impractical on modern designs. Buffer insertion considering discrete buffer sizes for hold violation becomes a necessity. Moreover, while buffers can help fix hold-time violations, they also increase the difficulty of routability and the utilization of a design. And the larger-area cells contribute more leakage power, which is a crucial factor in modern designs. Therefore, the objective of our buffer insertion problem is to find which buffers to be inserted in order to meet the timing constraints, meanwhile minimizing the total area of inserted buffers.

We first present a set of pruning rules, and then an optimal algorithm is proposed based on the pruning rules. However, due to the large complexity of the optimal algorithm, limiting the input size is necessary in order to achieve acceptable runtime. Then, based on an observation made on the relationship between buffer size and output transition, a heuristic algorithm is presented. Furthermore, in order to have better scalability with large scale designs, a machine learning technique is adopted, and an algorithm based on gradient descent and logistic regression is proposed. In the experiment, we incorporate each of our approaches into the industrial optimization flow, and then test it on the industrial designs. Different approaches show different results in terms of quality and runtime. Overall, the machine learning based approach shows better results in terms of quality and runtime.

The remainder of this chapter is organized as follows. Section 3.2 gives the background of the buffer insertion problem; our algorithms are proposed in Sections 3.3, 3.4, and 3.5; finally, the experiment results are given in Section 3.6 and a conclusion in Section 3.7.

## 3.2 Background

In this section, we first present our problem formulation, and then show how to adopt it in the hold optimization flow.

### 3.2.1 Problem Formulation

The input to the problem is a multi-pin net which consists of a driver pin  $p_D$  and a set of sink pins denoted as  $P_S$ . Each pin  $p$  is associated with a set of hold-time and a set of setup constraints.  $hold\_slack(p)$  and  $setup\_slack(p)$

represent the hold slack and the setup slack on pin  $p$ , respectively. The input also contains a target pin  $p_T$ , in which the buffers are only allowed to be inserted next to the location of pin  $p_T$ , and we want to fix hold-time violations at pin  $p_T$ . If there exists a buffering solution that makes positive hold slack at  $p_T$ , the buffer solution with smaller area is preferred. If no buffering solution can make positive hold slack at  $p_T$ , the goal becomes to maximize the hold slack at  $p_T$ . When two buffering solutions have the same hold slack, the area of the solution is used as a tiebreaker. Meanwhile, the time constraints at any other pin always have to be met, i.e. not worsening the slacks after buffer insertion. Therefore, given a set of buffer cells  $B$ , the objective of the problem is to find a chain of buffers  $S \in B$  that are inserted next to pin  $p_T$  such that the negative hold slack at  $p_T$  is maximized to zero, while the total area of the buffer chain is minimized and the timing constraints are always met. The problem then can be defined as follows:

$$\text{Maximize } \min(0, \text{hold\_slack}(S, p_T)) \quad (3.1)$$

$$\text{and Minimize } \sum_{b \in S} \text{area}(b) \quad (3.2)$$

$$\text{s.t. } \text{setup\_slack}(S, p) \geq \min(0, \text{setup\_slack}(p)) \quad \forall p \in P_S \quad (3.3)$$

$$\text{hold\_slack}(S, p) \geq \min(0, \text{hold\_slack}(p)) \quad \forall p \in P_S \quad (3.4)$$

where  $\text{hold\_slack}(S, p)$  represents the new slack at pin  $p$  after inserting a chain of buffers  $S$  at the target pin. It is known that the min-cost buffer insertion problem that only considers setup constraints is proved to be NP-hard ([28]). Since the hold-time constraints are similar to the setup constraints, the only difference is the way of calculating slack; our min-cost problem for hold violations is naturally NP-hard.

Note that the cell delay model used in this chapter is based on the lookup table where the two inputs are slew and load capacitance, respectively, where the slew is also based on a slew lookup table. As it is very costly to launch the timer to get the accurate slack after a buffer is inserted, we calculate the first-level timing information (i.e. delay and output slew of the driver-cell and the sink-cells) ourselves.

### 3.2.2 The Optimization Flow

This chapter only focuses on a target pin in a multi-pin net. In order to work on a circuit-level design, we need to have an optimization flow that can determine which target pin to optimize and provide the necessary timing information, etc. Then, our buffer insertion can find a solution at the given target pin. Iteratively, a new target pin would be specified to insert buffers until there is no hold violation or some convergence is reached. The linear programming based optimization presented Chapter 2.3 can be used to serve in this role.

## 3.3 An Optimal Algorithm

In this section, we introduce a set of pruning rules, and then propose an optimal algorithm based on these rules. Consider a chain of buffers  $S$  at a target pin  $p_T$ . Since buffers are only seen as delay elements in this chapter, if  $S$  already has an positive hold slack, it is unnecessary to add any buffer into the chain, as the area of the new chain would be always larger than that of  $S$ . Thus, we have the following pruning rule (note that the symbol  $>$  means dominating):

**Rule 1**  $S > S' = S + b \quad \forall b \in B$ , if  $hold\_slack(S, p_T) \geq 0$ .

Suppose there exists a chain of buffers  $S^*$  where  $hold\_slack(S^*, p_T) \geq 0$ . For any chain of buffers  $S$ , if  $area(S)$  is larger than  $area(S^*)$ , then we can easily prune out  $S$ , as smaller area is always preferred. Note that if their area is the same, then the hold slack is used as a tie-breaker, and the chain with the larger hold slack dominates the other.

**Rule 2**  $S^* > S$ ,  $\exists S^* : hold\_slack(S^*, p_T) \geq 0$  and  $S : area(S) > area(S^*)$ .

Next, consider two chains of buffers  $S_i = (b, h\_slew_i, s\_slew_i)$  and  $S_j = (b, h\_slew_j, s\_slew_j)$ , such that both chains end in the same buffer  $b$ .  $h\_slew(s\_slew)$  represents the hold(setup) output slew of the last buffer in the chain. Then, a pruning rule is defined as follows:

**Rule 3**  $S_i \geq S_j$  if  $h\_slew_i \geq h\_slew_j$ ,  
 $s\_slew_i \leq s\_slew_j$ ,

$$\begin{aligned}
& hold\_slack(S_i, p_T) \geq hold\_slack(S_j, p_T) \\
& setup\_slack(S_i, p_T) \geq setup\_slack(S_j, p_T), \text{ and} \\
& area(S_i) < area(S_j).
\end{aligned}$$

Since  $S_i$  and  $S_j$  both end in the same buffer, once the conditions listed in Rule 3 are met, considering the addition of a buffer into either chain,  $S_i$  is always better than  $S_j$  in terms of timing and area. The correctness of this rule can be proved by contradiction.

*Proof.* Assume  $S_i > S_j$  and a new chain  $S = S_j + S_k$  is the best solution to our problem, i.e. it has the largest hold slack if the slack is negative or the smallest area if non-negative. This implies that  $S > S' = S_i + S_k$ . W.l.o.g., we only consider the hold slack here. Since the last buffers  $b$  in  $S_i$  and  $S_j$  connect to the same chain  $S_k$ , they have the same output load capacitance, while  $h\_slew_i > h\_slew_j$ , the delay introduced by  $S_k$  in  $S_i$ , is larger than that in  $S_j$ , which means  $hold\_slack(S') > hold\_slack(S)$ , making the assumption  $S > S'$  a contradiction. And similarly, this proof can be extended to the setup slack and the area. By combining the rules of hold slack, setup slack, and area, Rule 3 can be proved correct.  $\square$

Based on these pruning rules, an optimal algorithm is proposed in Algorithm 3.1. Suppose the size of the longest chain is  $l$ , and then the time complexity of Algorithm 3.1 is  $O(|B|^l)$ . Given this large complexity, in order to solve the very large scale designs, we can limit the number of buffers (line 6 in Algorithm 3.1) and set an upper limit for the chain size to reduce the runtime. Let  $m \leq |B|$  be the number of buffers and  $n$  be the upper limit of the chain size, where  $m$  and  $n$  are both constant number. The runtime becomes  $O(m^n)$ .

### 3.4 A Heuristic Algorithm

When a buffer  $b$  is inserted into a chain  $S$ , typically we expect the hold delay introduced by the new chain  $S'$  is as large as possible. The delay can be contributed by either a large output slew from  $b$  or a large intrinsic delay of  $b$ . An observation we made on the buffer library is that, generally, a larger buffer tends to have a larger intrinsic delay but give a smaller output slew, while a smaller buffer tends to have a smaller intrinsic delay but generate a

---

**Algorithm 3.1** An Optimal Algorithm

---

```

1:  $S^* \leftarrow null$  ▷ Store the best solution
2:  $G \leftarrow S_{initial}$  ▷ An empty chain with the initial timing information
3:  $Map \leftarrow \emptyset$  ▷  $Map(b)$ : buffer chains end in buffer  $b$ 
4: while  $G$  is not empty do
5:    $G_{new} \leftarrow \emptyset$ ;
6:   for any  $b$  in  $B$  do
7:      $G_b \leftarrow \emptyset$ ;
8:     for any  $S$  in  $G$  do
9:       if  $hold\_slack(S, p_T) \geq 0$  then ▷ Rule 1
10:        continue;
11:       end if
12:        $S' = S + b$ 
13:       Calculate the timing information for  $(S', p_T)$ 
14:       if  $setup\_slack(S', p) < \min(0, setup\_slack(p))$  or
15:        $hold\_slack(S', p) < \min(0, hold\_slack(p)) \forall p \in P_S$  then
16:        continue
17:       end if
18:       if  $hold\_slack(S^*) \geq 0$  and  $area(S') > area(S^*)$  then ▷ Rule 2
19:        continue
20:       end if
21:        $G_b \leftarrow G_b + S'$ ;
22:        $S^* \leftarrow S'$  if necessary
23:     end for
24:     for any  $S$  in  $G_b$  do
25:       if  $S$  is dominated by  $S'$  for  $S' \in Map(b)$  then ▷ Rule 3
26:         $G_b \leftarrow G_b - S$ 
27:        break
28:       end if
29:     end for
30:      $G_{new} \leftarrow G_{new} + G_b$ ;
31:   end for
32:    $G \leftarrow G_{new}$ 
33: end while

```

---



larger output slew. Since we would like to minimize the area of the inserted buffers while the hold delay is maximized to zero, we can have a heuristic that considers a chain that ends in a buffer  $b'$ , and we are adding a buffer into the chain: (1) only buffers that have smaller size than  $b'$  can be inserted, as the smaller buffer should provide a larger output slew than  $b'$ ; (2) if  $b'$  is the smallest buffer, all the buffers in the library  $B$  are allowed to be inserted, as they should have larger intrinsic delay to compensate for the smaller output slew than  $b'$ . Therefore, a heuristic algorithm is proposed by modifying Line 12 in Algorithm 3.1, which is as shown in Algorithm 3.2.

---

**Algorithm 3.2** A Heuristic Algorithm

---

▷ Only the changes on Line 12 in Algorithm 3.1 are shown here, as the rest of the algorithm remains the same.

```

11: ...
12:  $b' \leftarrow$  the last buffer in  $S$ 
13: if  $b'$  is the smallest buffer or  $area(b) \leq area(b')$  then
14:    $S' = S + b$                                 ▷ insert  $b$ 
15: else
16:   continue                                    ▷ not insert  $b$ 
17: end if
18: ...

```

---

### 3.5 A Machine Learning Based Approach

Although the heuristic algorithm proposed in the previous section can eliminate a large solution search space, the heuristic algorithm still has very large time complexity. In order to work on very large scale designs, and not have the limitation on the buffer library size and the buffer chain size, in this section, an algorithm based on machine learning is proposed.

The motivation to adopt machine learning for the buffer insertion problem is that in the optimization flow, a set of target pins are optimized iteratively, and some of them might have similar timing constraints and share similar net configuration (i.e. similar net structures, driver cells, sink cells, etc.), which makes some of them similar buffer insertion problems. Therefore, by viewing those buffer insertion problems together, a solution to one problem might be already found in the other problem. The knowledge gained from

solving previous problems can be used to help solve the current problem, thereby reducing the runtime of solving all the problems.

We model the buffer insertion problem as a learning problem: Learn a hypothesis  $h$  such that for a given chain of buffers  $S$ ,  $h$  estimates the probability that  $S$  is not dominated by other buffer chains. Therefore, in the buffer insertion problem,  $h$  can be used as another pruning rule to determine if a buffer chain is allowed to be inserted in any buffer. The details of the learning process and the application of the learned hypothesis to the buffer insertion problem are presented in the following sections.

### 3.5.1 Data Preparation

The data for the machine learning problem is a set of buffer chains. In the learning process, each chain has to associate with a label, so the correctness of a hypothesis can be tested. We label the chains that are pruned out in Algorithm 3.1 as NEGATIVE, and those chains that are not dominated as POSITIVE. For the pruning rules, Rule 1 and Rule 2, they can only prune out the chain that is being processed at the moment. So, there is nothing that can be learned. While a chain is not dominated in the current iteration, Rule 3 can make it dominated by the chains generated by the later iterations. Thus, if we can learn that such a chain will be dominated in the later iterations and stop to insert any buffers into the chain in the present iteration, then fewer chains would be generated in each iteration that help reduce runtime and still keep the optimality of the algorithm. Due to different output slews impacted from the output slew of the driver cells which are likely different when optimizing different target pins, we only focus on chains with at least two buffers, such that the output slew would be bound by the output slews from the buffers, as the previous cells is always a buffer.

A set of examples  $T$  to the learning process then can be setup as follows: In the optimal algorithm (Algorithm 3.1),

- If a chain  $S$  is pruned out by Rule 3 (Line 25), add  $S$  to  $T$ , and label it NEGATIVE.
- Add the chains that are not pruned out in the iteration ( $G_b$  in Algorithm 3.1 to  $T$ , and label them POSITIVE.

Note that  $S$  must contain more than one buffer. Due to the nature of Algorithm 3.1, the chains generated in earlier iterations do not have a chance to test if it is dominated by the chains generated in later iterations. Rule 3 is examined again on  $T$  for correct labeling.

A set of features  $X$  is needed to represent an example. Since the learning process is based on Rule 3, naturally the items used in Rule 3 such as output slew, slack, and area are used as our feature set. However, as the setup slack is not always positive (i.e. the hold slack is still maximized under the condition of not worsening the setup slack when the setup slack is negative), setup delay generated by the chain is used in the feature set, instead of the setup slack. Thus, for an example  $S$ , its feature set  $X$  is defined as

$$\begin{aligned} X \equiv \vec{x} &= (x_0, x_1, x_2, x_3, x_4, x_5), \text{ where} \\ x_0 &= 1 \\ x_1 &= -h\_slew_S \\ x_2 &= s\_slew_S \\ x_3 &= -hold\_slack(S, p_T) \\ x_4 &= setup\_delay(S, p_T) \\ x_5 &= area(S) \end{aligned}$$

Note that the hypothesis that will be introduced in the next section has a bias parameter, so  $x_0$  is always 1 for this purpose. Then by combining the label, a training example  $i$  in  $T$  can be represent as  $(\vec{x}^{(i)}, label^{(i)})$ .

Since those features have different units and scales, a feature normalization is applied to standardize the range of the features. For a feature set  $\vec{x}$ ,

$$x_i = \frac{(x_i - \mu)}{\sigma} \quad \forall x_i \in \vec{x}$$

and then

$$\vec{x} = \frac{\vec{x}}{\|\vec{x}\|}$$

where  $\mu$  is the mean for feature  $x_i$  over all the examples, and  $\sigma$  is the standard deviation.

Since the examples are generated from optimizing different targets in the hold optimization flow, which are not stored in the memory, an online algorithm ([30, 31]) is used to calculate mean and standard deviation.

### 3.5.2 Learning: Gradient Descent and Logistic Regression

In this section, the learning process is presented. We choose the logistic regression as the hypothesis, as it estimates a probability of one event. In our formulation, given an example, it can predict the label based on the probability, so we can decide if the corresponding chain has to be pruned out or not. The framework of the logistic regression is as follows:

$$h_{\theta}(\vec{x}) = \frac{1}{1 + e^{-\vec{\theta} \cdot \vec{x}}}$$

$$\text{Predict} \begin{cases} y = 1, & \text{if } h_{\theta}(\vec{x}) \geq 0.5 \\ y = 0, & \text{if } h_{\theta}(\vec{x}) < 0.5 \end{cases} \quad (3.5)$$

where  $\theta$  is a set of parameters that the learning process has to learn. Note that  $y = 1$  means POSITIVE label, and  $y = 0$  means NEGATIVE label.

Then, for an example with a feature set  $x$  and a label  $y$ , a cost function is set as follows:

$$\text{cost}(h_{\theta}(\vec{x}), y) = \begin{cases} \log(h_{\theta}(\vec{x})), & \text{if } y = 1 \\ -\log(1 - h_{\theta}(\vec{x})), & \text{if } y = 0 \end{cases}$$

So, the total cost over the example set  $T$  is denoted as  $J(\theta)$ , and

$$\begin{aligned} J(\theta) &= \frac{1}{|T|} \sum_{i \in T} \text{cost}(h_{\theta}(\vec{x}^{(i)}), y^{(i)}) \\ &= -\frac{1}{|T|} \sum_{i \in T} \left( y^{(i)} \log(h_{\theta}(\vec{x}^{(i)})) + (1 - y^{(i)}) \log(1 - h_{\theta}(\vec{x}^{(i)})) \right) \end{aligned}$$

Then, in order to fit the parameters  $\theta$ ,  $J(\theta)$  is minimized by the gradient decent of  $J$  based on parameters. A gradient for a parameter  $\theta_j$  is

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{|T|} \sum_{i \in T} \left( h_{\theta}(\vec{x}^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

However, to avoid overfitting (i.e. the learned hypothesis may fit the training examples too well, but fail to predict unseen new examples), a regularization item is added into the gradient, so it makes

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{|T|} \sum_{i \in T} \left( h_{\theta}(\vec{x}^{(i)}) - y^{(i)} \right) x_j^{(i)} + \frac{\lambda}{|T|} \theta_j \quad (3.6)$$

where  $\lambda$  is called regularization parameter, and  $\lambda$  is a constant.

So,  $\theta$  is updated by its corresponding gradient until  $\theta$  reaches some convergence. The learning algorithm based on gradient ascent is shown in Procedure LEARNING. Note that in order to progress faster, we apply the mini-batch gradient descent.

---

```

1: procedure LEARNING(A testing example set  $E$ )
2:    $\vec{\theta} \leftarrow \vec{0}$ 
3:   for  $i = 1$  to  $\pi$  do                                      $\triangleright \pi$ : # of iterations. A constant
4:      $\theta_{prev} \leftarrow \theta$ 
5:     repeat
6:        $B \leftarrow$  the next  $\beta$  examples in  $E$                 $\triangleright \beta$ : batch size. A constant
7:       calculate  $\frac{\partial}{\partial \theta_j} J(\theta)$  over  $B$                   $\triangleright$  Equation (3.6)
8:        $\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$  for every  $\theta_j \in \vec{\theta}$     $\triangleright \alpha$ : learning rate. A
       constant
9:     until every example in  $E$  is processed
10:     $\Delta\theta \leftarrow \theta - \theta_{prev}$ 
11:    return if  $\Delta\theta < \gamma$                                     $\triangleright \gamma$ : convergence threshold. A constant
12:  end for
13: end procedure

```

---

### 3.5.3 The Algorithm for the Buffer Insertion

The machine learning based algorithm is based on Algorithm 3.1. The algorithm would work in two modes: either the training mode or the applying mode. For the training mode, the buffering solution is found in the same way as in Algorithm 3.1. The only extra work is to collect the testing examples as described in Section 3.5.1. Then the learning procedure is applied on the training examples to train the hypothesis. For the applying mode, the hypothesis is used to a pruning rule. A chain is pruned out if the prediction of Equation (3.5) is 1.

It is unrealistic to expect that one hypothesis can predict all the examples obtained from working on all the targets in the optimization process. So, we divided the setup slack and hold slack into different ranges, then for each target pin, depending on which range its initial hold slack and setup slack fall into, a corresponding hypothesis is picked up for this target. And, in the beginning  $N$  examples, the algorithm would work in the training mode,

and after that, it switches to the applying mode. Similarly, we break the hold slack into different ranges, and  $N$  is assigned based on which ranges the initial hold slack at the target pin belongs to.

## 3.6 Experimental Results

Our approaches are written in C++ and the experiments are performed on a machine with an 32-core 2.81GHz Intel Xeon processor and 500GB of memory.

The test cases are seven industrial designs, and the design statistics and the initial timing are shown in Table 3.1. Our experiment is set up by incorporating our buffer insertion approaches into the state-of-the-art industrial hold optimization flow. We test four approaches: the optimal algorithm (OPT), the optimal algorithm limited by  $m = 40$  and  $n = 3$  (Limited), the heuristic algorithm (HEU), and the machine learning based approach (ML). The results are shown in Table 3.2, and due to the confidentiality, they are normalized by the results from the industrial hold optimization (i.e. running the industrial optimization flow without combining our approaches). For WNS, TNS, WHS, THS, and area, a result  $v$  is normalized as  $(v - v_{ind})/v_{ind}$ , and for Runtime, a result  $v$  is normalized by  $v_{ind}/v$ , where  $v_{ind}$  is obtained from the hold optimization flow without our approaches. Note that WNS, TNS, WHS, and THS are negative slacks, so in the table, a positive result represents an increase in the negative slacks that means a degradation in timing compared to the industrial hold optimization; on the other hand, positive results mean timing improvement.

For design d1, 0ps means there is no negative hold slack, i.e. no hold violation. All four approaches show pretty much the same results on design d1. On design d2, OPT, HEU, and ML all show very good reduction in WHS and THS; however, they all degrade TNS compared to the industrial hold optimization. Limited performs better in terms of timing but uses a slight more area. Therefore, it is hard to decide which approach performs better.

On design d3, although OPT outperforms other approaches in terms of quality, it is 0.71X the speed of the industrial hold optimization. Therefore, among these 4 approaches, ML achieves better results in terms of quality and runtime on design d3. Moreover, not only for design d3, but also for designs

Table 3.1: Summary and the initial timing statistics of the industrial designs

Data	#Cells	WNS (ps)	TNS (ps)	WHS (ps)	THS (ps)	Area (10 <sup>6</sup> )
d1	213K	-425	-44K	-357	-113K	1.76
d2	659K	-1158	-79.4K	-501	-412K	6.32
d3	360K	-1152	-555K	-998	-262K	2.08
d4	244K	-696	-316K	-1192	-583K	1.23
d5	1168K	-4559	-12600K	-347	-2420K	9.07
d6	221K	-1979	-3060K	-1688	-1800K	1.83
d7	2076K	-1292	-18400K	-7613	-030K	79.1

d4, d6, and d7, OPT also shows very long runtime, which is probably due to its high time complexity.

For design d4, excluding OPT, it is difficult to decide which approach performs better, as they all have better results in different metrics. For design d5, HEU outperforms the other approaches in terms of both quality and runtime. And for design d6, it is also obvious that ML outperforms the other approaches. For design d7, Limited has better timing results but uses more area, while on the other hand, ML uses less area but has slightly worse timing. Both Limited and ML have acceptable runtime. Note that, although OPT is the optimal algorithm regarding our problem definition, when it is combined with the hold optimization flow, the solution found on one target will affect the optimization path in the flow, so that does not guarantee that OPT can always have the best results in terms of timing and area.

Based on our analysis of the experimental results for these seven designs, we can conclude that the machine learning based algorithm can constantly perform well in terms of quality and runtime, while the other approaches may make some large degradation on some designs.

### 3.7 Conclusion

In this chapter, we study the buffer insertion problem for hold violation removal. Three approaches are presented. One is proved to be an optimal algorithm. The second is a heuristic algorithm based on observations made on the industrial buffer library. The last is based on machine learning, which

Table 3.2: Experimental results (normalized)

Data	Method	WNS	TNS	WHS	THS	Area	Runtime
d1							
	OPT	0.00%	0.00%	0ps	0ps	-0.17%	1.25
	Limited	0.00%	-0.03%	0ps	0ps	-0.16%	1.00
	HEU	0.00%	-0.05%	0ps	0ps	-0.16%	1.04
	ML	0.00%	0.00%	0ps	0ps	-0.17%	1.00
d2							
	OPT	0.00%	0.51%	-73.65%	-99.54%	0.06%	0.81
	Limited	0.00%	-0.32%	-21.62%	-95.27%	0.08%	0.88
	HEU	0.00%	0.15%	-100.00%	-100.00%	0.07%	0.97
	ML	0.00%	0.53%	-74.32%	-99.23%	0.06%	0.91
d3							
	OPT	0.00%	-0.01%	-23.26%	-36.76%	-0.11%	0.71
	Limited	0.00%	-0.01%	-16.28%	-14.23%	-0.10%	0.96
	HEU	0.00%	-0.01%	-11.63%	-14.62%	-0.11%	1.09
	ML	0.00%	-0.02%	-20.93%	-24.90%	-0.11%	1.09
d4							
	OPT	0.00%	0.12%	-19.37%	-4.35%	0.15%	0.12
	Limited	0.00%	0.04%	-17.15%	-2.80%	-0.03%	1.03
	HEU	0.00%	0.05%	-16.88%	-3.34%	-0.07%	1.07
	ML	0.00%	0.04%	-13.87%	-1.41%	-0.10%	1.29
d5							
	OPT	0.00%	7.18%	-19.44%	-9.55%	-0.43%	1.12
	Limited	0.00%	7.01%	-20.37%	-14.43%	-0.27%	1.30
	HEU	0.00%	-0.14%	-19.44%	-17.22%	-0.34%	1.49
	ML	0.00%	7.06%	-19.44%	-14.97%	-0.39%	1.27
d6							
	OPT	0.00%	-0.22%	-91.49%	-14.46%	-0.88%	0.12
	Limited	0.00%	-0.12%	297.87%	36.89%	-0.89%	0.92
	HEU	0.00%	-0.27%	489.36%	224.51%	-0.89%	1.13
	ML	-0.05%	-0.10%	0.00%	-0.49%	-0.87%	0.87
d7							
	OPT	0.00%	0.00%	19.78%	-2.70%	-0.02%	0.11
	Limited	0.00%	-0.17%	11.00%	-50.59%	0.09%	0.78
	HEU	0.00%	0.00%	335.59%	1.30%	-0.01%	1.26
	ML	0.00%	-0.07%	3.32%	3.84%	-0.03%	0.61



adopts the logistic regression and gradient decent method. The proposed approaches are tested on industrial designs and compared to the state-of-the-art industrial hold optimization. Better results are seen on most designs. The analysis of the experimental results concludes that the machine learning based algorithm can achieve good reduction in terms of timing and area under a comparable runtime compared to the industrial hold optimization.

# CHAPTER 4

## AERIAL IMAGE SIMULATION ON MULTI-CORE SIMD CPU

### 4.1 Introduction

As semiconductor devices shrink, the lithography technology becomes increasingly complicated. To print correctly on the wafer from a mask is no longer a trivial task, especially when the feature size is significantly smaller than the wavelength of the light in lithography resulting in severe optical interference and diffraction. Therefore, aerial image simulation, which is the process of simulating light intensity on the wafer for given illumination conditions, is considered an essential step in the design for manufacturability (DFM) process. Aerial image simulation can be used in mask pattern synthesis, printability analysis, optical proximity correction, etc.

Aerial image simulation contains a huge number of numerical computations that makes it a time-consuming task. Therefore, an efficient yet accurate aerial image simulation becomes a necessity. Various approaches have been proposed to improve its efficiency with different accuracy [32, 11, 10, 33, 34, 35, 36]. Cong and Zou [10] show that the polygon-based approach can achieve comparable performance while maintaining acceptable accuracy. Therefore, we focus on the polygon-based approach in this chapter.

Due to the regularity of this problem and the extremely large volume of data, a parallel implementation method can effectively leverage today's high performance computing platforms such as multi-core CPUs, general-purpose graphic process units (GPGPUs), and field-programmable gate arrays (FPGAs), and thereby achieve great speedup. Recently, the FPGA-based approach [10] and GPU-based approach [11] have both been proposed to accelerate this polygon-based method with considerable speedup reported. FPGA programming that will reconfigure hardware for special implementations is generally harder than CPU or GPU programming, and [10] shows that many

hardware-oriented refinements need to be made to achieve good performance. Although GPGPUs provide great opportunity for parallelism, there are still many factors, as shown in [11], that impact the speed of a GPU implementation. Thus, a sequence of GPU-oriented optimizations is applied in [11] to achieve good speedup.

However, compared to the FPGA-based or GPU-based approach, not many efforts were made to improve the performance of the CPU baseline programming in the previous works. The CPU-based approach should be further optimized in terms of the total amount of computation. Note that recent CPU architectures have significant modifications toward high performance computing capabilities. But those computing capabilities require explicit tunings with respect to the SIMD (single-instruction multiple-data) extensions instructions and multi-core architectures [37, 38], and in previous works such tunings were missing for the CPU-based approach. Therefore, the comparisons of the proposed GPU-based approach or FPGA-based approach to CPU were not done thoroughly, when the baseline CPU programming did not fully exploit the computing power of modern CPUs. This has led to an under-estimation of the CPU performance when considering the question of which computing platforms should be adopted for those emerging huge problems. This is essentially a big challenge for the innovation of EDA tools to find the best integration with the runtime-efficient parallelized software and the cost-effective parallelized computing systems [39, 40, 41, 42].

In this chapter, we first present a more efficient approach by further optimizing the basic polygon-based approach in terms of the total amount of computation. Then, several implementations are proposed to accelerate the approach with multi-core SIMD CPU. We study the performance of the proposed implementations and compare them through experiments. To the best of our knowledge, the GPU-based approach in [11] is the state-of-the-art approach for polygon-based aerial image simulation. In our experiments, with a hex-core SIMD CPU, our fastest method achieves up to 73X speedup over the baseline serial approach, while the GPU-based approach in [11] achieves up to 34X speedup. The maximum absolute error of the algorithm and the GPU-based approach are both around  $10^{-6}$ , since both of them use single-precision floating point. We also extend our algorithm for double precision-floating point. Its maximum absolute error can almost be ignored (on the order of  $10^{-14}$ ), while it still manages to have a speedup up to 40X over the

base approach that is still faster than the GPU-based approach. Our results reveal that with explicit tunings, the multi-core CPU-based approach can achieve a considerable speedup.

The rest of the chapter is organized as follows. Section 4.2 introduces the background on aerial image simulation and multi-core SIMD programming. Then, our proposed approaches are presented in Section 4.3. In Section 4.4, we demonstrate our experimental result and finally Section 4.5 concludes this chapter.

## 4.2 Background

In this section, we will briefly introduce some background on aerial image simulation and multi-core SIMD CPU programming.

### 4.2.1 Aerial Image Simulation

Aerial image simulation is a fundamental problem in the regular lithography-related process. The task is to compute the light intensity distribution based on the illumination conditions. Although the fundamental physical behavior is extremely complicated during the illumination, to perform a full-chip simulation for aerial image the whole imaging process could be simplified as:

$$I(x, y) = \sum_i \lambda_i \times |\phi_i(x, y) \otimes f(x, y)|^2 \quad (4.1)$$

in which  $I(x, y)$  is the target aerial image,  $f(x, y)$  represents the transparency of the mask at location  $(x, y)$ <sup>1</sup>,  $\phi_i(x, y)$  is the  $i$ -th order complex convolution kernel and  $\lambda_i$  is the corresponding weight [32].  $\otimes$  denotes convolution and  $|\cdot|$  takes the modulus of a complex number. Normally for each illumination process, the optical systems are kept constant and the process parameters remain unchanged, the convolution kernel  $\phi_i$  and the weight factor  $\lambda_i$  are kept constant. So, once the optical models are solidly built, the remaining variable for the aerial image simulation is the mask  $f(x, y)$ , which could be 1 or 0, depending on the transparency of the field and printed feature.

---

<sup>1</sup>For thin mask modeling, 1 represents transparent region and 0 represents opaque regions.

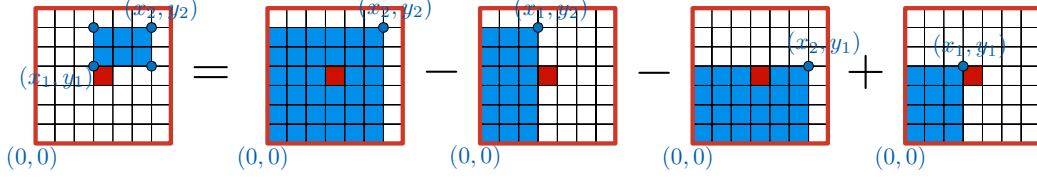


Figure 4.1: Rectangle decomposition. The impact of  $R((x_1, y_1) - (x_2, y_2))$  on the central pixel (the red square) can be calculated by looking up the impacts of the four shaped rectangles together[11].

The aerial image simulation could be simply described as:

***Aerial Image Simulation:***

*With a constant set of kernel  $\phi_i$  and weighting factor  $\lambda_i$ , generate the  $I(x, y)$  with any given mask  $f(x, y)$ .*

Because it is very costly to directly apply convolution for the full chip simulation, and mask features are usually Manhattan-type, the aerial image can be processed in a so-called polygon-based approach [32]. The idea is to pre-compute and store the convolution of certain basic rectangles into a lookup table, so the impact of any rectangle on a pixel could be calculated by only four times table lookup. Fig. 4.1 illustrates this operation. A lookup table  $T$  corresponds one convolution kernel, and the value of  $T(x, y)$  is the impact of a rectangle  $R((0, 0) - (x, y))$ , where  $(0, 0)$  is the left-bottom corner and  $(x, y)$  is the right-top corner of  $R$ , on the center pixel (the red dot in Figure 4.1). The operation of computing the impact of a rectangle  $R((x_1, y_1) - (x_2, y_2))$  on the center pixel is as follows:

$$impact = T(x_2, y_2) + T(x_1, y_1) - T(x_2, y_1) - T(x_1, y_2) \quad (4.2)$$

which consists of four table lookups and three floating point arithmetic operations. In order to compute the impact for a rectangle on any pixel, we need to calculate the coordinates of the four corners of the rectangle relative to the lookup table, as if the pixel was located at the center of the lookup table. This approach substantially reduces the runtime by avoiding the repetitive computation of convolutions. Although it is very complicated to set up the modeling table, once it is generated off-line, the LUT will remain constant for all the polygons in the mask. Details of this approach can be found in [33].

When calculating the coordinates of the corners of the rectangle relative

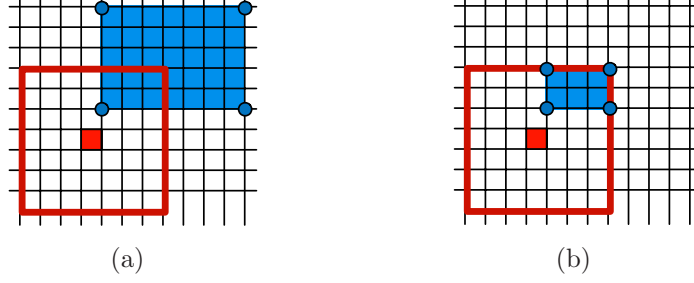


Figure 4.2: Rectangle (blue color) extending outside the lookup table in (a) is truncated at the boundary of the lookup table in (b).

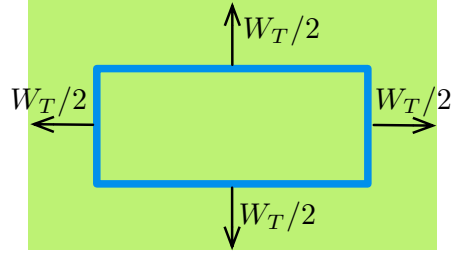


Figure 4.3: Example of a rectangle (blue rectangle) and its corresponding impact region (green zone).

to the lookup table, if part of the rectangle is outside the lookup table, then the rectangle has to be truncated at the boundary of the lookup table. The reason is the part of the rectangle that is outside the lookup table will not contribute any impact on that pixel. An example is illustrated in Figure 4.2. Furthermore, if one rectangle is totally outside the lookup table, it is unnecessary to compute the impact for that pixel, since the impact of the rectangle on that pixel is always 0. Only those pixels that will be impacted by the rectangle have to apply the above operation. Thus, supposing a lookup table has size  $W_T \times W_T$ , then an impact region for a rectangle can be defined by extending the rectangle outward by  $W_T/2$  (see Figure 4.3). For a rectangle, we only need to compute its impact on the pixels inside its impact region. Therefore, to compute the impact of a rectangle on the image, a naive implementation is to iterate all the pixels inside the impact region, and for each of the pixels, the above operation has to be applied to compute the impact of the rectangle on the pixel.

Suppose there are  $K$  convolution kernels each of which is complex. So, we will have  $2K$  lookup tables of which  $K$  tables correspond to the real

component and  $K$  tables correspond to the imaginary component. Since Equation (4.1) is nonlinear, we need to process each table separately, and take the square of them and sum them up in the end. A basic implementation of the polygon-based approach is shown in Algorithm 4.1. It serves as a base approach of our study.

---

**Algorithm 4.1** Polygon-based algorithm: base algorithm

---

**Input:**  $2K$  lookup tables  $T_k$ , each of which has size  $W_T \times W_T$ , and  $2K$  weights  $\lambda_k$ . A set of rectangles represented by the coordinates of its top, bottom, left and right boundaries.

**Output:** Aerial Image: a 2D array  $I$ , size  $W_I \times H_I$ .

```

1: for  $k = 0$  to  $2K - 1$  do
2:   for each rectangle  $R$  do
3:     Impact region  $G \leftarrow$  extend  $R$  by  $W_T/2$ 
4:     for each pixel  $(x, y)$  inside the impact region do
5:        $x_{ori} = x - W_T/2$  and  $y_{ori} = y - W_T/2$ 
6:        $R' \leftarrow$  compute the relative positions of  $R$  to  $(x_{ori}, y_{ori})$ 
7:        $R_t \leftarrow$  truncate  $R'$  at four sides of the lookup table
8:        $I'[k][y][x] += T_k[R_t \cdot top][R_t \cdot right]$ 
                         $- T_k[R_t \cdot top][R_t \cdot left]$ 
                         $- T_k[R_t \cdot bottom][R_t \cdot right]$ 
                         $+ T_k[R_t \cdot bottom][R_t \cdot left]$ 
9:     end for
10:  end for
11:  for each pixel  $(x, y)$  in the image do
12:     $I[y][x] = I[y][x] + \lambda_k \cdot (I'[k][y][x])^2$ 
13:  end for
14: end for

```

---

#### 4.2.2 Multi-Core SIMD CPU Programming

Modern CPUs support parallelism at two levels: instruction level and thread level. At the instruction level, CPUs support the single instruction multiple data (SIMD) computation model through the Streaming SIMD Extensions (SSE) instruction set. The main idea of SSE is to load multiple data into 128-bit wide registers and use special instructions to simultaneously perform computations on the multiple data in these registers. Figure 4.4 gives an example of SSE. We load four 32-bit floating point numbers X0, X1, X2, X3 to one 128-bit register and four 32-bit numbers Y0, Y1, Y2, Y3 to another

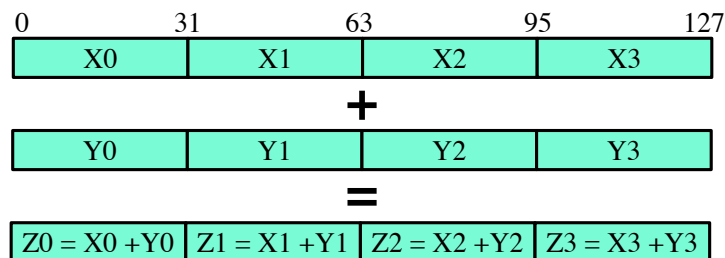


Figure 4.4: An illustration of using SSE so that all four elements inside the register are computed in parallel.

register. We can then perform additions simultaneously on the four pairs of floating point numbers by the use of a single SSE instruction. Consider that we would need four regular instructions to accomplish the same task; then using SSE gives us a 4X speedup.

To effectively use SSE, memory access from the CPU should be aligned to the 16-byte boundaries whenever possible [43]. Unaligned memory access will lead to extra memory access and additional operations to combine data from memory into the 128-bit register. Thus, reasonable attempts to align commonly used data sets are required to avoid some performance penalties.

At the thread level, parallelism can be realized in coarser grain. Modern CPUs are usually multi-core. When multiple threads are created, they can run on different cores of a CPU, achieving parallel computation. Such multi-threading can be implemented through APIs such as OpenMP. OpenMP is a shared-memory threading API that standardizes task and loop level parallelism [44]. Due to the behavior of the shared-memory multi-threading, programmers need to be careful to avoid race conditions and synchronization issues. Besides, different threads accessing the same memory address region should be avoided, since false sharing happens when different threads simultaneously access the same cache line. False sharing degrades the performance advantage provided by multi-threading.

### 4.3 Implementation Methods on Multi-Core SIMD CPU

In this section, we first present a more efficient sequential approach by further improving the base approach. Then, we show how to apply SSE and multi-



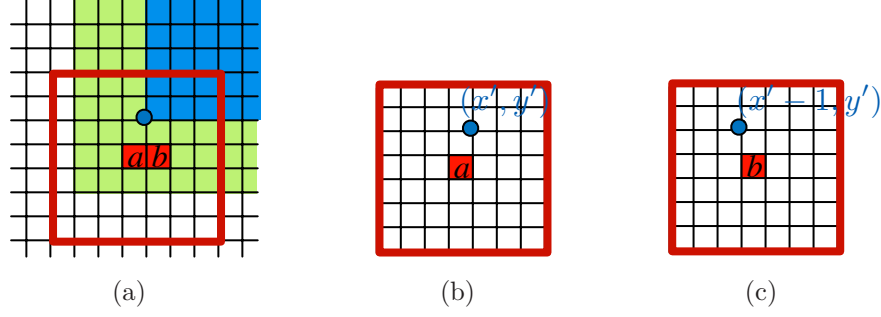


Figure 4.5: An example of computing the relative coordinates for the consecutive pixels. (a) The relative positions of the left-bottom corner of the blue rectangle for the two consecutive pixels  $a$  and  $b$  are shown in (b) and (c), respectively.

threading on our approaches.

#### 4.3.1 An Improved Approach

Based on the observations we made on the base approach, there are two directions that we use to improve the base approach in terms of the total amount of computations. The first one is to reduce the computation on calculating the coordinates of the corners of a rectangle relative to the lookup table (lines 5-6 in Algorithm 4.1). Such a procedure is repeated for every pixel inside the impact region in the base approach. However, most of the pixels can avoid this procedure, if we can carefully reuse the relative position obtained from the previous pixel for the next pixel, since the relative coordinates of one rectangle corner for the consecutive pixels would be on continuous locations of the lookup table as well. Let us take Figure 4.5(a) as an example. Suppose we work on pixel  $a$  first, and then pixel  $b$ . For the left-bottom corner of the rectangle (the blue rectangle), its coordinate relative to the lookup table is  $(x', y')$  when  $a$  is put on the center (see Figure 4.5(b)). For the same corner, the relative coordinate for  $b$  is  $(x' - 1, y')$  as shown in Figure 4.5(c). We can see these two blue dots  $((x', y')$  and  $(x' - 1, y')$ ) are located at the continuous locations of the lookup table, so the relative position for  $b$  can be obtained by simply decrementing  $x'$  that was already calculated for  $a$ . Therefore, there is no need to compute the relative positions of the four corners for every pixel. Starting at the left-bottom pixel in the impact region, we only need to compute the relative position  $(x', y')$  for the

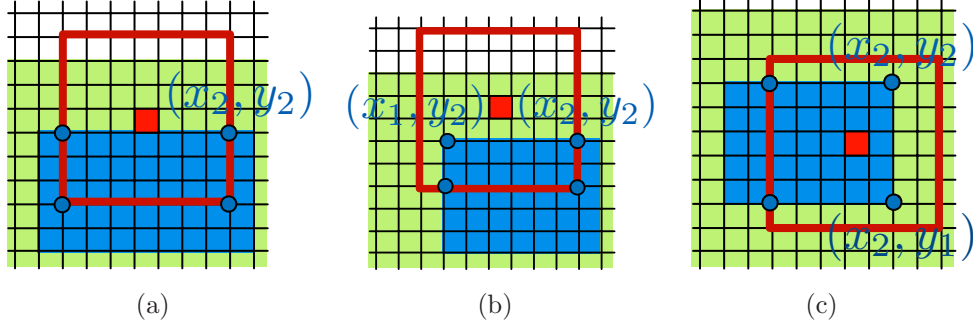


Figure 4.6: Impact of the blue rectangle on the pixel (the red dot): (a)  $T(x_2, y_2)$ ; (b)  $T(x_2, y_2) - T(x_1, y_2)$ ; (c)  $T(x_2, y_2) - T(x_2, y_1)$ .

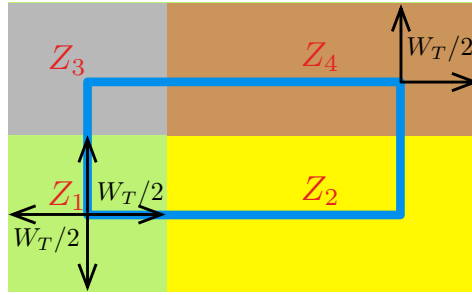


Figure 4.7: The impact region is decomposed into four regions:  $Z_1$  (the green zone),  $Z_2$  (the yellow zone),  $Z_3$  (the gray zone), and  $Z_4$  (the brown zone).

starting pixel, and then when traversing the impact region from left to right and bottom to top, the relative position for a new pixel can be obtained by decrementing  $x'$  and  $y'$ , respectively, so a certain number of computations used in calculating the relative positions can be saved compared to the base approach.

Next, the values obtained from the table lookups based on some rectangle corners relative to some pixels are always 0. Calculating the coordinates of those corners relative to the lookup table is redundant and should be avoided. Figure 4.6 gives three examples in which only the right-top corner has impact on the pixel (see Figure 4.6(a)), only the right-top corner and the left-top corner have impacts on the pixel (see Figure 4.6(b)), and only the right-top corner and the right-bottom corner have impacts on the pixel (see Figure 4.6(c)). Based on these observations, for a rectangle, its impact region can be further decomposed into four small regions as shown in Figure 4.7, pixels in  $Z_1$  will be impacted by the four corners of the rectangle, pixels

---

**Algorithm 4.2** the improved (IMP) approach

---

```
1: for  $k = 0$  to  $2K - 1$  do
2:   for each rectangle  $R$  do
3:     Compute  $Z_1, Z_2, Z_3, Z_4$ , and the impact region  $G$ 
4:      $p_s \leftarrow (G \cdot \text{left}, G \cdot \text{bottom})$  // starting pixel
5:      $(x_c[4], y_c[4]) \leftarrow$  compute the relative positions for the four corners
      for  $p_s$  (index 0: left-bottom corner, 1: right-bottom corner, 2: left-top
      corner, 3: right-top corner)
6:     for  $y = G \cdot \text{bottom}$  to  $G \cdot \text{top}$  do
7:        $(x'_i, y'_i) \leftarrow (x_c[i], y_c[i])$  for  $i = 0, 1, 2, 3$ 
8:       for  $x = G \cdot \text{left}$  to  $G \cdot \text{right}$  do
9:         if  $(x, y)$  inside  $Z_1$  then
10:           $(x'_i, y'_i) \leftarrow$  move  $(x_c[i], y_c[i])$  to the nearest boundary for  $i =$ 
            0, 1, 2, 3
11:           $I'[k][y][x] += (T_k[y'_0][x'_0] - T_k[y'_1][x'_1] - T_k[y'_2][x'_2] + T_k[y'_3][x'_3])$ 
12:          else if  $(x, y)$  inside  $Z_2$  then
13:             $(x'_i, y'_i) \leftarrow$  move  $(x_c[i], y_c[i])$  to the nearest boundary for  $i =$ 
              1, 3
14:             $I'[k][y][x] += (-T_k[y'_1][x'_1] + T_k[y'_3][x'_3])$ 
15:            else if  $(x, y)$  inside  $Z_3$  then
16:               $(x'_i, y'_i) \leftarrow$  move  $(x_c[i], y_c[i])$  to the nearest boundary for  $i =$ 
                2, 3
17:               $I'[k][y][x] += (-T_k[y'_2][x'_2] + T_k[y'_3][x'_3])$ 
18:            else
19:               $(x'_3, y'_3) \leftarrow$  move  $(x_c[3], y_c[3])$  to the nearest boundary
20:               $I'[k][y][x] += T_k[y'_3][x'_3]$ 
21:            end if
22:             $x'_i = x'_i - 1$  for  $i = 0, 1, 2, 3$ 
23:          end for
24:           $y'_i = y'_i - 1$  for  $i = 0, 1, 2, 3$ 
25:        end for
26:      end for
27:    for each pixel  $(x, y)$  in the image do
28:       $I[y][x] = I[y][x] + \lambda_k \cdot (I'[k][y][x])^2$ 
29:    end for
30: end for
```

---

in  $Z_2$  will be impacted by the right-bottom corner and the right-top corner, pixels in  $Z_3$  will be impacted by the left-top corner and the right-top corner, and pixels in  $Z_4$  will be only impacted by the right-top corner. So, according to which region the pixel is located in, only the corresponding corners have to be calculated. Compared to the base approach that computes four corners for a pixel that is inside its impact region, we only compute the corners if necessary.

We call this approach the *improved (IMP)* approach, and the algorithm is as shown in Algorithm 4.2. Notice that, if the relative location for one pixel is outside the lookup table, the relative location has to be moved to the nearest boundary of the lookup table, as the truncation does.

### 4.3.2 Four Lookup Tables per SSE Approach

One approach to adopt SSE in the aerial image simulation is that each SSE instruction processes four lookup tables simultaneously. We call this approach the *four lookup tables per SSE (FTPS)* approach. In the sequential algorithm, we always have the following operations:

$$I'[k][y][x] = I'[k][y][x] + T_k[y'][x'] \quad \forall k = 0, \dots, 2K - 1 \quad (4.3)$$

where  $I'[k]$  represents the image for the  $k$ -th lookup table,  $T_k$ . The algorithmic operation can be either addition (as in Equation. (4.3)) or subtraction. With the SSE instruction set, we can pack images  $k, k + 1, k + 2$ , and  $k + 3$  at pixel  $(x, y)$  to one register, and then pack lookup tables  $k, k + 1, k + 2$ , and  $k + 3$  at location  $(x', y')$  to the other register. Then we can perform one SSE addition/subtraction on these two registers, and write the final result back to the image  $k, k + 1, k + 2$ , and  $k + 3$  at pixel  $(x, y)$ . The process is illustrated in Figure 4.8, and four copies of the image (one copy for one table) have to be kept in the memory during the program execution. Then such a process can be done for  $k = 0, 4, 8, \dots$ , etc., so that Equation (4.3) is vectorized to boost the performance.

However, when we load/store data between memory and SSE registers, we need to be very careful with the data layout. If the data layout of the image is arranged just like the one shown in Figure 4.8 (where the table lookup index is the first dimension in the array), then four individual accesses are

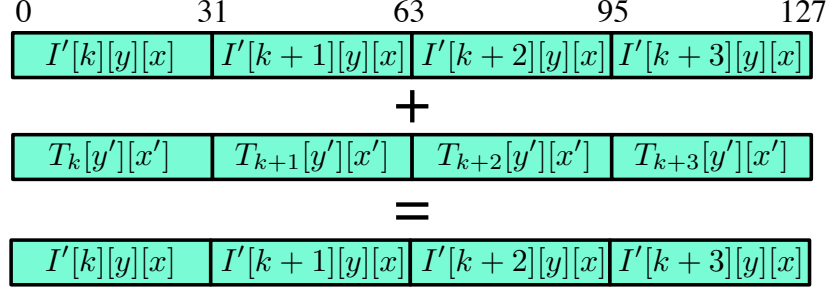


Figure 4.8: Each SSE instruction updates four images at pixel  $(x, y)$  for four corresponding lookup tables.

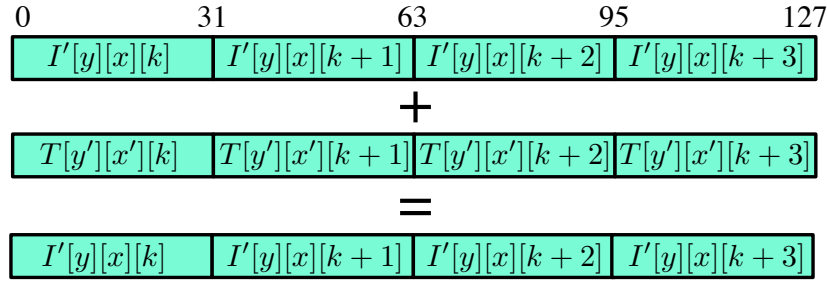


Figure 4.9: Rearrange the data layouts of the images and lookup tables, so one SSE instruction can load/store the register from/to memory.

needed in order to pack images  $k$ ,  $k+1$ ,  $k+2$ , and  $k+3$  at pixel  $(x, y)$  to one register, which will drastically hurt the performance. The images  $k$ ,  $k+1$ ,  $k+2$ , and  $k+3$  at pixel  $(x, y)$  have to be stored in continuous memory locations and aligned with 16-byte boundaries, so only one SSE instruction is enough to load/store them between memory and an SSE register. Similarly, the same procedure has to be done for lookup tables  $k$ ,  $k+1$ ,  $k+2$ , and  $k+3$  at location  $(x', y')$ . Therefore, by rearranging the data layout of the images and the lookup tables, we can have the process in Figure 4.9 in which one SSE instruction is for loading, one for addition, and another for storing. Furthermore, if the total number of the lookup tables is not a multiple of four, padding it into a multiple of four is needed to enforce the alignment. Notice that we need to combine the images for the lookup tables into the final aerial image (see lines 11-13 in Algorithm 4.1). The squaring and weight multiplying of the images for the lookup tables can be done in parallel by SSE instructions, however, the summation back to the final image cannot be done in parallel since only one pixel of the image is updated at a time. Note that the base approach and the improved approach both can apply this

$$\begin{array}{cccc}
0 & 31 & 63 & 95 & 127 \\
\boxed{I'[k][y][x]} & \boxed{I'[k][y][x+1]} & \boxed{I'[k][y][x+2]} & \boxed{I'[k][y][x+3]} & \\
+ & & & & \\
\boxed{T[k][y'][x']} & \boxed{T[k][y'][x'-1]} & \boxed{T[k][y'][x'-2]} & \boxed{T[k][y'][x'-3]} & \\
= & & & & \\
\boxed{I'[k][y][x]} & \boxed{I'[k][y][x+1]} & \boxed{I'[k][y][x+2]} & \boxed{I'[k][y][x+3]} & 
\end{array}$$

Figure 4.10: Each SSE instruction updates four continuous pixels that start from  $(x, y)$  for the  $k$ -th lookup table.

FTPS technique.

### 4.3.3 Four Pixels per SSE Approach

SSE instructions can be used to enhance the improved approach in the way that four continuous pixels are updated at the same time. Notice that this technique cannot be applied on the base algorithm, since only the improved approach is able to process the continuous pixels by simply shifting the corresponding relative positions in the lookup table, while the base approach has to recompute the relative coordinates for each pixel. We call this approach the *four pixels per SSE (FPPS)* approach.

We use Figure 4.5 as an example. Suppose the coordinate of the pixel  $a$  is  $(x, y)$ , the relative position of the left-bottom corner of the rectangle for  $a$  is  $(x', y')$ , and we are processing the  $k$ -th lookup table. We want to update the four consecutive pixels from  $a$  at the same time. With SSE instructions, we can pack the image pixels  $(x, y)$ ,  $(x + 1, y)$ ,  $(x + 2, y)$ , and  $(x + 3, y)$  to one register, pack the  $k$ -th lookup table at locations  $(x', y')$ ,  $(x' - 1, y')$ ,  $(x' - 2, y')$ ,  $(x' - 3, y')$  to the other register, perform an SSE algorithmic operation on them, and then write the result back to the image at pixels  $(x, y)$ ,  $(x + 1, y)$ ,  $(x + 2, y)$ , and  $(x + 3, y)$ . The process is illustrated in Figure 4.10. Such a process can be repeated for every four continuous pixels until all pixels in this row of the impact region are completed, and then started again on the next row until all pixels in the impact region are processed. However, in order to effectively utilize SSE instructions, we applied several techniques.

First, we have to store the lookup table in a reverse order, since the lookup

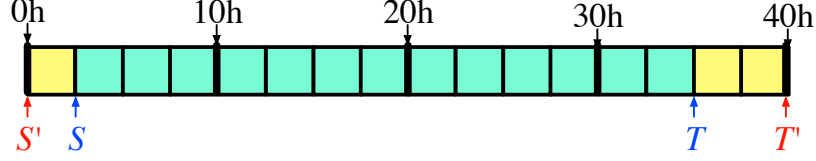


Figure 4.11: An example shows the memory address of the consecutive pixels are not aligned with the 16-byte boundary, where green squares represent the memory address for a row of image pixels in the impact region.

table is accessed in a backward direction. Second, if the total number of pixels in the image is not a multiple of 4, padding it into a multiple of four is necessary. However, this padding cannot guarantee the memory address that each SSE instruction accesses is always aligned with a 16-byte boundary. Since there are a lot of memory accesses to the pixels, it will hurt the performance when SSE instructions access a lot of unaligned memory (individual accesses have to be used). So in our implementation, we enforce SSE to make aligned memory access. Figure 4.11 can be used as an example to illustrate it. For a row of pixels that we are interested in accessing,  $S$  and  $T$  are the starting address and the ending address of the row of pixels, respectively. With simple calculations,  $S$  can be moved backward to  $S'$  and  $T$  can be moved forward to  $T'$ , where  $S'$  and  $T'$  are aligned with the 16-byte boundary and are the nearest address to  $S$  and  $T$ , respectively. Then, we let SSE update the pixels between  $S'$  and  $T'$ , so there is no unaligned memory access. However, some pixels at the first four or the last four will get unnecessary updates (see yellow squares in Figure 4.11). We then apply an undo procedure on the first four and the last four pixels, respectively, where the procedure works on the four pixels simultaneously with SSE instructions and some predefined 128-bit masks. Note that this unaligned issue also exists for the lookup tables, so the same techniques are applied on them as well.

Although we make sure every SSE instruction always makes memory access aligned with the 16-byte boundary, it is still possible that the address of  $I'[k][y][x]$  is not aligned with the address of  $T[k][y'][x]$ , which means we need more than one SSE instruction to perform the addition/subtraction on them. An example is illustrated in Figure 4.12, where the memory access to the image  $I$  and the lookup table  $T$  are both aligned with the 16-byte boundary (from 0h to 40h). By making one SSE access to load the data from memory

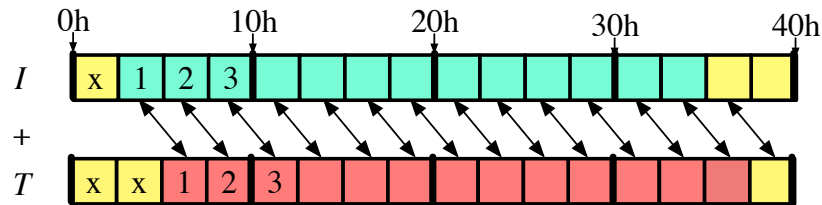


Figure 4.12: The address of the image is not aligned with the address of the lookup table, while their memory accesses are both aligned with 16-byte boundary.

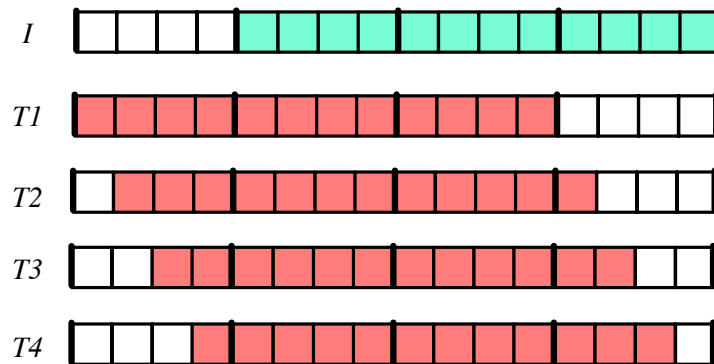


Figure 4.13: An example that shows the four possible behaviors regarding the alignment between an image and a lookup table.



to the registers, we have  $I(x, 1, 2, 3)$  and  $T(x, x, 1, 2)$  on the registers, where  $x$  represents the unnecessary access due to the alignment technique presented above. Then one SSE addition will only get us  $I(x, 1, 2, 3) + T(x, x, 1, 2)$  which is apparently wrong, since the correct operation is  $I(1, 2, 3, 4) + T(1, 2, 3, 4)$ . So only when both addresses are aligned with each other (as shown in Figure 4.10), can we use one SSE instruction to get the correct result. Therefore, a lookup table duplication procedure is applied here to guarantee that the memory access to the image and the lookup table are always aligned with each other. The procedure is for a lookup table; three additional lookup tables are generated by shifting the lookup table right by 1, 2, and 3 positions, respectively. In Figure 4.13, by shifting  $T1$  right by 1, 2, and 3 positions, we can have  $T2, T3$  and  $T4$ , respectively. And  $I$  is aligned with  $T1$ , misaligned 3 positions with  $T2$ , misaligned 2 positions with  $T3$ , and misaligned 1 position with  $T4$ . So, for any image address, one lookup table that is aligned with the image address must exist among these four lookup tables. For the example shown in Figure 4.12, the lookup table obtained by shifting  $T$  right by 3 positions will be aligned with  $I$ . Therefore, for any image, we can always select a lookup table that is aligned with the image, so that we can use only one SSE instruction to do the addition/subtraction and still get the correct result. Notice that since the size of the lookup table is quite small compared to the image size, this lookup table duplication procedure will not take much memory space, and this space is also reused for different tables.

Notice that it is possible that  $T[k][y'][x' - 3]$  and  $T[k][y'][x']$  are both outside of the lookup table, which means we can directly pack the value at the nearest boundary to the register. It is also possible that not all of them are outside the boundary; in this case, techniques similar to those of the undo procedure can be used here to guarantee correctness.

#### 4.3.4 Multi-Threading

There are two approaches to apply multi-threading. One approach is to let each thread handle one lookup table. Each thread owns a copy of the image for the lookup table, and then each thread updates its own image and accesses its lookup table without worrying about some common multi-threaded problems such as race conditions and false sharing. A synchronization barrier

is necessary before combining all images to the final image, which can guarantee that all threads are done computing their own images. When combining images, we let each thread handle a subset of the image, like tiling the image. We cannot afford to create a number of threads equal to the number of lookup tables, since the number of lookup tables is larger than the number of CPU cores. So we have to let each thread handle a subset of tables; however, we do not need to make any adjustment for it. Each thread still keeps its own copy of the image, so that the images for the lookup table will be summed sequentially, and the correctness of the final image is still guaranteed. We call this approach *Multi-threading by table (MTT)*.

The other approach is to let one thread be responsible for one row of pixels when updating the impact region. Since each row updating is independent, every thread is updating the same image but in different rows. So we don't need to worry about any synchronization issue. The final image summation can also be paralleled by the same philosophy, that is, each thread handles one row of the image. This approach is called *Multi-threading by row (MTR)* and requires less memory than the MTT approach.

## 4.4 Experimental Results

Various methods are implemented based on the approaches we presented in the previous sections, and we compare them in our experiments. To compare with the GPU-based approach, the source code of the fastest approach proposed in [11] is directly used in our experiment. The experiments are performed on a machine with a dual-socket, six-core 2.67GHz Intel Xeon processor, 12GB of memory, and a NVIDIA Tesla C2050 GPU. As for input data, we use the data set presented in [11]. There are a total of 26 lookup tables, each of which has size  $257 \times 257$ . Three data sets are used in our experiment: a *small* set, a *medium* set, and a *large* set. Each set contains four data, and the details of each data, such as the number of rectangles and the number of pixels in the image, are shown in Table 4.1.

Based on the approaches we presented in the previous section, we can have different methods with different configurations. Each configuration has three setups: 1) choose an underlying approach, then apply 2) which SSE approach and 3) which multi-threading approach on top of the underlying

Table 4.1: Data set for our experiments

data	#Rectangles	#Pixels
small_1	209	$463 \times 275$
small_2	212	$438 \times 275$
small_3	217	$413 \times 275$
small_4	225	$463 \times 275$
medium_1	1104	$1025 \times 775$
medium_2	1297	$1055 \times 775$
medium_3	1214	$1025 \times 775$
medium_4	1292	$1025 \times 775$
large_1	4904	$2025 \times 1525$
large_2	4897	$2080 \times 1525$
large_3	4892	$2025 \times 1525$
large_4	4814	$2025 \times 1525$

Table 4.2: Configuration setups and the options

Setup	Options
Underlying approach	the base approach or the improved (IMP) approach
SSE	none, the FTPS approach, or the FPPS approach
Multi-threading (MT)	none, the MTT approach, or MTR approach

approach. The options for these setups are listed in Table 4.2. Suppose a method uses the base approach as the underlying approach, and then applies FPPT and MTR; such method is then called Base+FPPT+MTR in the rest of this section.

In order to extensively test the proposed approaches, we implement 11 methods, each of which has its own configuration. The comparison of run-time and speedup for these methods are shown in Table 4.3 and Table 4.4, respectively. Note that only the base approach uses double-precision floating points which serves as the base of the comparison. The other methods all use single-precision floating points, including the GPU-based approach. The maximum absolute error among the pixels of the image is measured as well. All of these methods (excluding the base approach) have the maximum absolute error on the order of  $10^{-6}$ .

For the sequential approach, the IMP approach achieves a 4X speedup over the base approach. IMP apparently has fewer computations than the base approach as we discussed previously. Among those one-threaded approaches that applies with SSE, the IMP+FTPS approach is the fastest one on all the

data sets. Obviously that IMP+FTPS is more efficient than Base+FTPS due to the difference between the efficiency of the underlying approaches. Comparing IMP+FTPS with IMP+FPPS, there is a certain amount of overhead for FPPS to make aligned memory access (i.e. calculating aligned memory address, undo those unnecessary updates, etc.), while FTPS just simply rearranges the data layout of the image and the lookup table to achieve aligned memory access, so IMP+FTPS runs slightly faster (IMP+FTPS has 10X speedup and IMP+FPPS has 8.6X speedup). However, we have to keep in mind that FTPS requires more memory space than FPPS since it has to keep four copies of images for the four lookup tables during the program execution. IMP+FTPS and IMP+FPPS have a 2.5X and 2.1X speedup over the IMP approach, respectively, while the theoretical performance upper bound of SSE is 4X. The speedup is acceptable since the whole program does not use all the SSE instructions and there is some overhead in order to utilize SSE.

Now, let us see the performance of those multi-threaded approaches. We include a very simple multi-threaded approach (Base+MTT) in the experiment, so we can see that although Base+MTT runs with multiple threads, its performance is worse than Base+FTPS that only has one thread. This shows careful tuning is necessary in order to boost the performance. We believe that the key reason Base+FTPS outperforms Base+MTT is that FTPS always reallocates/rearranges the data layout before using them, which ensures those memory accesses are in a contiguous fashion so that they would likely exist in the cache; however, those preallocated lookup tables accessed by MTT might be far away from each other. For those approaches that apply both SSE and multi-threading, IMP+FTPS+MTR is the fastest, achieving up to 70X speedup. FTPS+MTR provides a good balance in terms of parallelizing multiple images (for different tables) and multiple pixels on one image, while FTPS+MTT only parallelizes multiple images, making it less parallel. FPPS+MTT and FPPS+MTR both have a certain amount of overhead caused by FPPS. Another thing is that the performance of IMP+FPPS+MTT decreases as the input size increases, while the performance of IMP+FPPS+MTR increases as the input size increases. That is because MTR can run with more threads as the input size increases, while the number of threads for MTT is always set to the number of lookup tables. Moreover, in terms of the memory used, IMP+FPPS+MTR uses less

than IMP+FTPS+MTR (since FTPS needs four copies of the image for four lookup tables), while IMP+FPS+MTR only needs to keep four copies of a lookup table whose size is usually much smaller than the image size. Note that the memory of the MTT-based approach is much larger than that of the MTR-based approach (since each thread owns one copy of the image in MTT), making it a poor candidate when the input size is large.

For fair comparisons, the GPU-based approach and our approaches all use single-precision floating points, since GPU has much higher computing capacities when using single-precision points. Our approaches that use both SSE and multi-threading have a runtime comparable to or smaller than the GPU-based approach. The GPU-based approach achieves an average speedup 31X, while the average speedups achieved by IMP+FPS+MTR and IMP+FTPS+MTR are 39X and 64X, respectively. Thus, our fastest approach achieves 2X speedup over the GPU-based approach. Furthermore, we extend the IMP+FTPS+MTR approach to use double-precision floating points. Theoretically, the performance of using double-precision floating points would be cut in half since one SSE instruction only takes two double-precision points instead of four single-precision. The extended approach achieves an average 36X speedup over the base approach which meets our theory, and it still runs faster than the GPU-based approach. Its maximum absolute error is on the order of  $10^{-14}$ , which can simply be ignored. Therefore, there is a tradeoff between precision and speedup.

## 4.5 Conclusion

In this chapter, we first present a more efficient approach than the straightforward serial approach. Then, we present several approaches that apply SSE and multi-threading. On a hex-core SIMD CPU, our fastest method achieves a speedup up to 73X over the baseline serial approach, and 2X speedup over the GPU-based approach. This also demonstrates that an explicit tuning is necessary in order to fully exploit the computing power of modern CPUs. Furthermore, with the latest Advanced Vector Extensions (AVX) instruction set that introduces 256-bit registers, we can expect a larger speedup by applying AVX to our approach.

Table 4.3: Runtime comparison. Unit is second.

data/setup	Method														GPU [11]
	Underlying approach	Base	IMP	Base	IMP	Base	IMP	Base	IMP	Base	IMP	IMP	IMP	IMP	
	SSE	none	none	F'TPS	F'TPS	F'TPS	F'TPS	F'TPS	F'TPS	F'TPS	F'TPS	F'TPS	F'TPS	F'TPS	
	MT	none	none	none	none	none	none	MTT	MTT	MTT	MTT	MTT	MTT	MTT	
	small_1	3.22	0.79	0.47	0.31	0.34	0.34	0.63	0.098	0.078	0.047	0.065	0.134	0.103	
	small_2	3.24	0.8	0.46	0.32	0.4	0.4	0.59	0.099	0.077	0.046	0.077	0.104	0.103	
	small_3	3.16	0.76	0.46	0.3	0.32	0.32	0.58	0.097	0.075	0.045	0.062	0.112	0.121	
	small_4	3.52	0.86	0.5	0.35	0.37	0.37	0.67	0.108	0.086	0.050	0.069	0.108	0.124	
	medium_1	13.21	3.32	1.94	1.34	1.62	1.62	2.29	0.536	0.436	0.290	0.396	0.376	0.386	
	medium_2	16.07	4	2.35	1.69	1.95	1.95	2.77	0.569	0.510	0.307	0.462	0.448	0.467	
	medium_3	15.11	3.77	2.22	1.51	1.84	1.84	2.64	0.585	0.501	0.314	0.440	0.431	0.451	
	medium_4	15.17	3.78	2.22	1.52	1.85	1.85	2.62	0.593	0.495	0.315	0.449	0.444	0.461	
	large_1	106.47	26	15.18	10.41	12.59	12.59	17.83	3.21	2.86	1.54	3.07	2.28	3.31	
	large_2	112.86	27.4	16.05	10.92	13.27	13.27	18.79	3.40	3.19	1.54	3.13	2.37	3.62	
	large_3	114.19	27.71	16.18	11.07	13.48	13.48	19.02	3.47	3.14	1.58	3.18	2.41	3.72	
	large_4	112.86	27.4	16.05	10.92	13.27	13.27	18.79	3.40	3.19	1.54	3.13	2.37	3.59	

Table 4.4: Speedup comparison. Speedup = runtime of base / runtime of the method.

data/setup	Method												GPU [11] <sup>a</sup>
	Base	IMP	Base	IMP	Base	IMP	Base	IMP	Base	IMP	IMP	IMP	
Underlying approach	Base	IMP	Base	IMP	Base	IMP	Base	IMP	Base	IMP	IMP	IMP	
SSE	none	none	FTPS	FTPS	FTPS	FTPS	FTPS	FTPS	FTPS	FTPS	FTPS	FTPS	
MT	none	none	none	none	MTT	MTT	MTR	MTR	MTT	MTT	MTR	MTR	
small_1	1X	4.08X	6.85X	10.39X	5.15X	9.47X	32.81X	68.97X	41.34X	49.82X	23.97X	31.32X	
small_2	1X	4.05X	7.04X	10.13X	5.46X	8.10X	32.68X	70.01X	42.24X	41.99X	31.17X	31.43X	
small_3	1X	4.16X	6.87X	10.53X	5.46X	9.88X	32.54X	70.62X	42.19X	51.15X	28.22X	26.20X	
small_4	1X	4.09X	7.04X	10.06X	5.22X	9.51X	32.68X	69.87X	41.16X	50.73X	32.52X	28.39X	
medium_1	1X	3.98X	6.81X	9.86X	5.77X	8.15X	24.65X	45.48X	30.33X	33.38X	35.14X	34.22X	
medium_2	1X	4.02X	6.84X	9.51X	5.80X	8.24X	28.24X	52.32X	31.50X	34.81X	35.89X	34.44X	
medium_3	1X	4.01X	6.81X	10.01X	5.73X	8.21X	25.85X	48.11X	30.17X	34.33X	35.08X	33.51X	
medium_4	1X	4.01X	6.83X	9.98X	5.78X	8.20X	25.60X	48.16X	30.66X	33.77X	34.13X	32.90X	
large_1	1X	4.10X	7.01X	10.23X	5.97X	8.46X	33.17X	69.35X	37.28X	34.66X	46.75X	32.13X	
large_2	1X	4.12X	7.03X	10.34X	6.01X	8.50X	33.18X	73.14X	35.42X	36.10X	47.54X	31.18X	
large_3	1X	4.12X	7.06X	10.32X	6.01X	8.47X	32.91X	72.50X	36.32X	35.95X	47.30X	30.67X	
large_4	1X	4.12X	7.03X	10.34X	6.01X	8.50X	33.18X	73.14X	35.42X	36.10X	47.54X	31.42X	

<sup>a</sup>Notice that we only observe around 33.5X speedup on the GPU implementation, which is lower than the 60X speedup claimed in [11]. The programs we use for the base approach and the GPU-based approach and the data we test are the same as those in [11], and our runtime for the GPU-based approach is similar to the one in [11]; however, the base approach runs much faster in our experiments. We believe this is because our experiments perform on a 2.67GHz Intel Xeon CPU, which has higher performance than the 2.4GHz AMD Opteron CPU used in [11].

## CHAPTER 5

# ESCAPE ROUTING ON STAGGERED PIN ARRAYS

### 5.1 Introduction

Escape routing is an important problem in package and printed circuit board (PCB) design. Its purpose is to route from specific pins inside a pin array to the boundary of the array. According to recent studies on PCB routing, escape routing can be further classified into three categories [45]: unordered escape, ordered escape, and simultaneous escape. These three types of escape routing problems all have many applications in package and PCB routing. The definition of the unordered escape routing problem is to route from pins inside one pin array to the boundary of the array without considering the pin ordering along the boundary. In this chapter, we focus on the unordered escape routing problem.

The pin array discussed in the unordered escape routing problem usually refers to the traditional square pin array. The traditional square pin array is constructed by placing pins in a pin grid (see Figure 5.1). However, since the constantly evolving technology continues to push the complexity of package and PCB design to a higher level, these complicated designs lead to very dense designs with high pin counts. Thus, staggered pin arrays are introduced to enable such designs with high pin density. A staggered pin array is constructed by shifting the specific columns of the traditional square pin array by a constant distance. The distance in the x-dimension between two adjacent columns of pins is a fixed value, while the distance in y-dimension between two adjacent rows of pins is another fixed value (see the left of Figure 5.2). Moreover,  $\Delta x$  and  $\Delta y$  do not need to correlate with each other. We consider a *tile* as a diamond which is formed by four adjacent pins. The design rules then limit the number of wires between each of the boundaries, the horizontal diagonal, and the vertical diagonal in the tile. We call such



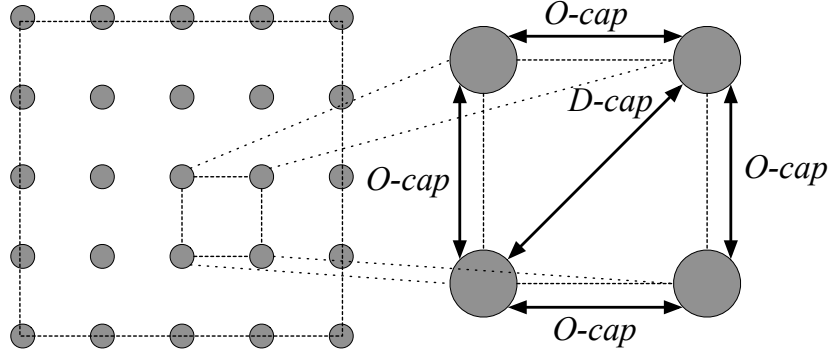


Figure 5.1: An example of a traditional square pin array (left) and the enlarged view of a tile (right).  $O-cap$  and  $D-cap$  are two capacity constraints within a tile.

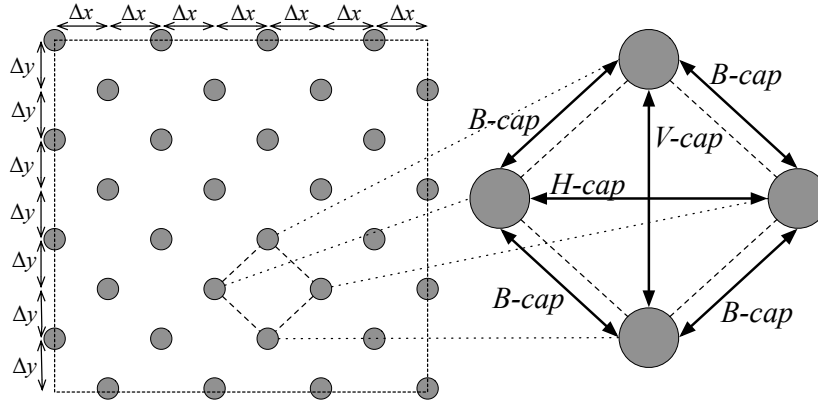


Figure 5.2: An example of a  $4 \times 8$  staggered pin array (left) and the enlarged view of a tile (right).  $\Delta x$  is the distance in the x-dimension between two adjacent columns of pins, and  $\Delta y$  is the distance in the y-dimension between two adjacent rows of pins.

constraints *boundary capacity*, *horizontally diagonal capacity*, and *vertically diagonal capacity*, respectively, or  $B-cap$ ,  $H-cap$ , and  $V-cap$  for short (see the right of Figure 5.2). Since  $\Delta x$  has no specific relationship with  $\Delta y$ , the values of  $B-cap$ ,  $H-cap$ , and  $V-cap$  can be all different.

Currently, the escape routing problem for staggered pin arrays is solved manually in industry. Ho et al. [13] recently presented an escape routing algorithm for staggered pin arrays. Note that the staggered pin array focused on [13] is the hexagonal array, where ( $2 \cdot B-cap = 2 \cdot H-cap = V-cap$ ) or ( $2 \cdot B-cap = H-cap = 2 \cdot V-cap$ ). However, there are other kinds of staggered pin array in current industrial designs. According to the correlation between  $B-cap$ ,  $H-cap$ , and  $V-cap$ , staggered pin arrays can be classified into three

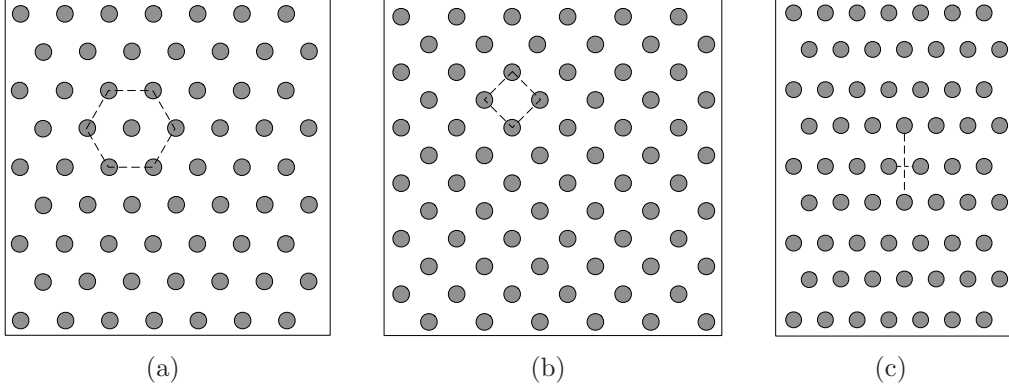


Figure 5.3: Examples of three types of staggered pin array that are available in industrial PCB designs. (a) A hexagonal array where the dashed polygon is a hexagon; (b) a rotated square pin array where the dashed polygon is a rotated square; (c) a staggered pin array that is neither a hexagonal array nor a rotated square pin array. In this example,  $H\text{-cap} = 1$  and  $V\text{-cap} = 3$ .

different types:

- Hexagonal arrays where  $2 \cdot B\text{-cap} = 2 \cdot H\text{-cap} = V\text{-cap}$  or  $2 \cdot B\text{-cap} = H\text{-cap} = 2 \cdot V\text{-cap}$  (see Figure 5.3(a)).
- Rotated square pin arrays where  $H\text{-cap} = V\text{-cap}$  (see Figure 5.3(b)).
- Staggered pin arrays which do not belong to the two types described above. An example is as shown in Figure 5.3(c).

The industrial data sets that we used in our experiment contain these three types of staggered pin arrays.

Given the capacity constraints, the escape routing problem is to find an escape routing satisfying the capacity constraints. The previous work for escape routing on traditional square pin arrays is to find an escape routing satisfying the orthogonal and diagonal capacity constraints defined by two adjacent pins ([46]). A valid escape routing must satisfy the capacity constraints; otherwise, the design rules are violated. Hence, for the escape routing problem on staggered pin arrays, the objective is to find an escape routing that satisfies the  $B\text{-cap}$ ,  $H\text{-cap}$ , and  $V\text{-cap}$  constraints. Although [13] shows that for a hexagonal array even capacity constraints are met, the wire spacing between two adjacent tiles can be violated under some conditions; such conditions never exist for all the industrial boards that we have seen, which means satisfying capacity constraints is enough to guarantee a design rule correct escape routing for all those industrial boards. Note that we have

to consider any two tiles, rather than two adjacent tiles, only when checking the condition. Therefore, in this chapter the escape routing problem we focus on is to find an escape routing satisfying the capacity constraints in the staggered pin arrays.

Network flow is pervasively used to solve the unordered escape routing problem [47, 48, 49, 50, 51, 46, 52] for traditional square pin arrays. The idea is to view each routing path as a unit flow from the pin to the boundary. Since no pin ordering is considered, a flow solution is able to be transformed into some non-crossing routing. Yan and Wong [46] proposed a network flow model to correctly model the capacities inside a square tile for the traditional square pin array. For the traditional square pins arrays, a tile contains two capacity constraints (*O-cap* and *D-cap* in the right of Figure 5.1), so the proposed model can be rotated to solve rotated square pin arrays which contain two capacity constraints as well. However, it cannot model the other two types of staggered pin arrays. Ho et al. [13] used a network based linear programming formulation to solve the escape routing problem on the hexagonal array, and the formulation cannot be extended to solve the other two types of staggered pin arrays. For the hexagonal array, Shi and Cheng [53] also provided some systematic escape routing strategies, but those strategies can only find a limited number of feasible routing solutions. In summary, none of the previous works can completely solve the escape routing on staggered pin arrays. Particularly, none of them can handle the third type of staggered pin arrays (other than hexagonal arrays and rotated square pin arrays), since there is no specific correlation between the capacity constraints, unlike the other two types of staggered pin arrays, which makes it the most difficult one to model. Network flow models proposed in this chapter can solve all three types of staggered pin arrays.

In this chapter, we introduce the problem formulation of escape routing on staggered pin arrays. Then network flow models are proposed to correctly model the capacity constraints of staggered pin arrays. Our models guarantee a legal routing if there exists one routing solution satisfying the *B-cap*, *H-cap*, and *V-cap* capacity constraints. An optimal escape routing algorithm is then built upon the models. Experimental results show that our algorithm has very short runtime.

The rest of this chapter is organized as follows: Section 5.2 formulates the escape routing problem for staggered pin arrays. Section 5.3 presents our

network flow models and escape routing algorithm. Experimental results are given in Section 5.4. Finally, Section 5.5 concludes the chapter.

## 5.2 Problem Formulation

The input to the problem is an  $m \times n$  staggered pin array with  $p$  pins specified as to-be-escaped pins. The definition of an  $m \times n$  staggered pin array is as follows: each row has  $m$  pins and there is a total of  $n$  rows. Figure 5.2 illustrates a  $4 \times 8$  staggered pin array.  $B\text{-cap}$ ,  $H\text{-cap}$ , and  $V\text{-cap}$  are given to specify the boundary capacity, horizontal diagonal capacity, and vertical diagonal capacity in a tile.

The expected output of the problem is an octilinear routing from the to-be-escaped pins to the boundary of the array satisfying the capacity constraints. We also would like the total length of the routing to be minimized.

The  $B\text{-cap}$  constraint limits the number of wires between one side of a tile which implies that at most  $2 \cdot B\text{-cap}$  wires can pass through a tile. Since the  $H\text{-cap}$  and  $V\text{-cap}$  constraints limit the number of wires between two diagonals of a tile, there are at most  $(H\text{-cap} + V\text{-cap})$  wires that can be put inside a tile. If  $B\text{-cap}$  is set such that  $2 \cdot B\text{-cap}$  is larger than  $(H\text{-cap} + V\text{-cap})$ , there are only  $(H\text{-cap} + V\text{-cap})$  wires allowed to pass through a tile. Therefore, it is natural that  $2 \cdot B\text{-cap} \leq H\text{-cap} + V\text{-cap}$  for all our inputs, and we call this the *capacity inequality*. The capacity constraints of the staggered pin arrays we have seen in industrial designs do satisfy the capacity inequality.

The capacity inequality is further analyzed as follows:

- Consider  $H\text{-cap}$  and  $V\text{-cap}$  are both even. Then,

$$\begin{aligned} 2 \cdot B\text{-cap} &\leq 2 \cdot \lfloor H\text{-cap}/2 \rfloor + 2 \cdot \lfloor V\text{-cap}/2 \rfloor \\ \Rightarrow B\text{-cap} &\leq \lfloor H\text{-cap}/2 \rfloor + \lfloor V\text{-cap}/2 \rfloor \end{aligned} \quad (5.1)$$

- Consider  $H\text{-cap}$  is odd and  $V\text{-cap}$  is even, and vice versa. Then,

$$\begin{aligned} 2 \cdot B\text{-cap} &\leq 2 \cdot \lfloor H\text{-cap}/2 \rfloor + 2 \cdot \lfloor V\text{-cap}/2 \rfloor + 1 \\ \Rightarrow B\text{-cap} &\leq \lfloor H\text{-cap}/2 \rfloor + \lfloor V\text{-cap}/2 \rfloor + 1/2 \\ &\leq \lfloor H\text{-cap}/2 \rfloor + \lfloor V\text{-cap}/2 \rfloor \end{aligned} \quad (5.2)$$

- Consider  $H\text{-cap}$  and  $V\text{-cap}$  are both odd. Then

$$\begin{aligned} 2 \cdot B\text{-cap} &\leq 2 \cdot \lfloor H\text{-cap}/2 \rfloor + 2 \cdot \lfloor V\text{-cap}/2 \rfloor + 2 \\ \Rightarrow B\text{-cap} &\leq \lfloor H\text{-cap}/2 \rfloor + \lfloor V\text{-cap}/2 \rfloor + 1 \end{aligned} \quad (5.3)$$

According to (5.1-5.3), for all our inputs, we can conclude that only the inputs where  $2 \cdot B\text{-cap}$  is equal to  $(H\text{-cap} + V\text{-cap})$  and both  $H\text{-cap}$  and  $V\text{-cap}$  are odd satisfy

$$B\text{-cap} = \lfloor H\text{-cap}/2 \rfloor + \lfloor V\text{-cap}/2 \rfloor + 1 \quad (5.4)$$

Otherwise, the inputs satisfy

$$B\text{-cap} \leq \lfloor H\text{-cap}/2 \rfloor + \lfloor V\text{-cap}/2 \rfloor. \quad (5.5)$$

## 5.3 Our Network Flow Models

In this section, we will first present a network flow model for Equation (5.5), and then another one for Equation (5.4) to cover all the staggered pin arrays.

### 5.3.1 Modeling Equation (5.5)

For each tile, we create four nodes, which are  $NW$ -node,  $NE$ -node,  $SE$ -node, and  $SW$ -node, respectively (see Figure 5.4(b)). Bidirectional edges (which are realized by two directed edges: a forward edge and a backward edge) are created between the node pairs  $(NW, NE)$ ,  $(NE, SE)$ ,  $(SE, SW)$ , and  $(SW, NW)$ , and then given capacities,  $\lfloor V\text{-cap}/2 \rfloor$ ,  $\lceil H\text{-cap}/2 \rceil$ ,  $\lceil V\text{-cap}/2 \rceil$ , and  $\lfloor H\text{-cap}/2 \rfloor$ , respectively. For connections between tiles, four directed edges are used to connect the  $NW$ -node of a tile to the  $SE$ -node of its northwest neighboring tile, the  $NE$ -node of a tile to the  $SW$ -node of its northeast neighboring tile, the  $SE$ -node of a tile to the  $NW$ -node of its southeast neighboring tile, and the  $SW$ -node of a tile to the  $NE$ -node of its southwest neighboring tile. We call such edges *inter-tile edges* and give each of them capacity  $B\text{-cap}$ . In order to escape from the to-be-escaped pins, we create a pin node for each pin, and then connect them to the nodes inside a tile. The pin node in the north corner of a tile is connected to the  $NW$ -node

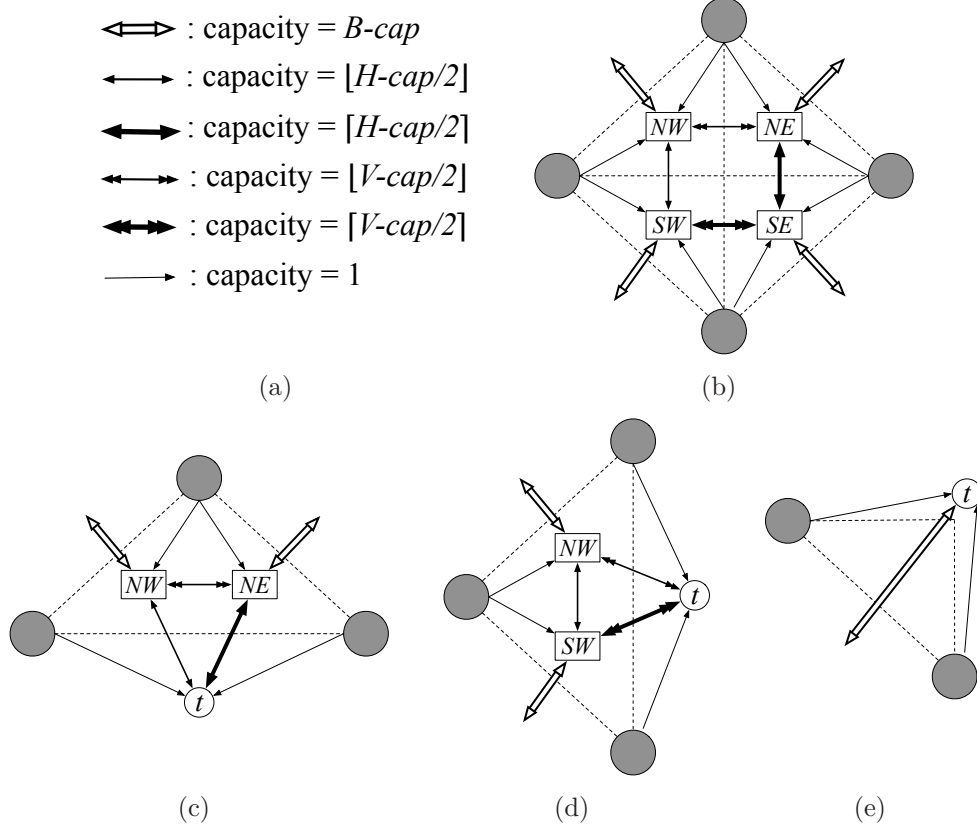


Figure 5.4: Our network model for Equation (5.5). (a) The capacity of each edges; (b) a tile; (c) a 1/2-tile along the bottom boundary of the array; (d) a 1/2-tile along the right boundary of the tile; (e) a 1/4-tile in the northeast corner of the array.  $t$  is the super sink.

and  $NE$ -node, the pin node in the east corner of a tile is connected to the  $NE$ -node and  $SE$ -node, and the pin node in the south corner of a tile is connected to the  $SW$ -node and  $SE$ -node, and finally the pin node in the west corner of a pin is connected to the  $NW$ -node and  $SW$ -node. All these edges have capacity 1.

Although we define a tile as a diamond formed by four adjacent pins, we can see those tiles along the boundary of the array can only contain two or three adjacent pins. We call a boundary tile with three pins a 1/2-tile, and a boundary tile with only two pins a 1/4-tile.

For the 1/2-tiles along the bottom boundary, each has two nodes, which are the  $NW$ -node and  $NE$ -node, respectively (see Figure 5.4(c)). Bidirectional edges with capacity  $\lfloor V\text{-cap}/2 \rfloor$  are created between the  $NW$ -node and  $NE$ -node, and the  $NW$ -node and  $NE$ -node both are connected to the outside of

the array by bidirectional edges with capacity  $\lfloor H\text{-cap}/2 \rfloor$  and  $\lceil H\text{-cap}/2 \rceil$ , respectively. A  $1/2$ -tile along the right boundary is as shown in Figure 5.4(d). Then the construction of the  $1/2$ -tiles along the top (left) boundary is symmetrical to that of the  $1/2$ -tiles along the bottom (right) boundary. Finally, the edges connecting a  $1/2$ -tile to the adjacent tiles and connecting the pin nodes to other nodes are the same as those for a complete tile. Note that those pins lying on the boundary are directly connected to the outside of the array.

The  $1/4$ -tiles can only be sitting in the three corners of the array, which are the northeast, southeast, and southwestern corners, respectively. For a  $1/4$ -tile, there is an inter-tile edge with capacity  $B\text{-cap}$ . According to Equation (5.5),  $B\text{-cap}$  limits the number of wires that can pass through the  $1/4$ -tile. Using the inter-tile edges to connect to the outside of the array is sufficient for the capacity constraints. The model of a  $1/4$ -tile in the northeast corner of the array is shown in Figure 5.4(e).

A super source  $s$  and a super sink  $t$  are also created in the network. We connect  $s$  to the pin nodes of all to-be-escaped pins by edges with capacity 1. Finally, all edges from the boundary tiles to the outside of the array are connected to the sink  $t$ .

Let us call an escape routing *legal* if it satisfies all the  $B\text{-cap}$ ,  $H\text{-cap}$ , and  $V\text{-cap}$  constraints within all tiles. A flow of the network is called *legal* if the flow on every edge does not exceed the edge capacity. Theorem 5.1 guarantees the correctness of our network flow model. The proof of the theorem will be provided in the following section.

**Theorem 5.1.** *Given a staggered pin array with  $k$  to-be-escaped pins that satisfies Equation (5.5), there exists a legal routing of  $k$  pins if and only our network flow model has a legal flow of value  $k$ .*

### 5.3.2 Proof of Theorem 5.1

In order to prove Theorem 5.1, we have to show the following two lemmas, Lemma 5.1 and Lemma 5.2, are both correct.

**Lemma 5.1.** *If a legal routing of  $k$  pins exists, there must exist a legal flow of value  $k$  in our model.*

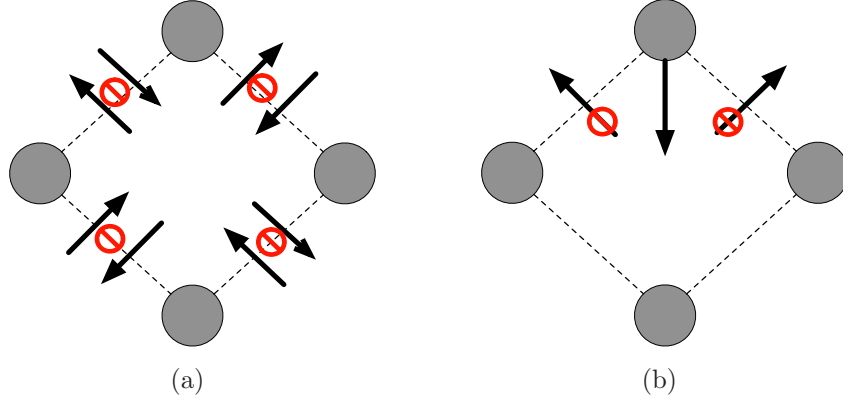


Figure 5.5: The properties of a directed routing  $R$  with the minimum crossings with the tile boundaries. (a) Property 1; (b) Property 2.

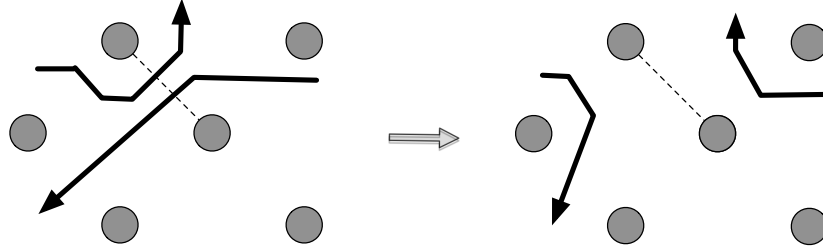


Figure 5.6: Reconnect the wires to solve a routing that violates Property 1. The number of crossings is further reduced while the legality of the routing is still maintained.

*Proof.* We prove this by construction. Let  $R$  be a legal routing of  $k$  pins such that the wires of the routing have the minimum crossings with the tile boundaries. By assigning a direction from the pin to the outside of the array,  $R$  has the following two properties: (1) a tile boundary cannot have the wires with opposite directions passing it (see Figure 5.5(a)); (2) if a pin is routed into one tile, no wires can exit the tile from the two neighboring boundaries of the pin (see Figure 5.5(b)). These two properties can be proved by contradiction.

Suppose  $R$  has two wires with opposite directions crossing the same tile boundary (as shown in the left of Figure 5.6). We can simply reconnect the two wires without affecting the legality of the routing (see the right of Figure 5.6), and it further reduces the number of crossings with the tile boundaries, which contradicts that  $R$  is the routing with the minimum number of crossings with the tile boundaries. Therefore, Property 1 is always true.

Similarly, we can suppose  $R$  has a wire violating Property 2. Figure 5.7



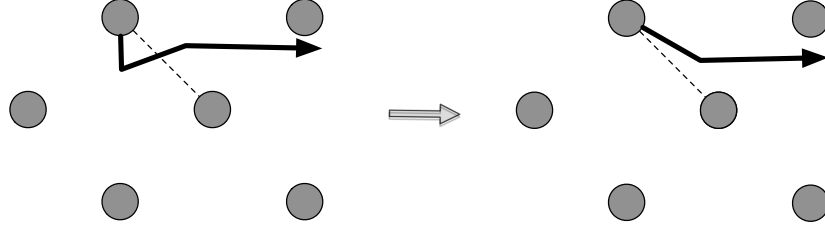


Figure 5.7: Shift a wire to its neighboring tile to solve a routing that violates Property 2. The number of crossings is further reduced without affecting the legality of the routing.

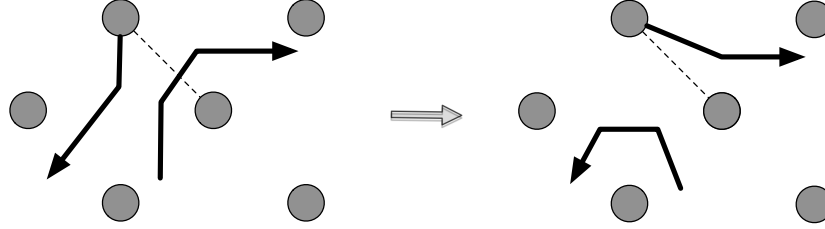


Figure 5.8: Two wires are reconnected to solve a routing that violates Property 2. The number of crossings is further reduced without affecting the legality of the routing.

shows the wire exiting the tile is also the pin that is routed into it, and such wire can be shifted into the neighboring tile to further reduce the number of crossings. Figure 5.8 shows the wire exiting the tile is not the pin that is routed into it, and by reconnecting these two wires, the number of crossings is further reduced. Note that the legality of the routing will not be affected by the above process. Therefore, Property 2 always holds.

The number of wires that cross each tile boundary is equal to the flow that crosses the tile boundary. We call the flow that enters a tile incoming flow,  $IF$ , and the flow that exits a tile outgoing flow,  $OF$ . Because of the continuity of the routing, the incoming flow is equal the outgoing flow for each tile. Therefore, we can apply the flow algorithm to each tile to obtain a flow solution on the inside-tile network. So, next we will show we can always obtain a legal flow solution for any incoming flow and outgoing flow configurations.

For the ease of presentation, let us denote the *NW*-node, *NE*-node, *SW*-node, and *SE*-node as the peripheral nodes. All the possible of configurations of the incoming/outgoing flow are as follows:

- (1) The flow enters (exits) only one peripheral node and exits (enters) three

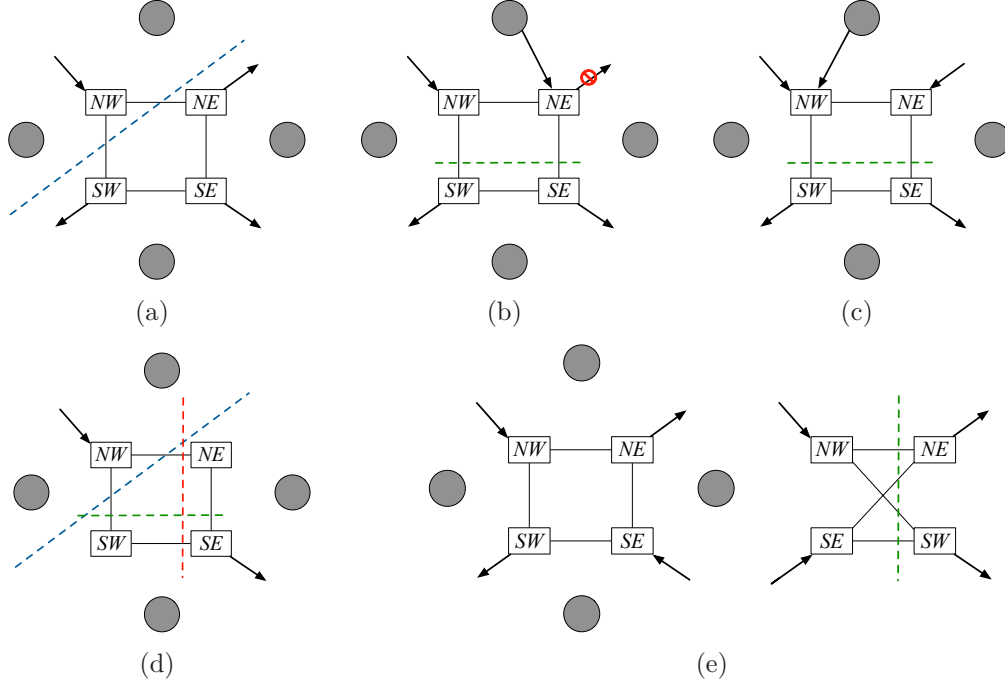


Figure 5.9: (a)-(e) Analysis of the possible flow configurations on the inside-tile network for Equation (5.5). The dotted lines represent a min-cut in the inside-tile network.

peripheral nodes (see Figure 5.9(a)). So  $IF \leq B\text{-cap}$ , and the minimum cut,  $MC$ , is equal to  $\lfloor H\text{-cap}/2 \rfloor + \lfloor V\text{-cap}/2 \rfloor$ . With Equation (5.5), we have  $IF \leq B\text{-cap} \leq MC$ , and thus,  $IF \leq MC$ .

(2) The flow enters only one peripheral node and there is another flow from the pin at the north corner. We assign the flow from the pin to a node that has no flow (see Figure 5.9(b)). Due to Property 2, there is no flow coming out at from the  $NE$ -node. So the wires must cross the horizontal cut to exit the tile. Thus,  $IF$  cannot exceed the  $H\text{-cap}$ , otherwise the routing is illegal. Thus, we have  $IF \leq H\text{-cap} = MC$ .

(3) The flow enters two adjacent peripheral nodes and exits two adjacent peripheral nodes (see Figure 5.9(c)). The wires must cross the horizontal/vertical cut of the tile depending on the locations of the two adjacent nodes. So,  $IF \leq H\text{-cap} = MC$ .

(4) The flow enters only one peripheral node and exits one peripheral node (see Figure 5.9(d)). Since the routing is legal,  $IF \leq \min(B\text{-cap}, H\text{-cap}, V\text{-cap})$ . And,  $MC \leq \min(\lfloor H\text{-cap}/2 \rfloor + \lfloor V\text{-cap}/2 \rfloor, H\text{-cap}, V\text{-cap})$ . Thus, we have  $IF \leq MC$ .

(5) The flow exits two nonadjacent peripheral nodes (see the left of Figure 5.9(e)). According to Property 2, there must exist no flow from a pin to the peripheral nodes. So, we know  $IF \leq B\text{-cap} + B\text{-cap} = 2B\text{-cap}$ . The minimum cut of this inside-tile network is shown as the right of Figure 5.9(e), and  $MC = \lfloor H\text{-cap}/2 \rfloor + \lceil H\text{-cap}/2 \rceil + \lfloor V\text{-cap}/2 \rfloor + \lceil V\text{-cap}/2 \rceil = H\text{-cap} + V\text{-cap}$ . So, with the capacity inequality, we have  $IF \leq 2B\text{-cap} \leq (H\text{-cap} + V\text{-cap}) = MC$ .

The above analysis can be easily extended into the boundary tiles. So, we know  $IF \leq MC$  for all tiles. According to max-flow min-cut theorem [54], if  $IF \leq MC$ , then there must exist a legal flow solution in the network. There we can always obtain a legal flow on any inter-tile network by the maximum flow algorithm.  $\square$

**Lemma 5.2.** *A legal flow of value  $k$  can be transformed into a legal routing of  $k$  pins.*

*Proof.* A procedure proposed in [55] can transform a flow of value  $k$  into a planar topology of routing with  $k$  pins. Although our network flow model is different from the one in [55], such procedure still can be applied on our flow solution, as long as the flow solution is legal. Since the  $B\text{-cap}$  constraint can be captured by the inter-tile edge,  $H\text{-cap}$  constraint can be captured by the two vertical edges in a tile with capacity  $\lfloor H\text{-cap}/2 \rfloor$  and  $\lceil H\text{-cap}/2 \rceil$ , and the  $V\text{-cap}$  constraint can be captured by the two horizontal edges in a tile with capacity  $\lfloor V\text{-cap}/2 \rfloor$  and  $\lceil V\text{-cap}/2 \rceil$ , our network flow model can correctly capture the capacity constraints within each tile. The routing transformed from a legal flow must be legal as well.  $\square$

### 5.3.3 Modeling Equation (5.4)

The model for Equation (5.4) is mostly identical to the former model. For each tile, we give both vertical edges a capacity of  $\lfloor H\text{-cap}/2 \rfloor$  and both horizontal edges a capacity of  $\lfloor V\text{-cap}/2 \rfloor$  (see Figure 5.10(b)). Each tile contains one more node than in the former model. The node is a center node, called  $C$ -node. The center node has a capacity 1. Node capacity can be realized by splitting the node into two nodes and adding an edge with the same capacity between them. Bidirectional edges are created to connect the  $C$ -node with the  $NW$ -node,  $NE$ -node,  $SE$ -node, and  $SW$ -node. We give such edges infinite capacity.

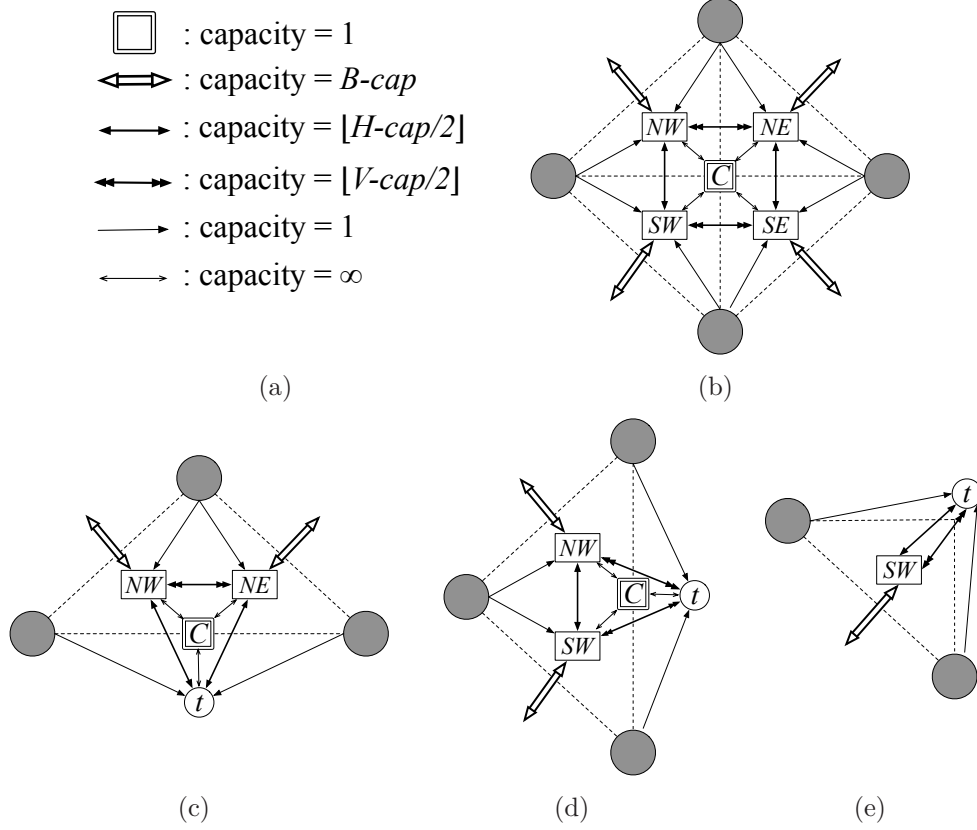


Figure 5.10: Our network model for Equation (5.4). (a) The capacity of each edges; (b) a tile; (c) a 1/2-tile along the bottom boundary of the array; (d) a 1/2-tile along the right boundary of the tile; (e) a 1/4-tile in the northeast corner of the array.  $t$  is the super sink.

For the 1/2-tiles, a center node with a capacity 1 is created as well (see Figure 5.10(c) and 5.10(d)). Bidirectional edges with infinite capacity are also used to connect the center node to other nodes. Edges with capacity  $\lfloor H\text{-cap}/2 \rfloor$  and  $\lfloor V\text{-cap}/2 \rfloor$  are created between the nodes inside them and connect to the outside of the array.

The 1/4-tile has a node inside it, and edges with capacity  $\lfloor H\text{-cap}/2 \rfloor$  and  $\lfloor V\text{-cap}/2 \rfloor$  are used to connect to the outside of the array. Similarly, a super source  $s$  and a super sink  $t$  are created as well, and they are connected with the to-be-escaped pins and the outside edges from the boundary tiles, respectively.

The following theorem guarantees the correctness of the network flow model. The proof will be presented in the next section.

**Theorem 5.2.** *Given a staggered pin array with  $k$  to-be-escaped pins that*

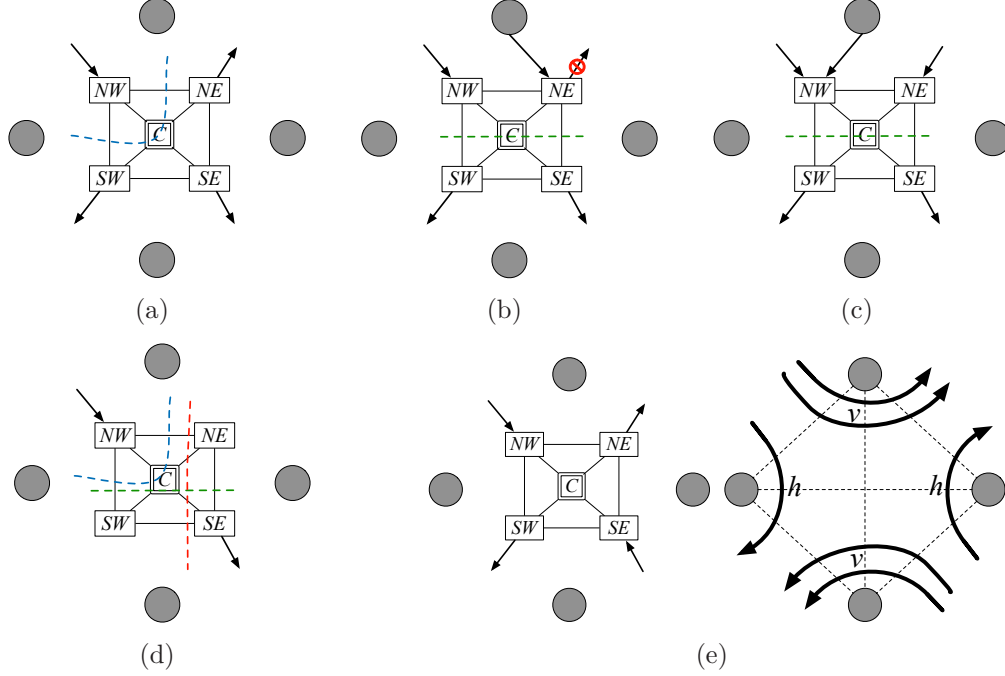


Figure 5.11: (a)-(e) Analysis of the possible flow configurations on the inside-tile network for Equation (5.4). The dotted lines represent a min-cut in the inside-tile network.  $h = \lfloor H\text{-cap}/2 \rfloor$  and  $v = \lfloor V\text{-cap}/2 \rfloor$ .

satisfies Equation (5.4), there exists a legal routing of  $k$  pins if and only if our network flow model has a legal flow of value  $k$ .

### 5.3.4 Proof of Theorem 5.2

We have to show that Lemma 3 and Lemma 4 are both correct.

**Lemma 5.3.** *If a legal routing of  $k$  pins exists, there must exist a legal flow of value  $k$  in our model for Equation (5.4).*

*Proof.* The construction from a routing to the inside-tile networks and the possible configurations of the incoming/outgoing flow for a tile are both identical to those shown in Section 5.3.2. An example of all the possible configurations is given in Figure 5.11. Since the detailed analysis of the configuration (1)-(4) is similar to the one in Section 5.3.2, the analysis is omitted. Here, we only focus on the configuration (5):

(5) The flow exits two nonadjacent peripheral nodes (see the left of Figure 5.11(e)). Suppose a tile has  $IF = 2 \cdot \lfloor H\text{-cap}/2 \rfloor + 2 \cdot \lfloor V\text{-cap}/2 \rfloor$ , which is shown

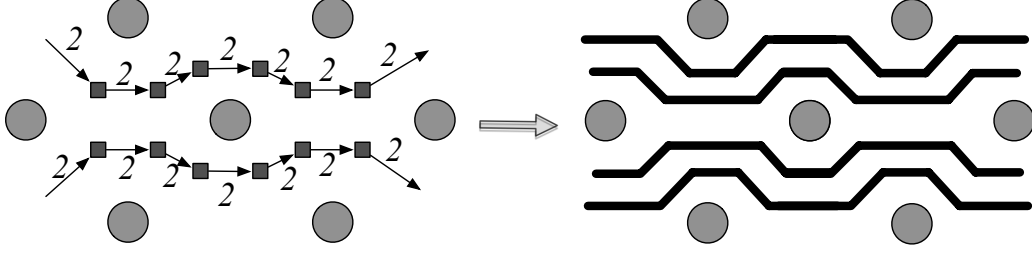


Figure 5.12: A flow solution (left) is transformed into an octilinear routing(right). The zigzag wires obtained from the octilinear routing style effectively utilize the routing area.

in the right of Figure 5.11(e). Since  $B\text{-cap} = \lfloor H\text{-cap}/2 \rfloor + \lfloor V\text{-cap}/2 \rfloor + 1$ , only one more flow can enter/exit the tile, otherwise the routing is illegal. So,  $IF$  is bounded by  $2 \cdot \lfloor H\text{-cap}/2 \rfloor + 2 \cdot \lfloor V\text{-cap}/2 \rfloor + 1$ . Thus,  $IF \leq MC$ .  $\square$

**Lemma 5.4.** *A legal flow of value  $k$  in our model for Equation (5.4) can be transformed into a legal routing of  $k$  pins.*

*Proof.* As described in the proof for Lemma 2, an existing approach can transform a flow of value  $k$  into a planar routing with  $k$  pins. In our network flow model, the horizontally diagonal capacity can be ensured by the two vertical edges and the center node with a total capacity of  $2 \cdot \lfloor H\text{-cap}/2 \rfloor + 1 = H\text{-cap}$ , while the vertically diagonal capacity can be ensured by the two horizontal edges and the center node with a total capacity of  $2 \cdot \lfloor V\text{-cap}/2 \rfloor + 1 = V\text{-cap}$ .  $\square$

### 5.3.5 Escape Routing Algorithm

Based on the input of the problem, we can decide a network flow model and then apply the maximum flow algorithm on it. The max-flow solution can be transformed into a topological routing by splitting the nodes and edges of the flow solution [46]. Then the topological routing can be converted into an octilinear routing by existing algorithms [56, 57]. An example that a flow of total value 4 turns into an octilinear routing is shown in Figure 5.12. We can see that the zigzag wires in the octilinear routing effectively utilize the routing area.

We can also assign cost 1 to the inter-tile edges, and zero cost to all other

Table 5.1: Experimental results

	array $m \times n$	escape pin no.	Eqn(5.4) or Eqn(5.5)	capacities			our results	
				$B$	$H$	$V$	routability	runtime
<i>case_1</i>	$35 \times 13$	157	(5.5)	2	3	4	100%	0.12s
<i>case_2</i>	$25 \times 17$	163	(5.5)	2	4	4	100%	0.13s
<i>case_3</i>	$35 \times 13$	160	(5.5)	2	4	4	100%	0.12s
<i>case_4</i>	$55 \times 17$	374	(5.5)	2	4	4	100%	0.29s
<i>case_5</i>	$17 \times 34$	86	(5.5)	2	2	4	100%	0.16s
<i>case_6</i>	$17 \times 34$	113	(5.5)	2	2	4	100%	0.16s
<i>case_7</i>	$8 \times 180$	704	(5.5)	2	4	4	100%	0.48s
<i>case_8</i>	$35 \times 13$	140	(5.4)	2	3	1	100%	0.12s
<i>case_9</i>	$11 \times 27$	42	(5.5)	2	2	3	100%	0.08s
<i>case_10</i>	$17 \times 34$	197	(5.5)	2	4	4	100%	0.17s
<i>case_11</i>	$25 \times 34$	172	(5.5)	2	2	4	100%	0.26s

edges. We can compute a min-cost max-flow solution to minimize the number of tiles each wire traverses and thus the total wire length can be minimized.

## 5.4 Experimental Results

We implement our network flow based escape routing algorithm in C++ and test it on several industrial data sets. The min-cost flow solution of our model is obtained by the min-cost flow solver CS2[58]. All experiments are performed on a workstation with two 2.4GHz Intel Xeon CPU.

We test our router on eleven data sets and the result is shown in Table 5.1. Among the eleven data sets, *case\_1* to *case\_7* are actual industrial data and *case\_8* to *case\_11* are derived from industrial data with some modification. The first seven columns of the table give the information on the data including the name, the pin array size, the number of to-be-escaped pins, the property of the input (satisfying Equation (5.4) or Equation (5.5)), and the capacity rules ( $B$ -cap,  $H$ -cap, and  $V$ -cap). The last two columns show the routability of our results as well as the runtime of our router. The routability is defined as the value of the number of routed pins/the number of to-be-escaped pins.

It can be seen that all data sets satisfy the capacity inequality. Among the eleven data sets, ten satisfy Equation (5.5) while one satisfies Equation (5.4). Our router can successfully route all data sets in very short time. Our routing solution of *case\_10* is shown in Figure 5.13. A dashed square is drawn

on top of the routing result to show our router can handle the dense designs. It can be seen that almost all the  $B\text{-cap}$ ,  $H\text{-cap}$  and  $V\text{-cap}$  along the square is used by our routing, which indicates that the routing is very dense.

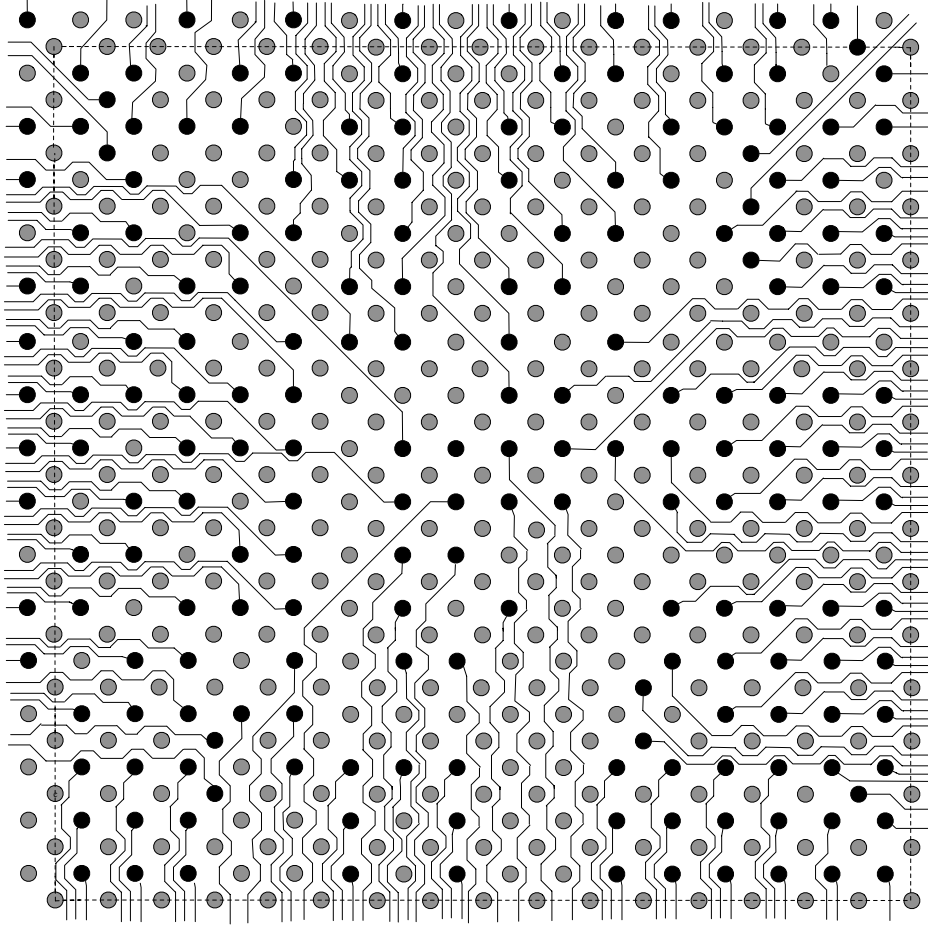


Figure 5.13: Routing solution of *case\_10*. The dashed square is drawn on top of the result to show that routing uses up almost all routing resources.

## 5.5 Conclusion

In this chapter, we studied the escape routing problem for staggered pin arrays. Based on the geometry of the staggered pin array, a tile can be defined as a diamond formed by four adjacent pins and hence the corresponding capacity constraints are generated for a tile. Then we formulated the escape routing problem for staggered pin arrays. We proposed two network flow models to model the capacity constraints of staggered pin arrays, and we



proved the correctness of our models. These network flow models led to an optimal algorithm. From the experimental results, it is shown that our escape routing algorithm can successfully route the industrial data in a very short time.

# CHAPTER 6

## AN ILP-BASED AUTOMATIC BUS PLANNER FOR DENSE PCBs

### 6.1 Introduction

Modern PCBs have to be routed manually since no EDA tools can successfully route these complex boards. Nowadays, a dense PCB contains thousands of pins [12]. On the other hand, the size of a package is kept minimum. This makes the footprint of such a package on PCB a very dense pin grid. Such a large net count and high pin density make manual design of PCBs an extremely time consuming and error-prone task. An auto-router for PCBs would improve design productivity tremendously since each board takes about 2 months to route manually. Therefore, design automation of PCB routing becomes a necessity.

It is observed from industrial manual solutions that the nets are grouped as buses, and the nets within the same bus are expected to be routed together [14, 15, 16, 17, 18]. A typical high-end PCB contains a number of components and a number of bus structures. The bus planning, which is to simultaneously solve the bus decomposition, escape routing, layer assignment and global bus routing, is an important yet difficult step. Figure 6.1 shows an example bus planning solution on one layer, which consists of three components and eight buses. During the bus planning, we need to decompose large buses into smaller buses (if necessary), then topologically route the buses on each layer in a planar fashion (including the escape routing within the components and the global routing outside the components), and use a given number of layers to accommodate all the buses. In the meantime, we also need to consider the routing congestion between the components as well as the min-max length constraints of the nets.

The bus planning problem of PCBs has been studied by a number of previous works, which fall into three categories:

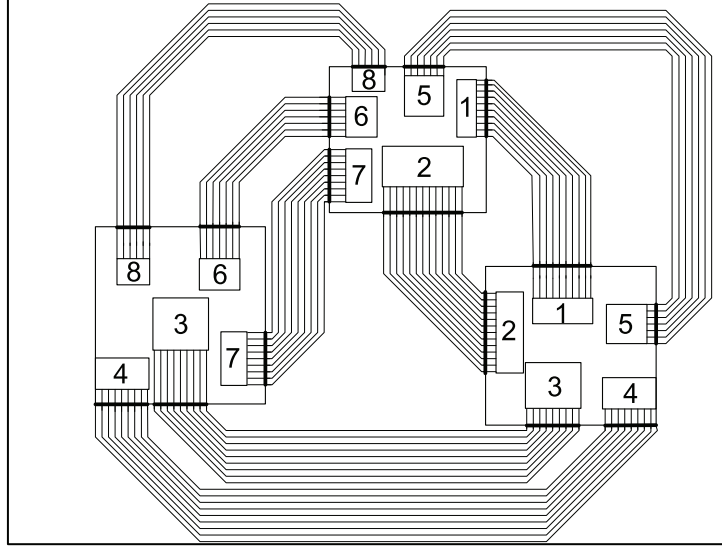


Figure 6.1: One layer of an example bus planning solution.

- **Single component bus planning** only considers the buses within a single component. Kong *et al.* proposed in [16] an optimal algorithm for finding a maximum nonconflicting subset of buses within one component. Ma *et al.* then presented in [19] an approximation algorithm to determine the escape directions of the buses within one component such that the resultant maximum density over the component is minimized (the maximum density is a good indicator of the number of layers needed).
- **Two components bus planning** considers a set of buses connecting between two components. Kong *et al.* proposed in [14] an optimal algorithm to compute a maximum subset of the buses that can be assigned to a single layer without conflicts. Yan *et al.* then optimally solved the layer assignment for multiple layers in [18]. These two works are based on the assumption that all the buses are escaped along the same boundary of a component. Recently in [17], Ma *et al.* presented a branch-and-bound based approach to solve the general layer assignment problem of the buses connecting two components, where the buses can be escaped to any boundary of the components.
- **Board level bus planning** considers all the buses connecting among multiple components on the whole board. Kong *et al.* presented a board level bus planner in [15]. In their approach, the escape directions of the

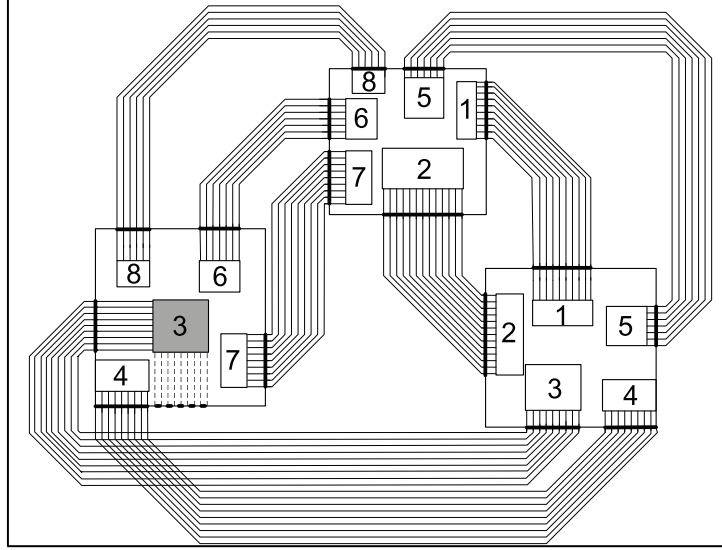


Figure 6.2: If the escape directions of the buses in Figure 6.1 are predetermined, and bus 3 in the lower left component is decided to escape to the left boundary instead of the bottom boundary, the set of buses can no longer be routed on one layer.

buses to the component boundaries are pre-determined. The buses are then topologically routed by performing a negotiated-congestion based router on a dynamic routing graph, after which simulated annealing is employed to do the layer assignment of the buses.

The works considering only one or two components provided elegant algorithms and are theoretically solid. They can find their applications, especially when the one or two components are dominating ones (much denser than others) on the PCB. However, in the more general case when all the components are almost equally important, the significance of these works greatly decreases, as they lack a global view of buses on the entire board.

The board level bus planning is a difficult yet important problem, where we need to simultaneously solve the bus decomposition, escape routing, layer assignment and global bus routing. Note that bus decomposition is also an important step, which is to properly decompose a bus into two or more smaller buses so that each of them can be routed on one layer. In practice, a good bus decomposition solution is essential to the success of the escape routing and layer assignment step. The board level bus planning problem was partially addressed in [15] where they only focused on the layer assignment and global bus routing, assuming bus decomposition and escape routing are

given. This simplified the entire bus planning problem at the expense of narrowing down the solution space. Fixing too many things at an early stage can possibly exclude the optimal solution(s) from the searching space. Also, due to the lack of a more global view at an early stage, it is difficult to decide a bus decomposition and an escape routing solution within the components that is going to be favored by the later stages during the bus planning. Let us take the set of buses in Figure 6.1 as an example. If the escape directions of these buses are predetermined, and bus 3 in the lower left component is decided to escape to the left boundary instead of the bottom boundary, the set of buses can no longer be routed on one layer, as shown in Figure 6.2. Intuitively, a seamless way of simultaneously solving everything together is preferable. To the best of our knowledge, there is no published work on a complete bus planner (i.e., one which does decomposition, escape, layer assignment and global routing). Although there are industrial internal bus planning tools, none can satisfactorily solve the bus planning problem to date. We are aware of a state-of-the-art industrial internal bus planning tool. It is a multi-stage system where the bus decomposition and escape routing are based on minimizing maximum bus intersection density (similar to [16] and [19]), and the layer assignment and global bus routing are based on the algorithm in [15]. For a complex industrial PCB with over 7000 nets which was previously routed manually in 12 layers, the automatic industrial planner was only able to route 84.7% nets but no previous auto bus planner could route more nets. We will present a new bus planner in this chapter which significantly outperforms the multi-stage industrial bus planner.

In this chapter, we present an ILP-based board level bus planner, which considers bus decomposition, escape routing and global routing simultaneously during layer assignment stage. Our planner has the following features:

- An escape router is first applied for each bus within the component in which it resides. A collection of candidate escape solutions and decomposition solutions (if necessary) for each bus is gathered. The conflict information between different buses within the components, namely, the internal conflict, is collected.
- A global router is proposed and a modified dynamic routing graph is adapted as an underlying graph. Each bus is tentatively routed using all of its candidate escape solutions generated, after which the conflict

information between different buses outside the components, namely, the external conflict, is collected.

- From the obtained candidate routes and bus decompositions, an ILP-based approach is proposed to resolve both the internal conflict and the external conflict simultaneously, after which a conflict-free layer assignment is obtained.
- To effectively utilize routing resources, empty components are determined on each layer, and then ILP is employed to resolve the congestion, after which a conflict-free and congestion-free layer assignment is obtained.

We test our bus planner on a dense industrial board, and compare with the state-of-the-art industrial internal bus planner. The results show that our bus planner outperforms the industrial one in terms of both solution quality and runtime. Our bus planner reports a 97.4% completion rate of all the nets within 2.5 hours, in contrast to an 84.7% completion rate within 5 hours by the industrial bus planner. Compared with manually routing all the nets from scratch, our proposed bus planner is able to save a significant amount of manual effort by automatically routing most of the nets. The remaining nets that are left unrouted can be routed manually or by vias.

The rest of this chapter is organized as follows: Section 6.2 introduces the bus planning problem. Section 6.3 presents our algorithm flow to solve this problem, and the details of our algorithm are described in Sections 6.4-6.7. Section 6.8 reports the experimental results on some industrial data, and Section 6.9 concludes this chapter.

## 6.2 Problem Formulation

A PCB contains a number of components, each of which is a rectangular region formed by a pin grid array. Each bus corresponds to two pin clusters in two components, and these two pin clusters need to be connected by a bundle of wires. A bus is also allowed to be decomposed into two or more smaller buses which can be routed on different layers. The route of a bus is composed of the escape route part and the global route part. The escape route refers to the part of routing from the pin cluster to its component

boundary, and global route refers to the part of routing connecting the two components. If the escape routes of two buses conflict, we call it *internal conflict*; if the global routes of two buses conflict, we call it *external conflict*. The buses conflicting with each other have to be assigned to different layers.

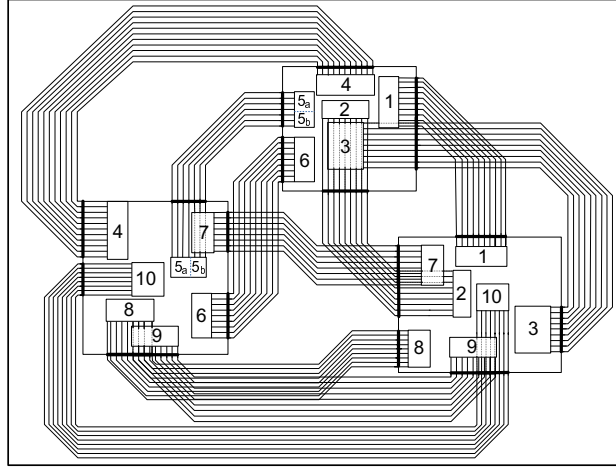
The input of our bus planning problem is a number of components and a set of buses as well as the number of available routing layers. The objective is to decide the bus escape routes and decomposition (if necessary) within the components, the global routes outside the components, as well as the layer assignment of the topological routes of buses, where the buses on each layer are routed in planar fashion considering crossings, routing congestions and min-max length bounds.

## 6.3 Methodology

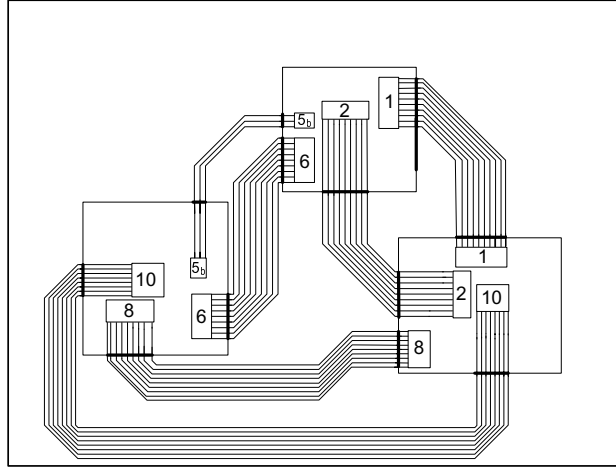
Our bus planning consists of four stages:

1. **Candidate routes generation:** A collection of candidate escape routes (together with bus decompositions, if necessary) and a collection of candidate global routes are generated. At the end of this stage, the conflict information between the candidate routes can be obtained as well.
2. **Layer assignment:** An ILP is formulated to compute a conflict-free layer assignment of the buses. The ILP considers the candidate escape routes, the bus decompositions, and the candidate global routes simultaneously. Figure 6.3 gives an example containing 10 buses. Figure 6.3(a) shows the candidate routes and bus decompositions (bus 5 is decomposed into bus  $5_a$  and  $5_b$ ) chosen for each bus by the ILP. The ILP assigns the buses into two layers with no conflicts, as shown in Figure 6.3(b) and Figure 6.3(c).
3. **Resolving congestions:** Generate a congestion free bus planning upon the layer assignment obtained from the previous stage.
4. **Postprocessing:** The rerouting of the unrouted nets is re-attempted to further improve the results.

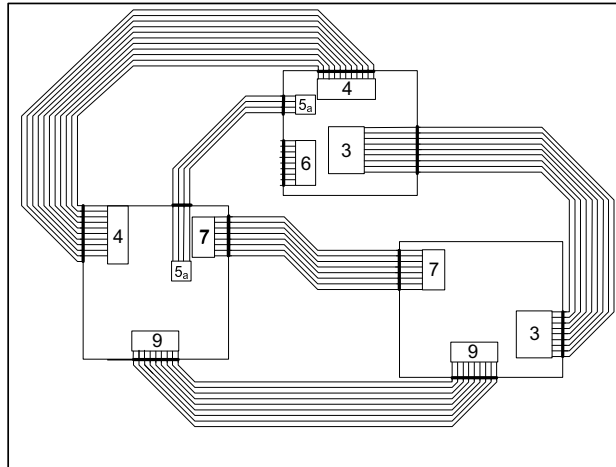
The details of our approach are explained in the following.



(a)



(b)



(c)

Figure 6.3: An example bus planning problem, where bus 5 is decomposed into  $5_a$  and  $5_b$ , and buses in (a) are assigned into two layers as shown in (b) and (c) with no conflicts.



Table 6.1: An example bus decomposition process for a bus  $b$ .

Iterations	Input #	Un-escaped #	Escaped #	Bus $b$
Iteration1	50	35	15( $b_1$ )	$\{b_1\}$
Iteration2	35	15	20( $b_2$ )	$\{b_1, b_2\}$
Iteration3	15	0	15( $b_3$ )	$\{b_1, b_2, b_3\}$

## 6.4 Candidate Routes Generation

In this section, we first generate a collection of escape routes for all buses, and, if necessary, we will decompose some buses into a set of smaller buses, and then a global router is applied to generate the global routes. With a number of options for escape routes and global routes, a bus may have more than one candidate route. The conflict information between these routes will be generated as well.

### 6.4.1 Escape Routing and Bus Decomposition

A pin cluster can escape to any boundary of the component in which it resides. The boundary that a pin cluster escapes to is called the *escaping boundary*. Since a bus corresponds two pin clusters, a bus has  $4 \times 4 = 16$  combinations of escaping boundaries for the two pin clusters. In addition, a rectilinear region is designated as the routing resource for a pin cluster to escape to the escaping boundary. A rectilinear region is a projection rectangle or an L-shaped rectangle in our implementation. An escape router is then called to compute an escape route from the pin clusters to the selected escaping boundaries within the rectilinear regions. An escape route of a bus is found when all the pins of the two pin clusters can be successfully escaped to their escaping boundaries. For each bus, multiple escape routes could be generated by using different escaping boundaries and rectilinear regions.

However, not all buses can be successfully escaped to the selected escaping boundaries. If a bus fails to be escaped, then the bus is unable to be escaped on one layer and thus has to be further decomposed into two or more smaller buses, each of which can be escaped on one layer. We decompose a bus by iteratively applying the escape router until all the pins are escaped. Those pins that are successfully escaped in one iteration are defined as a smaller bus, and the escape route found by the router forms its escape route. Then, the

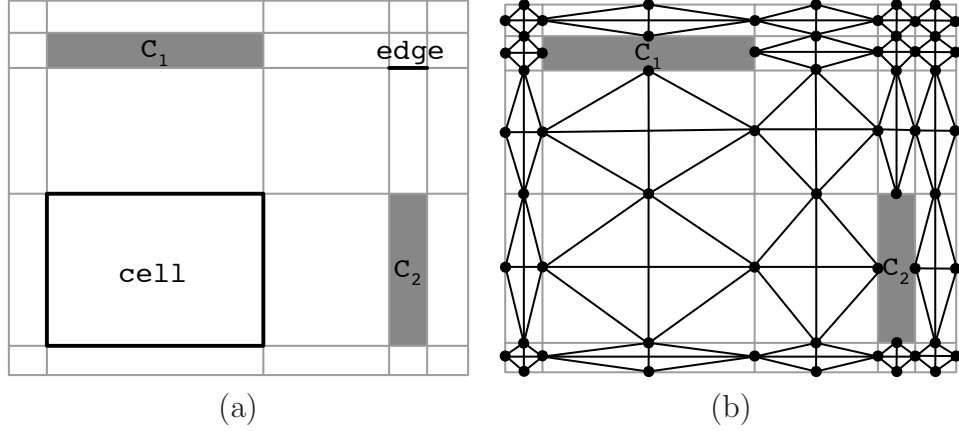


Figure 6.4: (a) The Hanan grid and (b) the routing graph constructed for an example PCB with two components  $C_1$  and  $C_2$ .

remaining un-escaped pins are fed to the escape router as input for the next iteration. This process is performed iteratively until all the pins are escaped. An example of the bus decomposition process is illustrated in Table 6.1. The bus  $b$  is decomposed into a set of smaller buses  $\{b_1, b_2, b_3\}$ , any two of which cannot be assigned onto the same layer. Note, a bus could be decomposed into different sets of smaller buses by using different escaping boundaries and rectilinear regions.

For each bus, all of its candidate escape routes can be obtained by enumerating all the combinations of the escaping boundaries and rectilinear regions. We can then easily gather the conflict information between the escape routes of different buses.

### 6.4.2 Global Routing

For each candidate escape route, we generate a global route. An escape route can be viewed as two intervals on the two escaping boundaries and they are going to be connected by a global route. Hence, the input to our global router is all pairs of intervals that need to be connected, and the output is a collection of global routes for each of the buses.

The dynamic routing graph proposed in [15] is modified and adapted as the underlying routing graph of our global router. The dynamic routing graph is constructed based on a Hanan grid, which is obtained by extending the boundaries of the components until the boundaries of other components

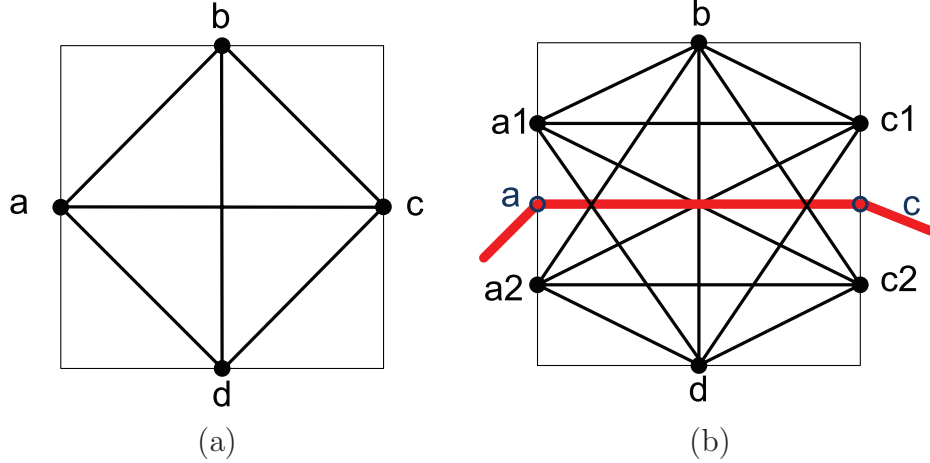


Figure 6.5: The dynamic routing graphs in a Hanan grid cell. (a) and (b) give the routing graphs before and after a bus is routed. After the edge  $(a, c)$  is occupied by the bus, four new vertices ( $a1$ ,  $c1$ ,  $a2$  and  $c2$ ) and their corresponding new edges are added. The weight of the new edges  $(a1, c1)$ ,  $(a1, c2)$ ,  $(a2, c1)$ , and  $(a2, c2)$  is equal to the weight of the old edge  $(a, c)$ .

or the boundaries of the board are encountered. Figure 6.4(a) shows the Hanan grid of an example PCB with two components. The dynamic routing graph's vertices are the middle points of all Hanan grid edges and its edges are connections of vertices in the same cell. Figure 6.4(b) shows the dynamic routing graph constructed from the Hanan grid in Figure 6.4(a). When an edge is occupied by a bus, the cell is split into two parts and new vertices and new edges are created. Figure 6.5(a) shows the initial routing graph in a Hanan grid cell, and Figure 6.5(b) shows the new routing graph after a bus passes through this cell, where two vertices ( $a$  and  $c$ ) and the edges incident to them are eliminated, while four new vertices ( $a1$ ,  $a2$ ,  $c1$  and  $c2$ ) together with the new edges representing the new connections are added. The weight of a dynamic edge is set to be the Manhattan distance between the middle points of the two Hanan grid edges that the edge connects. For example, the weight of the edge  $(a1, c2)$  is set to be the Manhattan distance between  $a$  and  $c$ . Then the shortest path algorithm is performed sequentially to connect all pairs of intervals. Although the dynamic routing graph is updated after a bus is routed, the way we set the edges' weights makes the weight of the new edges always equal to the weight of the old ones (see Figure 6.5). By doing so, the total weight of the shortest path of a bus remains the same regardless of its routing order in the router. During routing, the max length bound

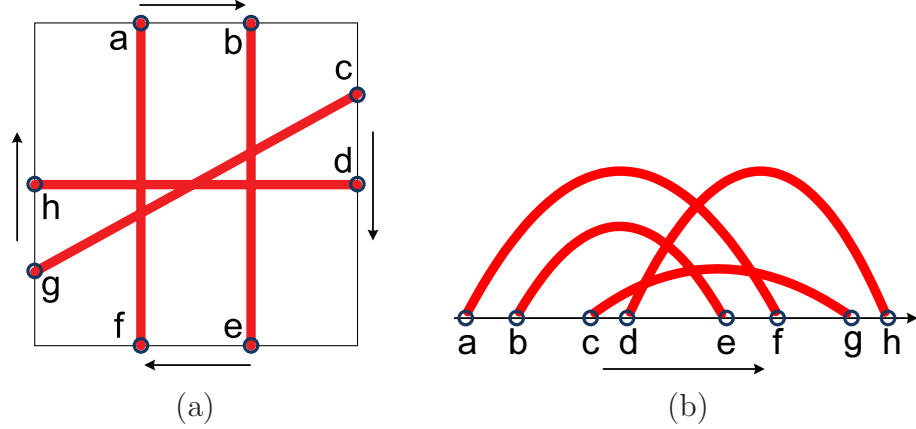


Figure 6.6: By traversing the boundaries of a Hanan grid cell, (a) the routes passing through this cell can be transformed to (b) a set of one-directional intervals. The two segments  $(a, f)$  and  $(c, g)$  in (b) overlap with each other, indicating that the two corresponding routes conflict with each other in (a).

constraint can be easily satisfied (while the min length bound constraint can be satisfied by using techniques like wires extension during postprocessing).

We also need to collect the external conflict information of the global routes. This can be done easily by taking advantage of the structure of our dynamic routing graph. Each Hanan grid cell is occupied by a set of routes, some of which may conflict with each other. By traversing the boundaries of the Hanan grid cell clockwise, the routes passing through this cell can be transformed to a set of one-directional intervals, and then the external conflict information of the routes can be easily computed by checking if the corresponding intervals overlap (see Figure 6.6). In addition, we use the total number of the external conflict as a tie-breaker when searching the shortest path. Choosing the shortest path with the fewest external conflicts as the global route can effectively avoid the unnecessary external conflict.

## 6.5 Layer Assignment

In this section, a layer assignment procedure based on an ILP model is proposed to assign the candidate routes obtained from the previous stage onto the given layers. The internal and external conflict information will be simultaneously considered to generate a layer assignment without any conflict.

### 6.5.1 ILP Formulation

The input of the ILP model is a set  $B$  of buses  $\{b_1, b_2, \dots, b_m\}$ , a set  $C$  of candidate routes  $\{c_1, c_2, \dots, c_n\}$  where  $n_{c_i}$  is the number of the nets of a candidate route  $c_i$ , and  $p$  routing layers. The conflict information and the bus decompositions are provided as well. Our ILP model assigns a set of candidate routes  $\in C$  onto the  $p$  layers, that has a maximum number of nets and no conflict on each layer.

In the ILP model, a set of 0-1 variables  $\{x_{i1}, x_{i2}, \dots, x_{ip}\}$  is introduced for each candidate route  $c_i \in C$ , indicating whether candidate route  $c_i$  is assigned to layer 1, layer 2,  $\dots$ , or layer  $p$ , respectively. The objective of the model is to select a maximum number of nets onto the  $p$  layers, so the objective is formulated as follows:

$$\text{Maximize} \quad \sum_{i:c_i \in C} \sum_{l=1}^p (x_{il} \times n_{c_i})$$

Then we can add the following constraints:

- **Candidate Constraints:** Let a set of candidate routes  $C_b \in C$  be the candidate routes of a bus  $b$ . Only a candidate route in  $C_b$  can be chosen for bus  $b$ , so we have the following set of constraints:

$$\sum_{i:c_i \in C_b} \sum_{l=1}^p x_{il} \leq 1, \quad \forall b \in B$$

- **Conflict Constraints:** Let  $c_i \perp c_j$  denote that  $c_i$  and  $c_j$  conflict. The conflicting candidate routes cannot be assigned to the same layer, so we have the following set of constraints:

$$x_{il} + x_{jl} \leq 1, \quad \forall l = 1, 2, \dots, p, \quad \text{if } c_i \perp c_j, \forall c_i, c_j \in C$$

- **Bus Decomposition Constraints:** Let  $D_b$  be a set of bus decompositions  $\{d_1, d_2, \dots\} \in B$  for a bus  $b$ , in which each  $d_j$  is a set of buses that  $b$  is decomposed into. Only a bus decomposition in  $D_b$  is allowed to be chosen. Suppose that  $b_1$  contains two bus decompositions,  $d_1$  and  $d_2$ , where  $d_1 = \{b_2, b_3, b_4\}$  and  $d_2 = \{b_5, b_6\}$ , respectively. We can see that it is illegal to assign  $b_2$  to one layer while  $b_5$  is assigned to

another layer, since  $b_2$  and  $b_5$  belong to the same bus but different bus decompositions. Only one bus decomposition among the bus decompositions can be selected by the ILP, so the candidate routes among different bus decompositions cannot be chosen simultaneously. Given a bus  $b \in B$ , we can have the following set of constraints for any pair of bus decompositions  $(d^*, d^\#) \in D_b$ :

$$\sum_{j:c_j \in C_{b^*}} \sum_{l=1}^p x_{jl} + \sum_{k:c_k \in C_{b^\#}} \sum_{l=1}^p x_{kl} \leq 1, \quad \forall b^* \in d^*, b^\# \in d^\#$$

So the bus decomposition constraints can be added by generating the above set of constraints for every bus in  $B$ .

By solving the above ILP model, we can obtain a conflict-free layer assignment that contains the maximum number of nets.

### 6.5.2 Algorithm

The layer assignment algorithm in our planner is a two-pass ILP process. Only the candidate routes constructed from the projection rectangles will be considered in the first run of the ILP. There are two reasons for considering projection rectangles first: (1) it is too expensive to include all the candidate routes to run the ILP, i.e., too many variables are introduced in the formulation; (2) an observation made from the manual solution is that most of the buses are directly escaped to the boundary. The L-shaped escape routes for those un-assigned buses will be considered in the second run of the ILP.

Given a set of candidate routes  $C$  and a set of buses  $B$ , which are obtained from the first stage in our planner, and  $p$  routing layers, the procedure  $\text{LAYERASSIGNMENT}(C, p, B)$  returns a conflict-free layer assignment  $L = \{l_1, l_2, \dots, l_p\}$ .

From the observation made from our experiment, our two-pass ILP process can be solved very efficiently by modern solvers. The runtime of the layer assignment algorithm is around 30 minutes on a state-of-the-art industrial PCB, while a modern ILP solver cannot finish within one day when the candidate routes are all included in the ILP formulation.

---

LAYERASSIGNMENT( $C, p, B$ )

---

```

 $C_1 \leftarrow$  the projection candidate routes in  $C$ 
Solve the ILP LAYERASSIGNMENT( $C_1, p, B$ )
 $C_L \leftarrow$  the L-shaped routes of the un-assigned buses
 $C_2 \leftarrow C_1 \cup C_L$ 
Solve the ILP LAYERASSIGNMENT( $C_2, p, B$ )
for  $e$  do each  $c \in C$ 
    if  $c$  then  $c$  is assigned onto layer  $j$ 
         $l_j \leftarrow c$ 
    end if
end for return  $L = \{l_1, l_2, \dots, l_p\}$ 

```

---

## 6.6 Resolving Congestion

Since we did not consider congestion in our layer assignment algorithm, congestion has to be resolved upon the obtained layer assignment. The benefit of not considering congestion until now is that the routes on each layer are known after the layer assignment, so we can accurately capture the available routing space on each layer to help resolve congestion. In this section, an ILP is employed on each layer to select the maximum number of nets without any congestion.

We adopt the critical cuts that were introduced in [15] to capture the routing space. Since the layer assignment is known in this stage, for some layer, we can find some components that do not contain any buses being assigned to the layer, which means that the routing spaces within these components on the layer are allowed to be used by other buses. Such components on that layer are called *empty components*. We need to take into account the routing spaces within the empty components in order to accurately capture the available routing space on each layer. So the corresponding empty components on each layer are determined and then ignored during the procedure of creating critical cuts. An example is shown in Figure 6.7, where the critical cuts ignore/cross the empty component  $C_2$  to capture the routing space within  $C_2$ . Once a critical cut is created, its capacity is set to the number of the nets it can accommodate. The capacity defines a congestion constraint, that is, the number of the nets passing the cut cannot exceed its capacity.

For a layer  $t$ , given a set of candidate routes  $l_t$  that are assigned to layer  $t$ , and a set  $R$  of critical cuts  $\{r_1, r_2, \dots, r_m\}$ , an ILP is proposed to select

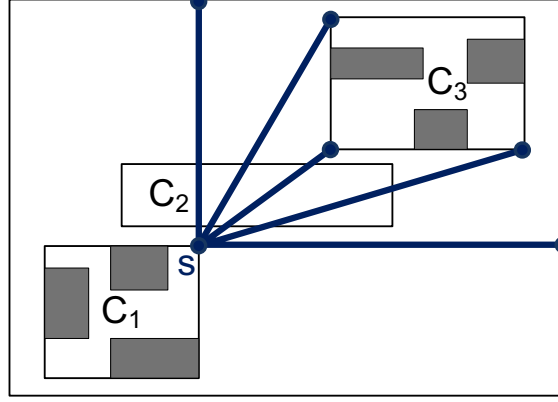


Figure 6.7: An example of the critical cuts considering empty components on one layer. Shaded rectangles denote the buses that are assigned to this layer. Since  $C_2$  has no buses being assigned,  $C_2$  is an empty component on this layer. Thick lines denote the critical cuts starting from the corner  $s$ , which ignore/cross  $C_2$  in order to capture the routing space within  $C_2$ .

the maximum number of nets from  $l_t$  while the congestion constraints are satisfied. Let  $c_i \cap r_i$  denote that the candidate route  $c_i$  crosses the cut  $r_i$ , and let the capacity of  $r_i$  be  $cap_{r_i}$ . The ILP model is formulated as follows:

$$\begin{aligned}
 & \text{Maximize } \sum_{i:c_i \in l_t} (x_{it} \times n_{c_i}) \\
 & \text{Subject to} \\
 & \sum_{i:c_i \in l_t} (x_{it} \times n_{c_i}) \leq cap_{r_i} \quad \text{if } c_i \cap r_i \quad \forall r_i \in R
 \end{aligned}$$

By solving the above ILP model for each layer, we can obtain a layer assignment without conflict and congestion while satisfying the length bounds, which is a valid output of our bus planning problem.

## 6.7 Postprocessing

We further improve the results by trying to reassign the unassigned buses to the routing layers. First we collect the set of all the unassigned buses, and try to reassign them to the first layer. An ILP is formulated by taking into account all the constraints among the set of unassigned buses and the set of buses currently assigned to this layer. By solving this ILP we obtain the maximum set of non-conflicting buses among all the buses thrown into the



ILP, and we assign the buses in this maximum set onto this layer. Note that this result is at least as good as the previous one, as in the worst case the set of buses previously assigned to this layer is selected. Then, we try to put the new set of unassigned buses to the second layer. Similarly, an ILP is formulated and solved, after which we assign the maximum set of non-conflicting buses indicated by the ILP to this layer. Such a procedure is performed iteratively until no further improvement can be gained. The experimental results show that our postprocessing process is simple yet effective and can be done in a short time.

## 6.8 Experimental Results

We implement our bus planner in C++ with *Gurobi Optimizer*[59] employed as our ILP solver. Our experiments are run on a workstation with two 3.0 GHz Intel Xeon CPUs and 4 GB memory.

Our bus planner is tested on a state-of-the-art PCB from industry that has 7000+ nets and 12 routing layers. We compare our results with some other methods, including manual planning, the bus planner proposed in [15], and a state-of-the-art industrial internal bus planner. The results are shown in Table 6.2. The manual design can achieve 100% completion rate. Although [15] achieves 98.5% routing completion, its bus decomposition and escape routing result were directly derived from the manually successfully routed result, and it is hard to reproduce them without the time-consuming manual design process. Only the industrial bus planner and our planner can process each stage automatically. Moreover, our bus planner successfully routed 97.4% of the nets, while the industrial planner only achieved 84.7% routing completion. The 2.6% of nets that are left unrouted by our planner can be routed by manual effort or by using vias. The runtime of our bus planner was less than 2.5 hours, while [15] and the industrial planner took 3 hours and 5 hours, respectively. We can see that our bus planner outperforms the industrial internal planner in terms of both solution quality and runtime, and our bus planner apparently benefits from simultaneously considering the escape routing, bus decomposition, and global routing during layer assignment stage. Compared with manually routing the nets from scratch, our planner is able to save a lot of manual effort by automatically routing most of the

Table 6.2: Each stage of our planner and other methods, and the completion rate on a state-of-the-art industrial PCB

method	Bus decomposition	Escape routing	Layer assignment	Global routing	Completion rate
Manual design	manual	manual	manual	manual	100%
[15]	manual	manual	automatic	automatic	98.5%
Industrial bus planner	automatic	automatic	automatic	automatic	84.7%
Our planner	automatic	automatic	automatic	automatic	97.4%

nets.

## 6.9 Conclusion

In this chapter, we presented an automatic bus planner, which plans everything from scratch: including bus decomposition, escape routing, global routing and layer assignment. In our planner, candidate routes and bus decompositions are generated at first. Then, an ILP-based approach is proposed to generate a conflict-free layer assignment from the candidate routes. Congestion is resolved upon the obtained layer assignment, and then the results are further improved by a postprocessing step. We test our bus planner on a state-of-the-art printed circuit board with over 7000 nets and 12 signal layers. Our bus planner is able to successfully route 97.4% of the nets within 2.5 hours, while a state-of-the-art industrial internal bus planner can route 84.7% of the nets within 5 hours.

# CHAPTER 7

## CONCLUSIONS AND FUTURE WORK

In this dissertation, we study several challenging EDA problems. Topics that have been covered in our study fall into these three categories: designs, manufacturing, and packaging.

First, we study the timing closure problem. In Chapter 2, we present an optimization flow that can work on a circuit-level design. The optimization flow is to solve hold violations by inserting buffers as delay elements. The flow is based on linear programming that captures the different delays introduced between setup constraints and hold-time constraints. Ours is the first work that identifies this issue and models it into the linear programming model. In order to work on industrial modern designs, the main challenges in modern industrial designs such as discrete cell sizes, accurate cell timing models, and complex timing constraints are considered in our approach. Then, in Chapter 3, we study the min-cost buffer insertion problem for hold violations. The goal of the min-cost buffer insertion problem is to insert a buffer chain to solve the hold violations while the timing constraints are still met, meanwhile keep the area of the buffer chain to a minimum. We first propose an optimal algorithm. And then based on the optimal algorithm, a heuristic algorithm and a machine learning based algorithm are proposed. Ours is the first work to apply machine learning to the buffer insertion problem.

We then study the aerial image simulation problem. Due to the regularity of this problem and the extremely large volume of data, a parallel implementation method can effectively leverage today’s high performance computing platforms. We first further improve the sequential approach. Then, two approaches are proposed based on the SIMD (single-instruction multiple-data) CPU. Finally, we further accelerate the approaches by using multi-threading. The experiment shows that an explicit tuning is necessary in order to fully exploit the computing capabilities of modern multi-core SIMD CPU.

Finally, we study the problems for the modern PCBs. In Chapter 5, we

study the escape routing problem on staggered pin arrays. Network flow models are used to model the capacities of the staggered pin arrays. With the correctness of the proof for the network flow models, optimal algorithms are proposed. Then, in Chapter 6, we study the bus planning problem. Ours is the first complete bus planner in the literature. A complete bus planner has to simultaneously solve the bus decomposition, escape routing, layer assignment and global bus routing. Our approach is able to model these stages into an integer linear programming model.

To conclude this dissertation, we would like to point out some future directions for the timing closure problem, which still has a lot of constraints on modern designs that have to be considered. For example, the congestion of the neighborhood of a target pin has to be taken into consideration during the optimization, otherwise the buffer insertion would increase the difficulty of a detailed router, which is already a much more complicated problem as the technology advances. Similarly, the number of inserted buffers has to be minimized while the area of the inserted buffers is kept to a minimum, as it also impacts the routability. Hold violation removal typically happens in the last few stages of the design flow, so less disturbability in the designs is preferred.

## REFERENCES

- [1] A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu, *VLSI Physical Design: From Graph Partitioning to Timing Closure*. Springer Science & Business Media, 2011.
- [2] L. P. Van Ginneken, “Buffer placement in distributed rc-tree networks for minimal elmore delay,” in *Proc. Int. Symp. Circuits and Systems*, 1990, pp. 865–868.
- [3] J. Lillis, C.-K. Cheng, and T.-T. Y. Lin, “Optimal wire sizing and buffer insertion for low power and a generalized delay model,” *Solid-State Circuits, IEEE Journal of*, vol. 31, no. 3, pp. 437–447, 1996.
- [4] W. Shi and Z. Li, “A fast algorithm for optimal buffer insertion,” *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 24, no. 6, pp. 879–891, 2005.
- [5] C. Alpert, C. Chu, G. Gandham, M. Hrkic, J. Hu, C. Kashyap, and S. Quay, “Simultaneous driver sizing and buffer insertion using a delay penalty estimation technique,” *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 23, no. 1, pp. 136–141, 2004.
- [6] T. Okamoto and J. Cong, “Buffered steiner tree construction with wire sizing for interconnect layout optimization,” in *Proc. Int. Conf. on Computer-Aided Design*. IEEE Computer Society, 1997, pp. 44–49.
- [7] M. Kang, W.-M. Dai, T. Dillinger, and D. LaPotin, “Delay bounded buffered tree construction for timing driven floorplanning,” in *Proc. Int. Conf. on Computer-Aided Design*. IEEE, 1997, pp. 707–712.
- [8] H. Zhou, D. Wong, I.-M. Liu, and A. Aziz, “Simultaneous routing and buffer insertion with restrictions on buffer locations,” *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 19, no. 7, pp. 819–824, 2000.
- [9] M. M. Ozdal, S. Burns, and J. Hu, “Gate sizing and device technology selection algorithms for high-performance industrial designs,” in *Proc. Int. Conf. on Computer-Aided Design*, 2010, pp. 724–731.

- [10] J. Cong and Y. Zou, "FPGA-based hardware acceleration of lithographic aerial image simulation," *ACM Trans. on Reconfigurable Technology and Systems*, vol. 2, no. 3, Sep. 2009.
- [11] H. Zhang, T. Yan, M. D. F. Wong, and S. J. Patel, "Accelerating aerial image simulation with GPU," in *Proc. Int. Conf. on Computer-Aided Design*, 2011, pp. 178–184.
- [12] J. C. Whitaker, Ed., *The Electronics Handbook*, 2nd ed. CRC Press, 2005.
- [13] Y.-K. Ho, H.-C. Lee, and Y.-W. Chang, "Escape routing for staggered-pin-array PCBs," in *Proc. Int. Conf. on Computer-Aided Design*, 2011, pp. 306–309.
- [14] H. Kong, T. Yan, M. D. F. Wong, and M. M. Ozdal, "Optimal bus sequencing for escape routing in dense PCBs," in *Proc. Int. Conf. on Computer-Aided Design*, 2007, pp. 390–395.
- [15] H. Kong, T. Yan, and M. D. F. Wong, "Automatic bus planner for dense PCBs," in *Proc. Design Automation Conf.*, 2009, pp. 326–331.
- [16] H. Kong, Q. Ma, T. Yan, and M. D. F. Wong, "An optimal algorithm for finding disjoint rectangles and its application to PCB routing," in *Proc. Design Automation Conf.*, 2010, pp. 212–217.
- [17] Q. Ma, E. F. Y. Yong, and M. D. F. Wong, "An optimal algorithm for layer assignment of bus escape routing on PCBs," in *Proc. Design Automation Conf.*, 2011, pp. 176–181.
- [18] T. Yan, H. Kong, and M. D. F. Wong, "Optimal layer assignment for escape routing of buses," in *Proc. Int. Conf. on Computer-Aided Design*, 2009, pp. 245–248.
- [19] Q. Ma, H. Kong, M. D. F. Wong, and E. F. Y. Young, "A provably good approximation algorithm for rectangle escape problem with application to PCB routing," in *Proc. Asia and South Pacific Design Automation Conf.*, 2011, pp. 843–848.
- [20] N. V. Shenoy, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Minimum padding to satisfy short path constraints," in *Proc. Int. Conf. on Computer-Aided Design*, 1993, pp. 156–161.
- [21] S.-H. Huang, G.-Y. Jhuo, and W.-L. Huang, "Minimum buffer insertions for clock period minimization," in *Proc. Computer Communication Control and Automation*, vol. 1, 2010, pp. 426–429.

- [22] S.-H. Huang, C.-H. Cheng, C.-M. Chang, and Y.-T. Nieh, "Clock period minimization with minimum delay insertion," in *Proc. Design Automation Conf.*, 2007, pp. 970–975.
- [23] C. Lin and H. Zhou, "Clock skew scheduling with delay padding for prescribed skew domains," in *Proc. Design Automation Conf.*, 2007, pp. 541–546.
- [24] W.-P. Tu, C.-H. Chou, S.-H. Huang, S.-C. Chang, Y.-T. Nieh, and C.-Y. Chou, "Low-power timing closure methodology for ultra-low voltage designs," in *Proc. Int. Conf. on Computer-Aided Design*, 2013, pp. 697–704.
- [25] L.-D. Huang, M. Lai, D. Wong, and Y. Gao, "Maze routing with buffer insertion under transition time constraints," in *Proc. Conf. on Design Automation and Test in Europe*. IEEE, 2002, pp. 702–707.
- [26] C. N. Sze, C. J. Alpert, J. Hu, and W. Shi, "Path-based buffer insertion," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 26, no. 7, pp. 1346–1355, 2007.
- [27] Y. Gao and D. Wong, "A graph based algorithm for optimal buffer insertion under accurate delay models," in *Proc. Conf. on Design Automation and Test in Europe*. IEEE Press, 2001, pp. 535–539.
- [28] W. Shi, Z. Li, and C. J. Alpert, "Complexity analysis and speedup techniques for optimal buffer insertion with minimum cost," in *Proc. Asia and South Pacific Design Automation Conf.* IEEE, 2004, pp. 609–614.
- [29] R. Chen and H. Zhou, "Fast min-cost buffer insertion under process variations," in *Proc. Design Automation Conf.* ACM, 2007, pp. 338–343.
- [30] B. Welford, "Note on a method for calculating corrected sums of squares and products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962.
- [31] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, Revised Edition*. Reading, Mass.: Addison-Wesley, 1969.
- [32] N. B. Cobb, "Fast optical and process proximity correction algorithms for integrated circuit manufacturing," Ph.D. dissertation, University of California at Berkeley, 1998.
- [33] A. K.-K. Wong, *Optical Imaging in Projection Microlithography*. SPIE Press, 2005.



- [34] I. Torunoglu, A. Karakas, E. Elsen, C. Andrus, B. Bremen, and P. Thoutireddy, "OPC on a single desktop: A GPU-based OPC and verification tool for fabs and designers," in *Proc. of SPIE*, vol. 7641, no. 764114, 2010.
- [35] I. Uzun, A. Amira, and A. Bouridane, "FPGA implementations of fast fourier transforms for real-time signal and image processing," in *Proc. Vision, Image, Signal Process*, vol. 152, no. 3, 2005, pp. 283–296.
- [36] Y.-T. Wang, C.-M. Tsai, and F.-C. Chang, "Lithographic simulations using graphical processing units," 2006, US Patent Application 20060242618.
- [37] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobbs Journal*, vol. 30, no. 3, pp. 202–210, 2005.
- [38] A. Heinecke, T. Auckenthaler, and C. Trinitis, "Exploiting state-of-the-art x86 architectures in scientific computing," in *Proc. Int. Symp. on Parallel and Distributed Processing Workshops and Phd Forum*, June 2012, pp. 47–54.
- [39] R. Goering, "DAC report: GPUs or multicore for EDA applications?" <http://www.cadence.com/Community/blogs/ii/archive/2009/07/31/dac-report-gpus-or-multicore-for-eda-applications.aspx>, July 2009.
- [40] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-Core CPU and GPU," in *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 78–88.
- [41] D. Payne, "GPU versus multicore in EDA," <http://www.chipdesignmag.com/payne/2009/04/13/gpu-versus-multicore-in-eda>, Apr. 2009.
- [42] S. P. Khatri and K. Gulati, *Hardware Acceleration of EDA Algorithms: Custom ICs, FPGAs and GPUs*, 1st ed. Springer, 2010.
- [43] Intel<sup>®</sup> 64 and IA-32 Architectures Software Developer's Manual, *Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C*, Intel Corporation, Aug. 2012.
- [44] "OpenMP: The OpenMP API specification for parallel programming," <http://openmp.org/wp>.
- [45] T. Yan and M. D. F. Wong, "Recent research development in PCB layout," in *Proc. Int. Conf. on Computer-Aided Design*, 2010, pp. 398–403.

- [46] T. Yan and M. D. F. Wong, "A correct network flow model for escape routing," in *Proc. Design Automation Conf.*, 2009, pp. 332–335.
- [47] W.-T. Chan, F. Y. L. Chin, and H.-F. Ting, "A faster algorithm for finding disjoint paths in grids," in *Proc. Int. Symp. on Algorithms and Computation*, 1999, pp. 393–402.
- [48] J.-W. Fang and Y.-W. Chang, "Area-I/O flip-chip routing for chip-package co-design," in *Proc. Int. Conf. on Computer-Aided Design*, 2008, pp. 518–522.
- [49] J.-W. Fang, I.-J. Lin, Y.-W. Chang, and J.-H. Wang, "A network-flow-based RDL routing algorithm for flip-chip design," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 26, no. 8, Aug. 2007.
- [50] D. Wang, P. Zhang, C.-K. Cheng, and A. Sen, "A performance-driven I/O pin routing algorithm," in *Proc. Asia and South Pacific Design Automation Conf.*, 1999, pp. 129–132.
- [51] R. Wang, R. Shi, and C.-K. Cheng, "Layer minimization of escape routing in area array packaging," in *Proc. Int. Conf. on Computer-Aided Design*, 2006, pp. 815–819.
- [52] M.-F. Yu, J. Darnauer, and W. W.-M. Dai, "Interchangeable pin routing with application to package layout," in *Proc. Int. Conf. on Computer-Aided Design*, 1996, pp. 668–673.
- [53] R. Shi and C.-K. Cheng, "Efficient escape routing for hexagonal array of high density I/Os," in *Proc. Design Automation Conf.*, 2006, pp. 1003–1008.
- [54] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Feb. 1993.
- [55] T. Yan and M. D. F. Wong, "Correctly modeling the diagonal capacity in escape routing," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 31, no. 2, pp. 285 – 293, Feb. 2012.
- [56] D. Staepelaere, J. Jue, T. Dayan, and W. W.-M. Dai, "SURF: Rubber-band routing system for multichip modules," *IEEE Des. Test. Comput.*, vol. 10, no. 4, pp. 18–26, Dec. 1993.
- [57] D. J. Staepelaere, "Geometric transformations for a rubber-band sketch," M.S. thesis, University of California at Santa Cruz, Santa Cruz, CA, USA, Sep. 1992.
- [58] "CS2: min-cost flow solver," <http://www.igsystems.com/cs2/index.html>, copyright: 1995-2006 IG Systems, Inc.

[59] “Gurobi optimizer,” <http://www.gurobi.com>.