The Dissertation Committee for Yilin Zhang
certifies that this is the approved version of the following dissertation:

Interconnect Optimizations for Nanometer VLSI Design

Committee:

David Z. Pan, Supervisor

Andreas Gerstlauer

Nur A. Touba

Michael Orshansky

Salim Chowdhury

Interconnect Optimizations for Nanometer VLSI Design

by

Yilin Zhang, B.S.; M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2014

This dissertation is lovingly dedicated to my my mother Yanjun Huang and my father Shijin Zhang who constantly provide all their love and all they own to me in every single day of my life. It is also dedicated to friends who earnestly help me on my paper writing and problem solving during my PhD.

Acknowledgments

I want to thank my adviser, Prof. David Z. Pan, first, for his guidance, understanding and support during my Ph.D. studies at the University of Texas at Austin. He could always find a clear path for me when I was confused or trapped in certain problem. It was him that made me understand that staying with big picture was way more important than optimizing some sub-problems. Also, he had great understanding about my situations and feelings outside research/study. He provided me long enough time to spend with my families when my families and me preferred that. He also very kindly asked about my families when he knew that my father was not in a good health condition. Retrospectively, I was so lucky to meet Prof. Pan during his visit in my college, Peking University, when I was considering to apply for PhD. It was a correct decision to join UTDA and finished my PhD under the supervision of Prof. Pan.

I would like to thank the current and former UTDA members for all great days we spent together no matter for research discussions or ball sporting together. They are Ashutosh Chakraborty, Kun Yuan, Anurag Kumar, Ou He, Katrina Lu, Jae-Seok Yang, Xiaoqing Xu, Shuojing Wang, Jerrica Gao, Yibo Lin, Duo Ding, Jiwoo Pak, Bei Yu, Subhendu Roy, Yongchan (James) Ban, Wooyoung Jang, Samuel Ward, Wen Zhang, Yang Li, Boyang Zhang, Yen-

Hung Lin, Joydeep Mitra, Jiaojiao Ou, Abhishek Bhaduri and Che-Lun Hsu. From their self-giving help, I learned lots of helpful skills, such as how to properly arrange everything to boost overall efficiency, how to devote yourself on one project for days and nights, etc. Also, the time we played tennis, Pingpong or basketball together really make us good friends and I would miss every moment we spent together.

I am also deeply thankful to Salim Chowdhury and Zhuo Li who were my mentors during my interns at Oracle and IBM, respectively. Basically I started my PhD project during the cooperation between UTDA and Oracle. Dr. Chowdhury always had a lot of creative ideas so we can work on together. We keep exchange our ideas during my whole PhD and honorably he accepted my invitation to be one of my committee members. I would also like to thank other co-workers during my intern at Oracle: Rajendran Panda, Akshay Sharma, Duo Ding, Kevin Grant, Yong Li, Boyang Zhang, Zhen Liu, et al. Dr. Li was my mentor when I spent six months doing intern at IBM. His amazing working efficiency and problem solving ability was one of great lessons I learned at IBM. I deeply thank him for many interesting projects he provided me. I would deeply thank to my colleagues in IBM. They are Yuhan Zhou, Chuck Alpert, Tiago Reimann, Ying (Nancy) Zhou, Yaoguang Zhou, Cliff Sze, Gi-Joon Nam, Natarajan Viswanathan, Myung-Chul Kim, et al.

My sincere thanks also go to my three other Ph.D. committee members, Prof. Nur A. Touba, Prof. Michael Orshansky and Prof. Andreas Gerstlauer. Thank you for bring out various questions during my Ph.D. proposal exam

which inspired me to explore more interesting problems that I could not find by myself.

Last but not least, I would like to thank my parents. They always give me maximum understanding when I studied PhD in another country and could not well perform my responsibility as a son to take care of them when they are growing old. I love you.

Interconnect Optimizations for Nanometer VLSI Design

Publication No. _____

Yilin Zhang, Ph.D.
The University of Texas at Austin, 2014

Supervisor: David Z. Pan

As the semiconductor technology scales into deeper sub-micron domain, billions of transistors can be used on a single system-on-chip (SOC) makes interconnection optimization more important roughly for two reasons. First, congestion, power, timing in routing and buffering requirements make interconnection optimization more and more challenging. Second, gate delay getting shorter while the RC delay gets longer due to scaling.

Study of interconnection construction and optimization algorithms in real industry flows and designs ends up with interesting findings. One used to be overlooked but very important and practical problem is how to utilize over-the-block routing resources intelligently. Routing over large IP blocks needs special attention as there is almost no way to insert buffers inside hard IP blocks, which can lead to unsolvable slew/timing violations. In current design flows we have seen, the routing resources over the IP blocks were either dealt as routing blockages leading to a significant waste, or simply treated in the

same way as outside-the-block routing resources, which would violate the slew constraints and thus fail buffering.

To handle that, this work proposes a novel buffering-aware over-the-block rectilinear Steiner minimum tree (BOB-RSMT) algorithm which helps reclaim the "wasted" over-the-block routing resources while meeting user-specified slew constraints. Proposed algorithm incrementally and efficiently migrates initial tree structures with buffering-awareness to meet slew constraints while minimizing wire-length.

Moreover, due to the fact that timing optimization is important for the VLSI design, in this work, timing-driven over-the-block rectilinear Steiner tree (TOB-RST) is also studied to optimize critical paths. This proposed TOB-RST algorithm can be used in routing or post-routing stage to provide high-quality topologies to help close timing.

Then a follow-up problem emerges: how to accomplish the whole routing with over-the-block routing resources used properly. Utilizing over-the-block routing resources could dramatically improve the routing solution, yet require special attention, since the slew, affected by different RC on different metal layers, must be constrained by buffering and is easily violated. Moreover, even of all nets are slew-legalized, the routing solution could still suffer from heavy congestion problem. A new global router, BOB-Router, is to developed solve the over-the-block global routing problem through minimizing overflows, wire-length and via count simultaneously without violating slew constraints. Based on my completed works, BOB-RSMT and BOB-Router tremendously

improve the overall routing and buffering quality.

Experimental results show that proposed over-the-block rectilinear Steiner tree construction and routing completely satisfies the slew constraints and significantly outperforms the obstacle-avoiding rectilinear Steiner tree construction and routing in terms of wire-length, via count and overflows.

Table of Contents

List of Tables

List of Figures

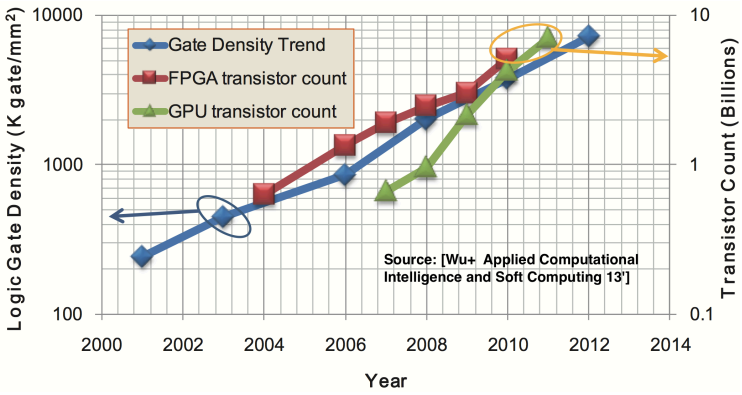
Chapter 1

Introduction

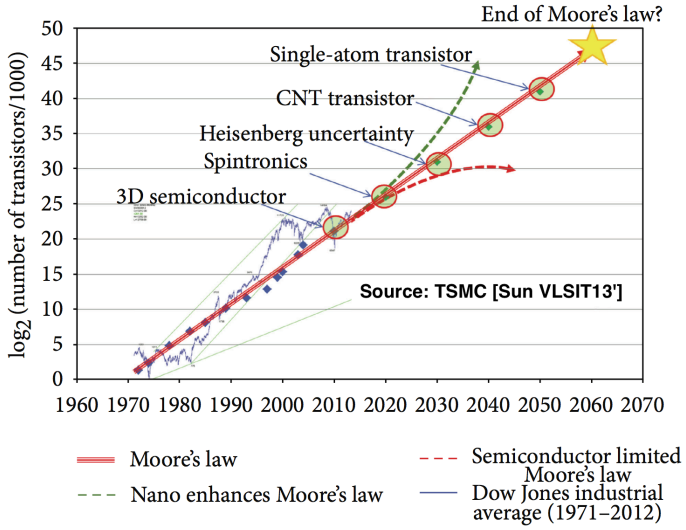
1.1 Challenges in Interconnection Optimization

Due to technology scaling, the number of transistors on a single system-on-chip (SOC) is expanding dramatically. According to ITRS [3], the combination of 3D device architecture and low power device will usher the (Third) Era of Scaling. New device, such as spin wave device (SWD) is able to convert input voltage signals into the spin waves, compute with spin waves and convert the output spin waves into the voltage signals. Materials, such as III-V and Ge can further improve device performance with higher mobility. Stacking multiple layers of transistors continually compacts more transistors per unit chip area. In brief, by these novel inventions, semiconductor technology will keep in scaling continually which puts more gates in a chip and increases cell density (Fig. 1.1). Thus, the emerging challenge is: modern VLSI designs with hundred billions of gates will turn out to be extremely complicated to design, which in turn places more demands on computer aided design (CAD), especially physical design.

Physical design is the stage which turns RTL code into GDSII before fabrication. Physical design consists of partitioning, floorplanning, placement,



(a)



(b)

Figure 1.1: (a) Transistor count and gate density continues to increase. [49] (b) Moores Law history, future, limited factors, and Nano-technology-enhance factors for Moores Law and compared with Dow Jones industrial average in the same period of time (1971—2012). [50]

CTS, routing, etc (Fig. 1.2). This dissertation will focus on interconnection optimization which includes routing, in particular global routing, buffering,

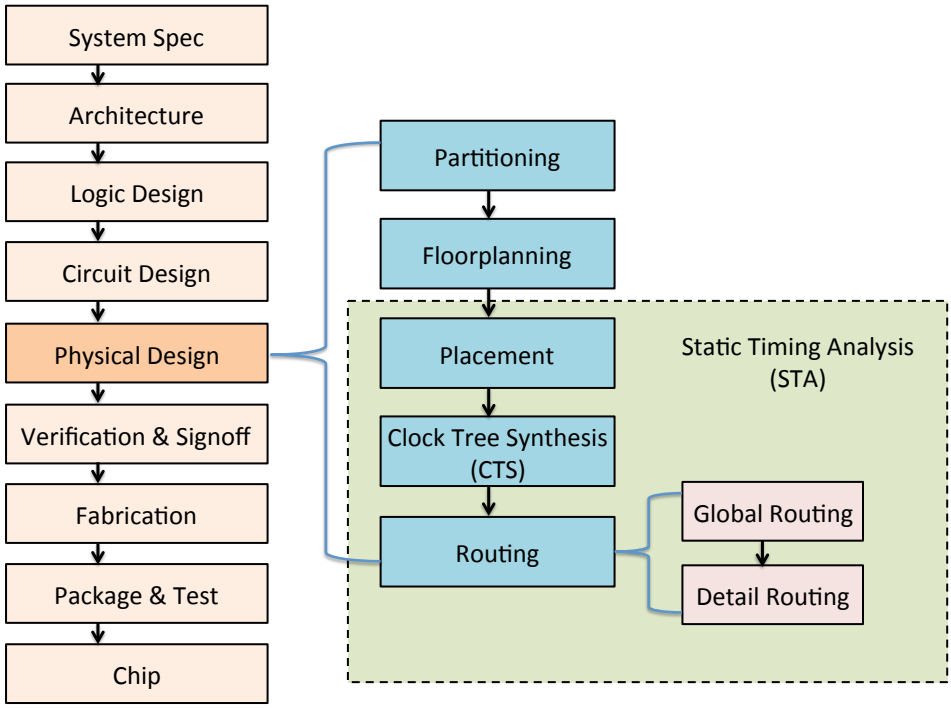


Figure 1.2: A simplified VLSI design and physical design flow.

tree construction, together with static timing analysis (STA). Interconnection optimization plays an important role in the physical design flow. It will automatically generate and optimize topologies for all nets in modern VLSI design with both performance and power considerations. With technology scaling, interconnection optimization is becoming more challenging for two reasons:

1. The portion of RC interconnection delay in the overall delay is dramatically growing due to increasing RC interconnection delay and decreasing

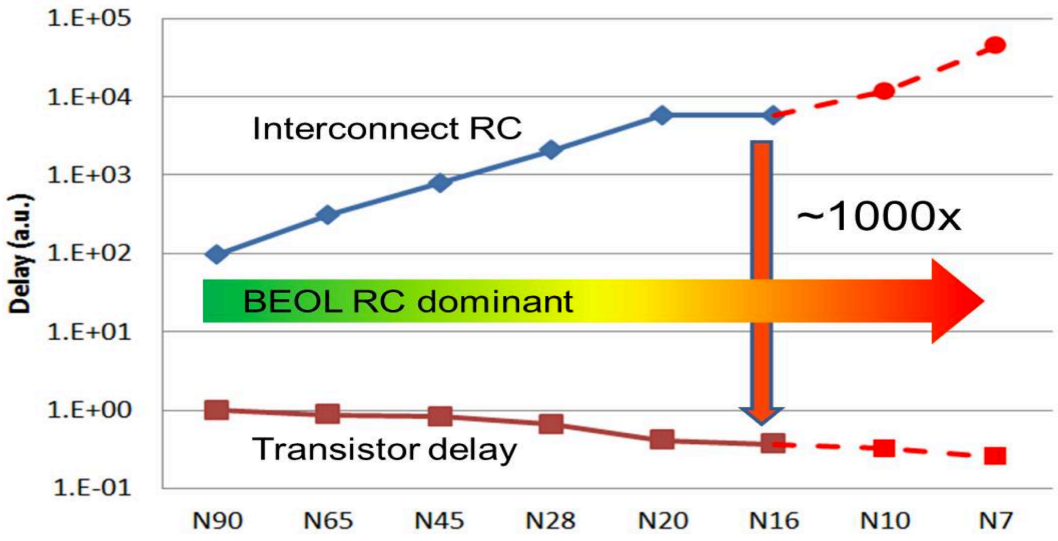


Figure 1.3: Interconnect RC delay v.s. gate delay.

gate delay as CMOS scaling (Fig. 1.3 [54]).

2. Interconnection optimization becomes more challenging as increasing total wire-length and cell density amplifying the congestion and routability problem.

1.2 How Interconnection Optimization Works

After placement is done, all functional blocks and gates are fixed with location but not wiring between logic gates. As above mentioned, the process of planning all wires is interconnection optimization which becomes more and more challenging and crucial nowadays.

Routing, particularly global routing, is the main part of the intercon-

nection planning process. As shown in Fig. 1.2, routing consists of two parts: global routing followed by detail routing. This division separates this extremely complex problem into two relatively easier sub-problems. Global routing is performed on a coarse-grain grid, which depicts the rough shape of each net. With coarse-grain grid, it provides smaller solution space, which stands for relatively less runtime for this NP-complete problem [33]. On the other hand, detail routing is based on the global routing solution with a fine-grain grid. Detail routing solves legalization issues with exact routes. Because detail routing is to find the exact routes based on global routing, the quality of final routes is primarily depend on the global routing solution. Therefore, a powerful global routing needs to find route for each net with wire-length, routability and timing co-optimization.

Other components in interconnection optimization is actually surrounding routing process. Rectilinear Steiner minimum tree (RSMT) construction is one fundamental physical design problem to achieve routing and buffering quality. RSMT is to connect all pins in a net in horizontal or vertical way. Fig. 1.4 is one example of RSMT connecting twenty pins. During routing, every net among all hundred billions nets requires RSMT construction or incremental RSMT re-construction. This classical problem has long been proved as NP-complete [41] and many works have been performed including recent breakthrough, e.g. the well-known FLUTE [18].

Because RSMT is only targeting at minimize total wire-length, which is not enough for high performance VLSI design. Timing driven RSMT (TD-

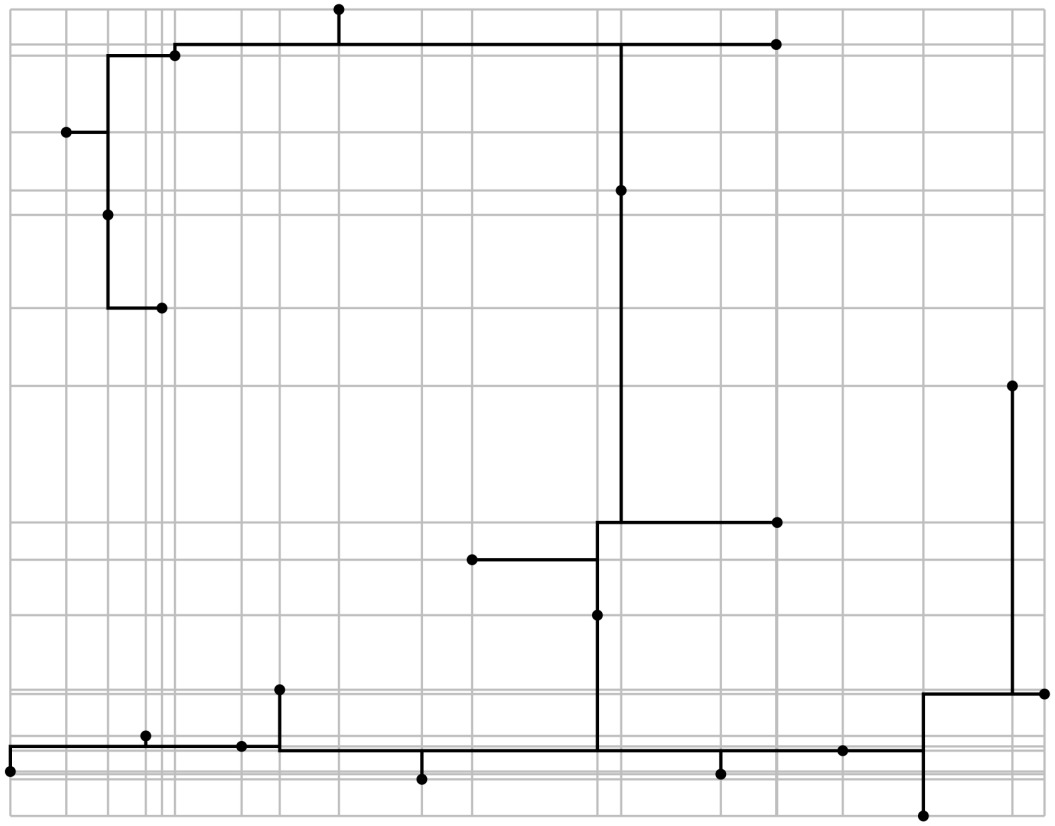


Figure 1.4: RSMT on hanan grid with black dots as pins.

RSMT), instead, forges better trade-off on timing and wire-length. With scaling, interconnection delay has become the dominant factor in determining circuit speed, contributing up to 50% $\sim$ 70% of the clock cycle in high performance circuit [19]. Since STA is performed during placement and routing stages, it is common that critical paths information is available during current RST construction process. With these criticality information, TD-RSMT will trade wire-length for shorter delay on critical paths. In Fig. 1.5, it shows

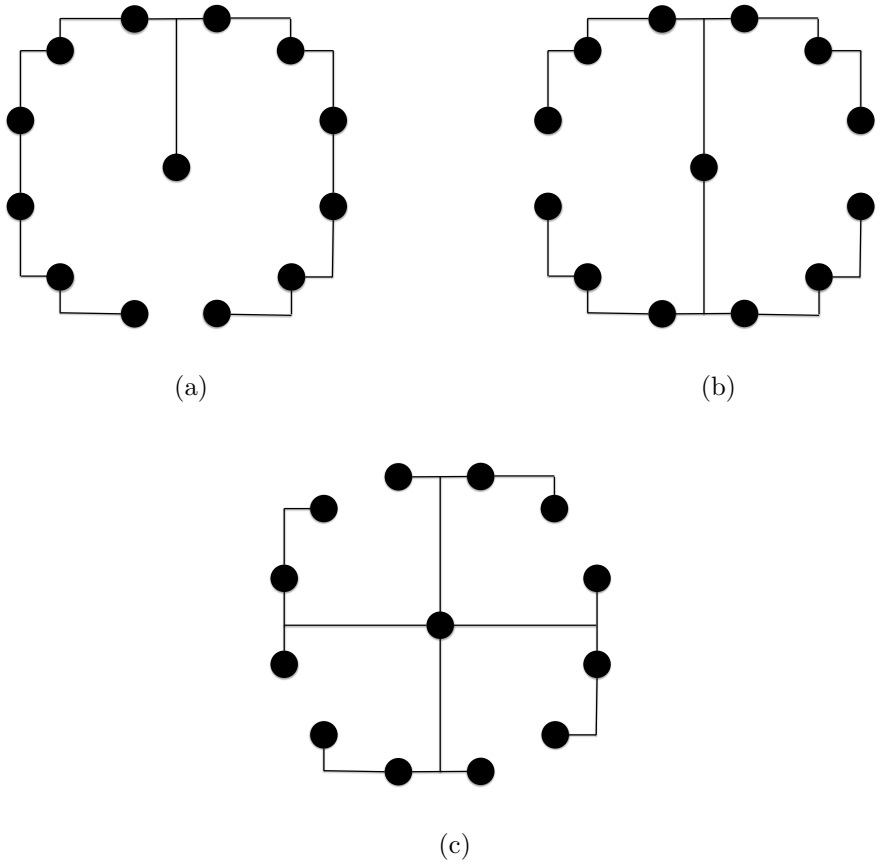


Figure 1.5: (a) RSMT topology. (b) Rectilinear Steiner tree with better source-to-critical-sink delay but worse wire-length. (c) Topology with the best source-to-critical-sink delay but worst wire-length.

the trade-off of wire-length and delay to sinks (assume all sinks are critical). RSMT will construct the net as Fig. 1.5(a) which consumes the least wire-length, yet delay to certain sinks is very long. On the other hand, if sacrificing some wire-length, a new topology as in Fig. 1.5(b) is generated with better source-to-critical-sink delay but more wire-length. To be extremely on delay

optimization, topology in Fig. 1.5(c) is the worst at wire-length but the best at timing optimization.

After RSMT construction, buffering will be performed over each tree to linearize the interconnection delay on long interconnections and shield branch-capacitance. Buffers can re-strengthen signals as well as reduce delays. Due to dominance of interconnection delay, the critical length, i.e. minimum distance beyond which inserting an optimal-sized buffer makes the interconnect delay smaller, is decreasing, which requires more and more buffers inserted in a chip. It is reported that in 32–nm technology, it reaches an alarming point that 70% of cells are buffers [48]. Besides, it is reported that in reality, *slew mode buffering* is more predominant than timing mode buffering [28, 44]. Only a fraction (roughly $5\% \sim 10\%$) of nets needs to be buffered for delay optimization while for the remaining (roughly $90\% \sim 95\%$) are sufficient with slew mode buffering to meet the slew constraints. Hence, a fast and powerful slew mode buffering algorithm is crucial during interconnection optimization.

In slew mode buffering, slew needs to be calculated over and over. Moreover, slew calculation is repeatedly performed in STA, clock tree synthesis, routing and sizing locally or globally. Because of these reason, this dissertation adopts a simple but effective slew calculation model, i.e. PERI [34] model, for slew calculation. It shows the error of PERI is within 1% [34], which is indistinguishable from what is obtained using SPICE simulation.

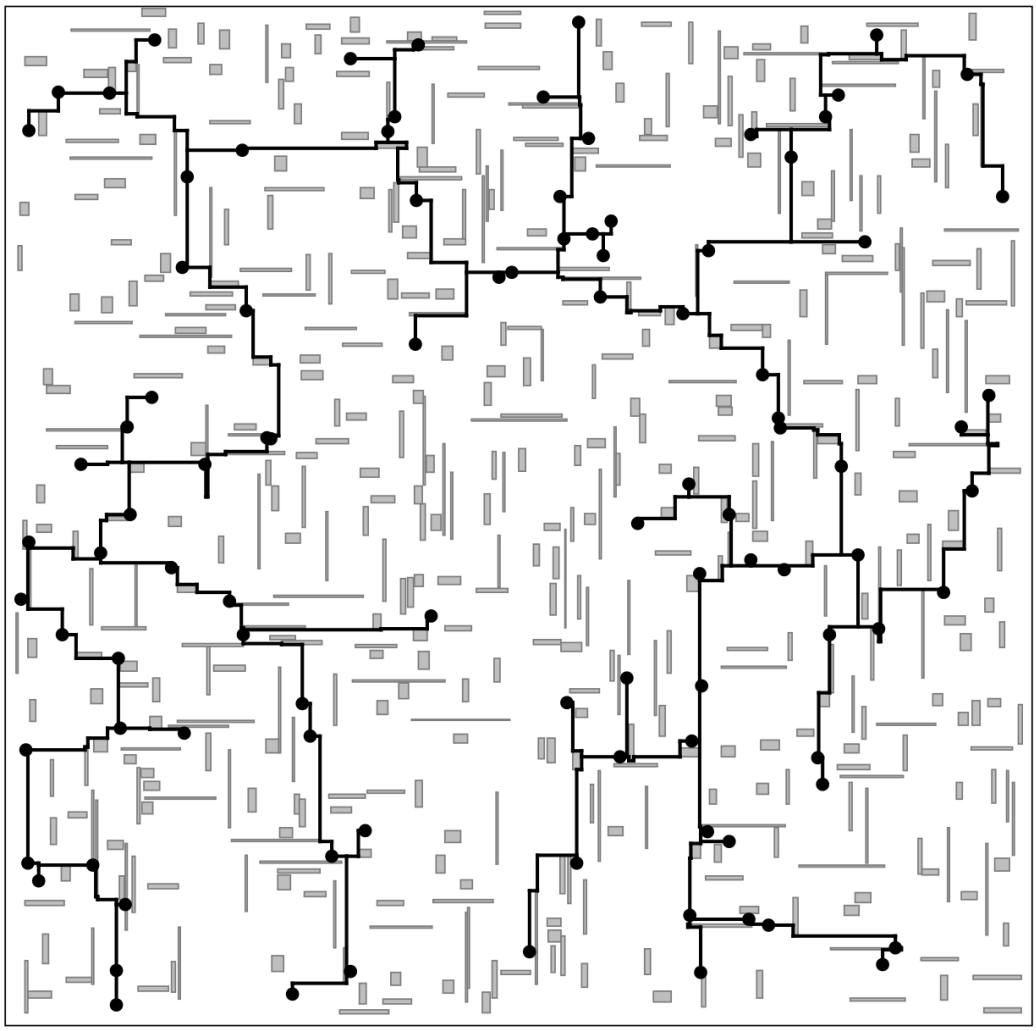


Figure 1.6: One example of OA-RSMT [37].

1.3 Critical Problems in Interconnection Optimization

Because of extensively using IP-blocks to shorten turn around time, SOC designs nowadays are packed with IP blocks or macros. Since it is forbid-

den to insert any buffer on those pre-designed IP blocks or macros, traditional RSMT algorithm will result in impractical topology which is unfeasible for buffering. Although there are studies of the so-called buffer planning [22] or suggestions to even put unconnected buffers inside IP blocks [7], in practice, most IP blocks still do not have pre-placed "idle" buffers. RSMT construction avoiding these blockages is the most simple and straight way to handle that. It is studied and well known as the OA-RSMT problem. OA-RSMT problem has been studied actively in the last few years (e.g., [6, 30, 35, 36]). Early approaches [35, 36] only deal with rectangular blockages, while a most recent study [30] can tackle rectilinear blockages without dissecting rectilinear blockages into rectangular ones. This approach can eliminate the unfeasible solutions which put wires and buffers between adjoining blocks. However, all these OA-RSMT algorithms simply treat IP blocks as routing blockages, which would significantly waste routing resources over these IP blocks and cause more congestion issues. Fig. 1.6 illustrates an example of OR-RSMT.

Indeed, most IP blocks such as SRAMs only use certain lower metal layers. There are still considerable amount of routing resources available at higher metal layers over these IP blocks, even if we take into consideration the resources reserved for power/ground and clock routing. If we simply treat the IP blocks as routing obstacles, these over-the-block routing resources will be "wasted", which leads to more routing demand elsewhere.

In order to use "wasted" routing resources while still enable feasible buffering, this dissertation studies a new class of buffering-aware over-the-

block rectilinear Steiner minimum tree (BOB-RSMT) problem. This dissertation develops an effective algorithm which tries to intelligently reclaim the "wasted", over-the-IP-block routing resources by previous approaches while ensuring slew constraints for high quality buffering. Proposed algorithm incrementally updates the initial RSMT structure obtained from FLUTE [18] to satisfy slew constraints while minimizing wire-length (FLUTE is chosen to be the initial RSMT generator because its low runtime and high quality). A restricted length, over-the-block maze routing algorithm is developed to reconnect any part of BOB-RSMT which is dissected during the optimization process.

RSMT and related extensions produce good results regarding wire-length minimization, which contributes to routability and power optimization. However, there are certain amount of nets which are critical nets. These critical nets are eager for timing optimization other than power and wire-length. Since straight paths will give less delay compared with detoured paths, it is necessary to place timing critical nets over-the-block than avoiding-the-block. Furthermore, using over-the-block routing resources could unburden the outside-the-block congestion which in turn decreases power and delay. This dissertation proposes a timing-Driven, over-the-block rectilinear Steiner tree (TOB-RST) construction algorithm with pre-buffering and slew constraints in consideration. TOB-RST intelligently utilizes over-the-block routing resources, and the resulted tree is buffering-feasible and slew-violation-free.

With the algorithm of building over-the-block RSMT, the whole global

routing problem considering over-the-block routing resources is the next emergent problem to solve. The CEDA-sponsored ISPD Global Routing Contests [4] and [5] attract attention from dozens of academic and industrial participants. Inspired by the competitions, many high-performance global routers are published.

However, due to guidance from two ISPD Global Routing Contests are similar, most published modern routers are aiming at the same problem: minimizing wire-length and via count in addition to alleviating congestion. However, the global routing problem has never been touched upon to not only consider wire-length, vias and overflows, but also properly use over-the-block routing resources. Studying this new problem is essential as to shorten the design cycle and improve the chip quality. If over-the-block routing resources are treated the same as that for out-the-block, long nets over the block will fail buffering, leading to additional manual work; whereas over-the-block routing resources are totally avoided, less remaining routing resources will significantly deteriorate the quality of the routing solution.

This dissertation studies a new class of buffering-aware over-the-block global router (BOB-ROUTER) which tries to intelligently reclaim the "wasted" over-the-IP-block routing resources while minimizing overflows, wire-length and via count as in "basic" routers. The generated topologies are aware of slew constraints which guarantees feasible buffering.

1.4 Contributions

This dissertation has the following major contributions:

- This is the first work that proposes a practical formulation of buffering-aware over-the-block RSMT. Quality of the Steiner tree and feasible buffering are ensured as considering slew constraints with wire-length simultaneously. Our algorithm is able to integrate with a buffering tool to generate a low buffering cost BOB-RSMT without violating maximum slew constraint, which can be used in floorplanning, placement and routing stages. An incremental approach of fixing slew violation one by one is used to satisfy slew constraints on over-the-block part of BOB-RSMT, followed by a restricted length, over-the-block maze routing algorithm which reconnects any dissected part of BOB-RSMT during the optimization process.

- It is first time a comprehensive timing-driven RST is studied which includes: (1) pre-buffering algorithm pre-characterizes the tree topology and buffer distribution to provide accurate timing information for final TD-RST construction, (2) proposed TOB-RSMT reclaims the wasted over-the-block routing resources while meeting user-specified timing (slack and slew) constraints, and (3) before fixing topology, a topology-tuning is performed based on location of buffers to improve timing without increasing buffering cost.

- For the first time, a router tries to solve the over-the-block global routing problem through minimizing overflows, wire-length and via count simultaneously without violating slew constraints. First, an integer linear programming (ILP) formulation is used to characterize the object (wire-length and via) and constraints (overflow and slew). Second, the ILP formulation is relaxed into a LP formulation. Third, solving the Lagrangian relaxation of the LP formulation provides the price of each edge in the 3D routing model. Last, a RC-constrained A* search is applied to help explore new buffering-aware topologies on all metal layers.

1.5 Organization of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 formalizes a practical and new problem in rectilinear Steiner construction: BOB-RSMT. It demonstrates the importance of buffering-aware over-the-block RSMT and lack of that in previous works. Chapter 3 presents a solution for timing-driven pre-buffering rectilinear Steiner tree with over-the-block consideration. It illustrates the trade-off of timing and power for critical nets and non-critical nets. Chapter 3 also exhibits a post-tree-construction tuning algorithm which can further improves timing without more buffering cost. Chapter 4 characterizes a new global routing problem which considers over-the-block routing tracks as well as slew constraints in additional to traditional routing problem. It outlines the relevant prior work in global routing first. Then it describes the importance of over-the-block router followed by a new problem formulation. Next,

Chapter 4 utilizes the physical meaning of Lagrangian multipliers to calculate the real value of each edge in the 3D-routing framework. Finally, it introduces how to use multi-level topology selection algorithm and A* search to discover new topologies. Chapter 5 summarizes the dissertation, and discusses topics for future research.

Chapter 2

Buffering-Aware RSMT Construction for Reclaiming Over-the-IP-Block Routing Resources

2.1 Introduction

As the semiconductor technology scales into deeper sub-micron domain, trillions of transistors and nets can be designed on a single system-on-chip (SOC). Routing becomes more and more challenging because of congestion, power, timing and buffering requirements. Rectilinear Steiner minimum tree (RSMT) construction is a fundamental physical design problem to achieve routing and buffering quality. This classical problem has long been proved as NP-complete [41] and many works have been performed including recent breakthrough, e.g. [18].

Because of extensively using IP-blocks to shorten turn around time, SOC designs nowadays are packed with IP blocks or macros. RSMT construction avoiding these blockages is well known as the OA-RSMT problem. OA-RSMT problem has been studied actively in the last few years (e.g., [6,30,35,36]). Early approaches [35,36] only deal with rectangular blockages, while a most recent study [30] can tackle rectilinear blockages without

dissecting rectilinear blockages into rectangular ones. This approach can eliminate the unfeasible solutions which put wires and buffers between adjoining blocks. However, all these OA-RSMT algorithms simply treat IP blocks as routing blockages, which would significantly waste routing resources over these IP blocks and cause more congestion issues.

In practice, most IP blocks such as SRAMs only use certain lower metal layers. Even if we take into consideration the resources reserved for power/ground and clock routing, there are still considerable amount of routing resources available at higher metal layers over these IP blocks. It leads to over-the-block routing resources waste if we simply treat the IP blocks as routing obstacles, which results in more routing demand elsewhere.

Besides blockage avoidance, other layout constraints are considered in [10, 11, 27, 28, 43, 57]. [10, 27, 43, 57] take timing, buffering, etc., into consideration in their tree construction. But slew constraint is not fully touched upon. It is reported that in reality, *slew mode buffering* is more predominant than timing mode buffering [28, 44]. Only a fraction (roughly $5\% \sim 10\%$) of nets needs to be buffered for delay optimization while for the remaining (roughly $90\% \sim 95\%$) are sufficient with slew mode buffering to meet the slew constraints. [11] extends the work in [10] with slew in consideration. However, the slew constraints are translated as length constraints, which may not guarantee meeting strict slew tolerances. [28] considers slew mode buffering and adopts the blockage avoidance algorithm in [8, 26] to benefit slew. But this approach either puts a Steiner node stationary in block or completely moves it

out of block. This might bring unnecessary wiring detours and high buffering cost.

The blockage avoidance approach in [28] is shown by a 3-pin net example in Fig. 2.1(a). S is the source and A, B are the sinks; moving the Steiner node to right leads to the minimum-cost solution. Fig. 2.1(b) shows the same net if BOB-RSMT is adopted. In this case, BOB-RSMT saves two buffers as well as some detour wire-length because it changes the structure of inside tree more efficiently. Buffer-aware tree construction has advantage over methods of tree construction which are independent of buffering.

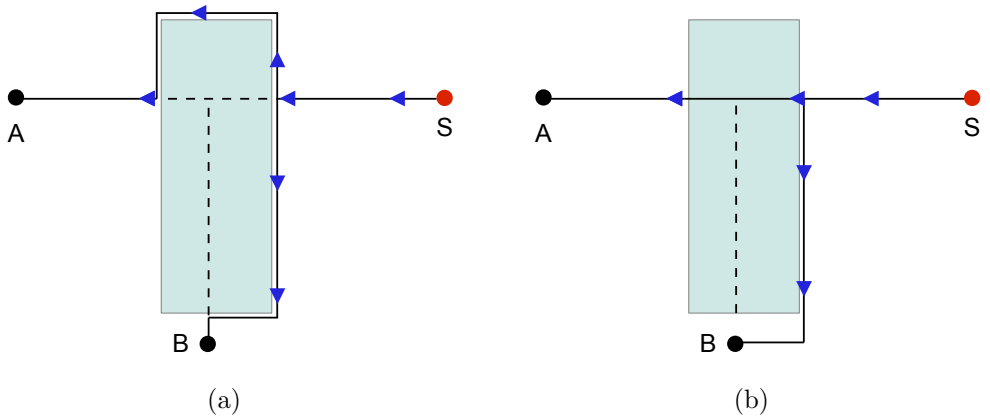


Figure 2.1: A motivational example compares [28] and our proposed BOB-RSMT, which saves wire-length and buffers.

In this chapter, we propose to study a new class of BOB-RSMT problem and develop an effective algorithm which tries to intelligently reclaim the "wasted", over-the-IP-block routing resources by previous approaches while ensuring slew constraints for high quality buffering. Our algorithm incremen-

tally updates the initial RSMT structure obtained from FLUTE [18] to satisfy slew constraints while minimizing wire-length (FLUTE is chosen to be the initial RSMT generator because its low runtime and high quality). A *restricted length, over-the-block maze routing* algorithm is developed to reconnect any part of BOB-RSMT which is dissected during the optimization process. This chapter has the following major contributions:

1. This is the first work targeting this kind of BOB-RSMT problem. Our algorithm is able to integrate with a buffering tool to generate a low buffering cost BOB-RSMT without violating maximum slew constraint, which can be used in floorplanning, placement and routing stages. An incremental approach of fixing slew violation one by one is used to satisfy slew constraints on over-the-block part of BOB-RSMT.

2. Wire-length outside blocks of our BOB-RSMT is remarkably less comparing with that in other algorithms which are not utilizing over-the-block routing resources. This will result in better timing, less power consumption and alleviate routing congestion. The total wire-length, which includes the inside wire-length as well, is also shorter than the results from OA-RSMT algorithms.

3. We formulate the incremental slew improvement problem into an integer linear programming (ILP) problem, which can be solved very fast as the number of variables are small.

4. A block-aware maze router is proposed to reconnect any part of BOB-RSMT dissected during the tree structure optimization.

Our incremental approach of tree structures optimization will be presented in Section 2.2, which includes five subsections. Section 2.3.1 discusses about how to find *possible point set*. Section 2.3.2 gives a method of shrinking search space. Section 2.3.3 formulates and solves the problem. Section 2.3.4 introduces a block-aware maze router to reconnect any dissected part of the tree. Section 2.3.5 describes a buffer insertion algorithm. Experimental results will be shown in Section 2.4, followed by summary in Section 2.5.

2.2 Problem Formulation

2.2.1 What is BOB-RSMT?

BOB-RSMT which utilizes the routing resource over the IP-blocks to improve wire-length and congestion. In a two-dimensional routing region, we are given a net with a set of pins $P = \{p_1, p_2, \ldots, p_n\}$. Let $B = \{b_1, b_2, \ldots, b_m\}$ be a set of non-overlapping rectilinear blocks in the $2-$dimensional space. For $\forall\ p_r \in P$, p_r is not inside the $2-$dimensional space occupied by B. Any area with high-density placed logic cells is not allowed for buffering is also taken as buffering blockage into B.

Our algorithm constructs BOB-RSMT to connect all the pins in P. BOB-RSMT might intersect with blocks in B, which confine a set of trees $T = \{T_1, T_2, \ldots, T_l\}$ inside blocks. We call trees in T *inside trees*. The outside-the-block part of BOB-RSMT is defined as T_0. For each inside tree $T_i \in T$, the

leaf nodes of T_i are on the boundaries of a block. Among all leaf nodes, one must be driving the signal and others are receiving. We name these leaf nodes which receive signals *escaping points* (EP), and the set of escaping points for T_i is $EP^i = \{EP^i_1, EP^i_2, \ldots, EP^i_{|EP^i|}\}$, in which $|EP^i|$ is the number of escaping points in EP^i. We denote the driver by D^i.

2.2.2 Basic Ideas and Optimization Primitives

For any inside tree $T_i \in T$, the worst slew part would occur at escaping points because no buffer is allowed to be inserted over the block. The best that a buffering tool can do to carry signal over the block to escaping points is to put the strongest buffer at D^i and a bunch of smallest buffers at EP^i to shield downstream capacitance. If for any $j, slew^i_j$ is still worse than $slew^i_{spec}$, then the slew from D^i to EP^i_j violates maximum slew constraint, which means that no buffering solution can be generated anyway. Further, because we want to leave more margin for buffering tool at critical timing path and buffer placement aspects, we use a middle size *hypothetical buffer* at D^i and middle size *hypothetical buffers* at EP^i to judge if thus the escaping points have slew violation. Using middle size hypothetical buffers instead of two extreme sizes will weaken the capability of utilizing more over-the-block routing resources, but the former will be a more practical assumption and leads to less buffering cost because more solutions can propagate through this inside tree. If any escaping point EP^i_j driven by a hypothetical buffer has $slew^i_j$ worse than $slew^i_{spec}$, then this EP^i_j is called *illegal escaping point*. Any inside tree with at least one

illegal escaping point is an illegal inside tree.

In order to legalize any illegal inside tree, we will change positions of its escaping points as well as inside Steiner nodes. We move escaping points closer to the driver and then update the positions of corresponding Steiner nodes to improve slew. Fig. 2.2(a) is a three-pin net with source S and sinks A and B. Fig. 2.2(b) is the updated tree after a parallel sliding of escaping point V. Comparing Fig. 2.2(b) to Fig. 2.2(a), the downstream capacitance from W is closer to driver point due to the parallel sliding of V. The less capacitance burden to the driver reduces the slew on both escaping points U and V.

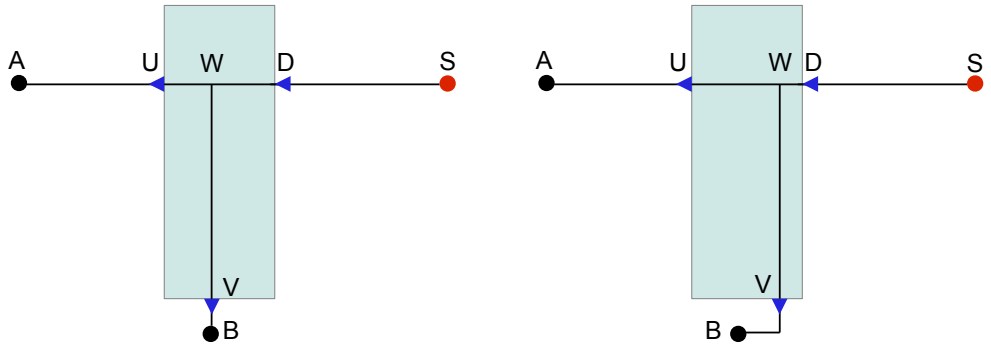


Figure 2.2: V moves to right in (b) compared to (a). This parallel sliding is providing slew improvement for escaping points U and V.

We adopt the following PERI model for slew calculation at the escaping points [34]:

$$S(v_j) = \sqrt{S(v_i)^2 + S_{step}(v_i, v_j)^2} \tag{1}$$

$S(v_j)$ is slew at any node v_j, which is the root-mean square of the *step slew* from v_i to v_j and *output slew* at node v_i. The experimental results in [34] shows the error of PERI is within 1%, which is indistinguishable from what is obtained using SPICE simulation. For simplicity we use Bakoglu's metric [12] for step slew calculation:

$$S_{step}(v_i, v_j) = \alpha * Elmore(v_i, v_j), \alpha = ln9 \tag{2}$$

The combination of Bakoglu's metric and the PERI model is shown to have error within 4% [34]. It is, in general, accurate enough for RSMT construction purpose.

We propose three slew optimization *primitives* including *parallel sliding*, *perpendicular sliding* and *EP merging* to improve the slew. The proposed primitives could guide illegal inside trees to migrate into legal ones with minimum wire-length increase. The analysis demonstrates that the capability of using these three primitives can fix slew violations under any $slew_{spec}$.

We first analyze parallel sliding which performs sliding to a new position on one of the block boundaries. As the escaping point sliding on the boundary, if its first upstream Steiner node ancestor can also slide to keep the wire segment between escaping point and the ancestor Steiner node in translation, then this sliding on the boundary is called parallel sliding. The requirement of a meaningful parallel sliding is that the sliding should shorten the length of path from the escaping point to D^i, i.e., sliding the escaping point closer to the driver.

The example in Fig. 2.3(a) provides an inside tree with the driver D and escaping points A, B, C, E. Fig. 2.3(b) shows that escaping point A performs a parallel sliding by a distance of Δl to new position A'. There will be a reduction of step slew on escaping point A and B. We adopt the following notations in Table 2.1 to calculate the slew improvement of parallel sliding in this example. The step slew reduction on A and B from Bakoglu's metric model will be:

$$\delta_A = -\alpha * r\Delta l(C_t(U) + 0.5 * c\Delta l)$$

$$\delta_B = -\alpha * r * \Delta l * (C_b + c * l(A, U))$$

The output slew of the driver D remains unchanged since the total downstream capacitance of the inside tree is the same. Then we can use (3.3) to calculate the corresponding slew change on escaping point A and B. The changes in slew of escaping point C and E are both zero because U is not on the path from these two escaping points to the driver D.

Table 2.1: Notation of variables

r	unit length wire resistance on chosen layer
c	unit length wire capacitance on chosen layer
R_b	chosen buffer output resistance
C_b	chosen buffer input capacitance
$l(U, V)$	length of edges between node U and V
$C_t(V)$	total capacitance of the sub-tree rooted at node V down to the nearest downstream buffer, including the buffer input capacitance

With parallel sliding we can decrease slew at escaping points, but we

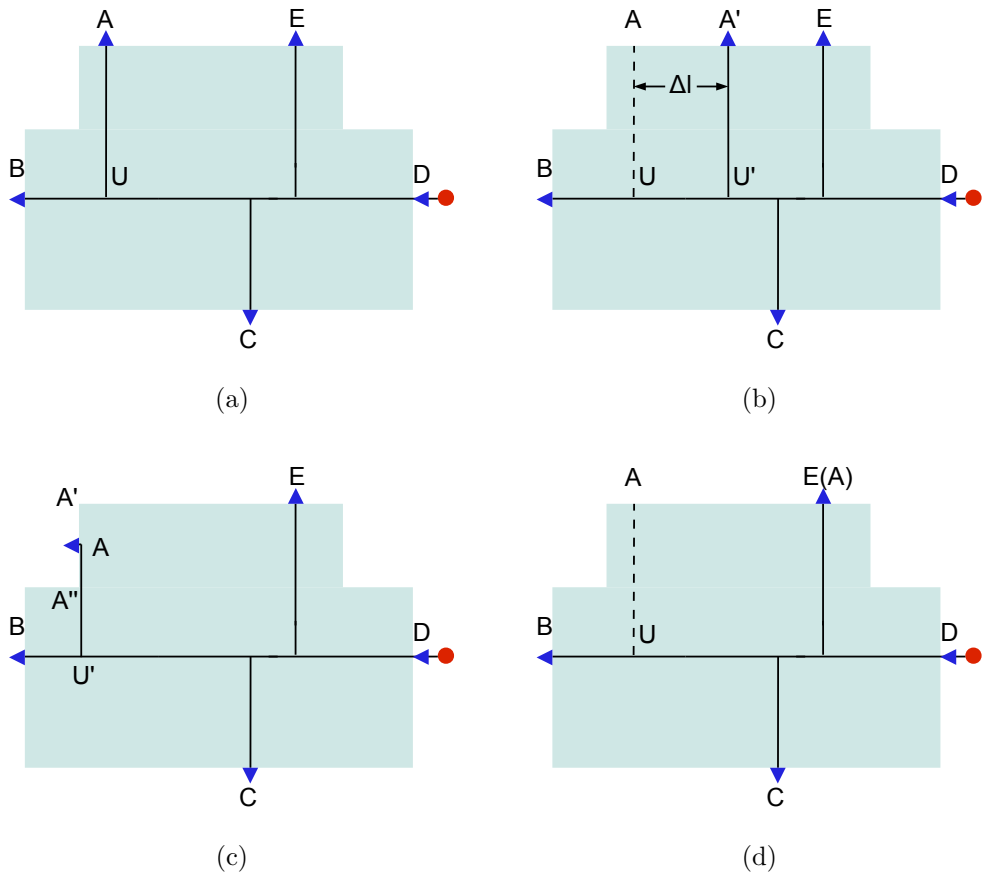


Figure 2.3: An example shows slew reduction from three primitives. (b) shows escaping point A slides to A' parallelly to improve slew on A and B. (c) shows the vertical sliding of A from A' to A''. (d) shows EP merging of escaping point A to E.

may have wire-length penalty because the position change of escaping points may need some additional wire connection from outside-the-block sub-tree. In the example shown in Fig. 2.3(a) to Fig. 2.3(b), escaping point A moves a distance of Δl to A' by a parallel sliding. The penalty of wire-length is at most Δl because the outside connection to A' can go through A along the edge

from A to A' with Δl more wire-length and there is no change in wire-length of inside tree.

Besides parallel sliding, we perform perpendicular sliding on edges which are not considered as parallel sliding edges. In Fig. 2.3(c), if A is sliding on the segment between A' to A'', the wire-length penalty will be zero during the whole sliding process because slide of A from A' to A'' is just slipping wire from inside block to outside. It is observed that as A reaches A'', all escaping points will have the largest slew improvement due to the least downstream capacitance from U'. The calculation of slew reduction is similar as of parallel sliding.

Complementary to parallel sliding and perpendicular sliding, EP merging removes one escaping point and all edges from this escaping point up to the first Steiner point ancestor in the inside tree. This will also bring down the slew of all escaping points based on the fact that this escaping point and the upstreaming edges from it to next Steiner point in the inside tree will be removed. The above process will reduce the total capacitance burden of the driver and hence improve slew for all escaping points.

In tree T_i, if EP merging joins one EP_i with another EP_j, the outside connection to EP_i will be reconnected to EP_j or other closer part of BOB-RSMT by a restricted length, over-the-block maze routing algorithm, which will be introduced in Section 2.3.4.

Considering the EP merging of escaping point A to E in Fig. 2.3(a)

Algorithm 1 *The overall BOB-RSMT Algorithm*

Require: Initial inside trees T, Slew required for the net: $slew_{spec}$
Ensure: BOB-RSMT

```
for each T_t do
    Sort EP^t in descending order of slew
    while slew_1^t > slew_spec do
        Build possible point set for all unfixed EP in EP^t
        Formulate the problem by a ILP
        Solve the ILP and update T_t
        Remove EP_1^t from EP^t
    end while
end for
return BOB-RSMT
```

and Fig. 2.3(d), the wire-length penalty will be at most the distance between A and E because the outside connection to A can go through original position of A and then along the edge to E as shown in Fig. 2.3(d). Actually due to the existence of tree outside this block, reconnecting to the outside part might have less wire-length penalty. But here, we take the previous conservative estimate as the wire-length penalty because it is guaranteed to be achieved. The calculation of slew reduction is similar.

2.3 BOB-RSMT Algorithms

To construct a legal BOB-RSMT, we first generate an initial RSMT by using FLUTE-3.1, and then we apply primitives to all illegal inside trees to fix the slew of them. Finally a proposed restricted length, over-the-block maze routing algorithm is used to reconnect all these parts to form the final BOB-RSMT. The approach is described in Algorithm 1.

For each $T_t \in T$ as an illegal inside tree, three primitives are applied to decrease slew on illegal escaping points until T_t becomes a legal inside tree. The procedure starts from calculating slew of each EP_i^t. From the calculated result, we first sort EP^t in descending order of their slew violations as line 2 of Algorithm 1. Then we choose the first illegal escaping point, EP_1^t, which should have worst slew violation based on the sorting. To improve slew for EP_1^t, each escaping point from $\{EP_1^t, EP_2^t, \ldots EP_{|EP^t|}^t\}$ might slide to a different position by taking a combination of primitives discussed in section 2.2.2. Taking these optimization primitives guarantees $slew_1^t$ to be within slew requirement. Because in the extreme situation where maximum slew constraint is zero EP_1^t can still become legal escaping point by merging one escaping point to another until only the driver is left. This slew fixing procedure is elaborated through line 4 to 6 of Algorithm 1.

After $slew_1^t$ has decreased below the required slew, EP_1^t is fixed at the current position and removed from EP_t as in line 7. Next iteration will start from the rest of EP^t. The current iteration will not degrade the result of previous iterations as we will remove solution space from current solution space if it degrades slew of fixed escaping points. This solution space elimination happens rarely because moving one escaping point closer to driver usually does not degrade slew on other points. This slew improvement method will keep being applied on EP_1^t at each iteration until all EP^t are fixed.

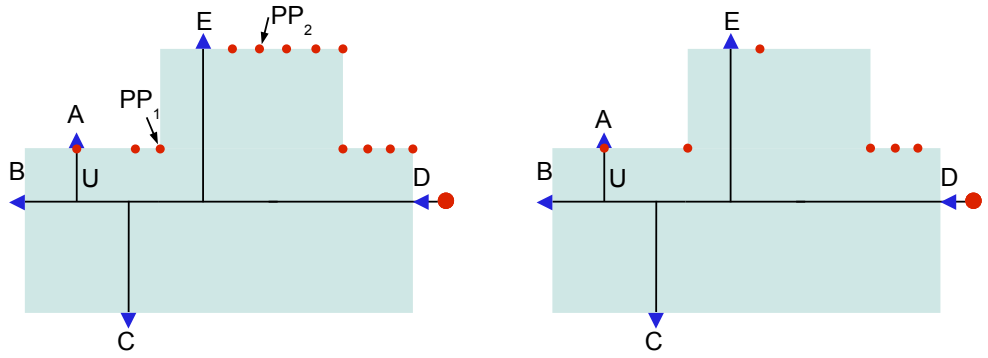


Figure 2.4: (a) is an inside tree with driver at D. It shows all possible points for E. (b) exhibits the refined possible point set for E.

2.3.1 Generating Possible Point Set

At every iteration we try to improve the $slew_1^t$ of inside tree $T_t \in T, t \in \{1, 2, \ldots, l\}$. One important step in that is to generate *possible point set* for each non-fixed escaping points. Possible point set is a set of all *possible points* of one non-fixed escaping point. Each possible point in the set is a point on a boundary edge where escaping point might end up. For any non-fixed $EP_i^t \in \{EP^t\}$, the j^{th} possible point associated with EP_i^t is denoted as PP_{ij}. PP_{ij} is stored in a 3-tuple format $\{E_{ij}, B_{ij}, W_{ij}\}$. E_{ij} and B_{ij} denote the step slew at EP_1^t and output slew reduction of the driver if EP_i moves to PP_{ij}. W_{ij} is the corresponding estimated wire-length penalty. The possible point set associating with EP_i^t in the current iteration is denoted as PPS_i^t. $PPS_i^t = \{PP_{i1}^t, PP_{i2}^t, \ldots, PP_{ir}^t\}$, where r is the number of possible points inside.

For each EP_i^t in current iteration, we generate the possible point for EP merging first. Assume the target escaping point for EP_i^t to merge with is EP_j^t. The estimated wire-length penalty is the outside-the-block distance from EP_i^t to EP_j^t. Thus for EP merging, we always choose the EP_j^t with minimum outside-the-block distance from EP_i^t. The slew reduction and the estimated wire-length penalty of this choice will be added to the PPS_i^t as the 3-tuple $\{E_{ij}, B_{ij}, W_{ij}\}$. For example in Fig.2.4(a), where EP_1^t is B and EP_i^t is E, the EP merging point for E is escaping point A.

Secondly, we consider the sliding for EP_i^t. We first search all edges on $block_t$ for sliding by the criteria discussed in Section2.2.2. The $block_t$ here refers to the block confining T_t. Then for each parallel sliding edge, we chop it at a number of points. Moving EP_i^t to any one of these points can improve slew on EP_1^t. For each perpendicular sliding edge, we pick the possible point at one end of it, as discussed in Section2.2.2. Each chop point is a possible point, which will be added into possible point set. We set distance between two chop points to be a fixed value depending on the scale of the chip. For example in Fig.2.4(b), D is the driver and A, B, C, E are escaping points. The possible point set for E are shown with red color dots.

2.3.2 Refinement of Possible Region Set

For any escaping point EP_i^t, after collecting PPS_i^t, we will do a refinement on $\forall PP_{ij} \in PPS_i^t$ to reduce the potential solution space. The refinement is based on Pareto efficiency [13].

The refined possible point set should form a Pareto frontier in the sense of estimated wire-length penalty and slew reduction(both output slew reduction at the driver and step slew reduction at EP_1^t), which is restricting attention to the set of choices that either has less estimated wire-length penalty or more slew reduction. After applying refinement on Fig.2.4(a), the possible points turn into Fig.2.4(b). One example of a pruned possible point in Fig.2.4(a) is: PP_2 is pruned by PP_1 as the latter has less estimated wire-length penalty and more slew improvement.

2.3.3 Primitive Choice Based on a Fast ILP

In order to construct the inside tree under the slew constraint with minimum wire-length as target, $\forall EP_i^t \in EP^t$ we need to decide which possible point to choose. We use an incremental way to update positions of all escaping points at each iteration. In each iteration, in order to meet the slew constraint for the worst violated escaping point, all escaping points in EP^t will move and the whole inside tree will be updated. Only through moving all $EP_i^t \in EP^t$ at the same time can we attain an optimal solution with minimum estimated wire-length penalty. This stems from the reason that $\forall PPS_i^t, i \in \{1, 2, \ldots, |EP_i^t|\}$ has a Pareto frontier to choose one point from. The choice depends on what choices are made at other Pareto frontiers because the total slew reduction summed up from all these choices has to diminish the slew violation of EP_1^t.

The new slew has to satisfy the slew constraints,

$$\sqrt{(S^t_{step1} + \sum_{i=1}^{|EP^t_i|} E^t_i)^2 + (S^t(D^t) + \sum_{i=1}^{|EP^t_i|} B^t_i)^2} < slew^t_{spec}$$

The simultaneous step slew reduction is same with calculating one by one, and the simultaneous output slew reduction is close enough to be represented by the summation of individuals.

This simultaneous selection problem is exact knapsack problem: Given a set of possible points, each with a slew improvement and a wire-length penalty, determine possible point to move to so that the total slew improvement is more than or equal to a given limit and the total wire-length penalty is as small as possible [2]. We use maximum number of possible points on one edge to limit the maximum number of candidates from one EP. Then the total number of candidates for inside-tree t is $O(n)$ where n is the number of EPs. Moreover, the number of EPs on one block for each net equals to the number of interactions between one topology and one block. Since this decision problem is NP-complete and the problem size for each block is $O(n)$, we can apply ILP to solve it. The simultaneous point choice problem can be formulated in an optimization problem as follows (notation in Table 2.2):

Table 2.2: Notation of variables in our formulation

X_{ij}	binary variable denoting the choice of PPS_{ij}^t, $X_{ij} = 1$ if it is chosen, otherwise $X_{ij} = 0$
E_{ij}	step slew reduction at EP_1^t if EP_i^t moves to PPS_{ij}
B_{ij}	output slew reduction on D^t if EP_i^t moves to PPS_{ij}
W_{ij}	estimated wire-length penalty of EP_1^t if EP_i^t moves to PPS_{ij}
Y_{rsij}	binary variable equals to one only if $X_{rs} = 1$ and $X_{ij} = 1$

$$\text{min.} \sum_{i=1}^{|EP^t|} \sum_{j=1}^{|PPS_i^t|} X_{ij} W_{ij} \tag{3}$$

$$\text{s.t.}(S_{step1}^t + \sum_{i=1}^{|EP^t|} \sum_{j=1}^{|PPS_i^t|} X_{ij} E_{ij}^t)^2 +$$

$$(S^t(D^t) + \sum_{i=1}^{|EP^t|} \sum_{j=1}^{|PPS_i^t|} X_{ij} B_{ij}^t)^2 <= {slew_{spec}^t}^2 \tag{3a}$$

$$\sum_{j=1}^{|PPS_i^t|} X_{ij} = 1 \qquad \forall i \in \{1, 2, \ldots, |EP^t|\} \tag{3b}$$

The objective function (5) is to minimize the total estimated wire-length penalty. Constraint ($3a$) restricts that the total slew reduction on EP_1^t has to be able to pull $slew_1^t$ down below requirement. Constraint ($3b$) is used to limit only one position chosen for each escaping point.

This formulation is a non-linear integer programming formulation (NLIP).

We expand the step slew part in constraint $(3a)$ as:

$$S_{step1}^{t}{}^{2} + 2S_{step1}^{t} \sum_{i=1}^{|EP^t|} \sum_{j=1}^{|PPS_i^t|} (X_{ij}^t)(E_{ij}^t) + \sum_{r=1}^{|EP^t|} \sum_{s=1}^{|PPS_i^t|} \sum_{i=1}^{|EP^t|} \sum_{j=1}^{|PPS_i^t|} X_{rs}^t E_{rs}^t X_{ij}^t E_{ij}^t$$

We observe that the only quadratic item is $X_{rs}^t X_{ij}^t$. We can substitute this item for a new binary variable Y_{rsij}. We constrain Y_{rsij} such that Y_{rsij} always behaves same as $X_{rs}^t X_{ij}^t$. The constraint needed is (for output slew part, it is similar):

$$Y_{rsij} <= X_{rs}^t \qquad \forall r, s, i, j \in \{1, 2, \ldots, |EP^t|\} \tag{3c}$$

$$Y_{rsij} <= X_{ij}^t \qquad \forall r, s, i, j \in \{1, 2, \ldots, |EP^t|\} \tag{3d}$$

$$Y_{rsij} >= X_{rs}^t + X_{ij}^t - 1 \ \forall r, s, i, j \in \{1, 2, \ldots, |EP^t|\} \tag{3e}$$

By adding constraint $(3c) \sim (3e)$ to (5), we turn the NLIP problem into integer linear programming formulation (ILP), which can be solved by solver Gurobi Optimizer [1] quickly. The formulation of ILP is shown as follows:

$$\text{min.} \sum_{i=1}^{|EP^t|} \sum_{j=1}^{|PPS_i^t|} X_{ij} W_{ij}$$

$$\text{s.t.} {S_{step1}^t}^2 + S^t (D^t)^2 + 2 S_{step1}^t \sum_{i=1}^{|EP^t|} \sum_{j=1}^{|PPS_i^t|} (X_{ij}^t)(E_{ij}^t) +$$

$$2 S^t (D^t) \sum_{i=1}^{|EP^t|} \sum_{j=1}^{|PPS_i^t|} (X_{ij}^t)(B_{ij}^t) +$$

$$\sum_{r=1}^{|EP^t|} \sum_{s=1}^{|PPS_i^t|} \sum_{i=1}^{|EP^t|} \sum_{j=1}^{|PPS_i^t|} (B_{rs}^t B_{ij}^t + E_{rs}^t E_{ij}^t) Y_{rsij} <= {slew_{spec}^t}^2$$

$$\sum_{j=1}^{|PPS_i^t|} X_{ij} = 1 \qquad \forall i \in \{1, 2, \ldots, |EP^t|\}$$

$$Y_{rsij} <= X_{rs}^t \qquad \forall r, s, i, j \in \{1, 2, \ldots, |EP^t|\}$$

$$Y_{rsij} <= X_{ij}^t \qquad \forall r, s, i, j \in \{1, 2, \ldots, |EP^t|\}$$

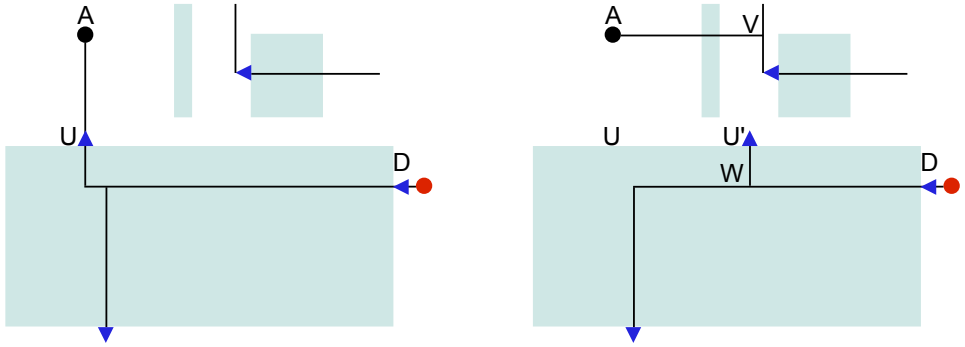
$$Y_{rsij} >= X_{rs}^t + X_{ij}^t - 1 \; \forall r, s, i, j \in \{1, 2, \ldots, |EP^t|\}$$

Due to the number of choices for each escaping point is limited by the number of possible sliding edges and their length, the total number of variables in our formulation is very limited. The ILP solver can get the solution very fast.

2.3.4 Block-aware Maze Routing Algorithm

After final positions of all escaping points are fixed, a restricted length, over-the-block maze routing will be applied. This maze routing features ability of routing over-the-blockage. The maximum length it can route over the block is decided by the distance a middle size buffer could drive itself over the block without slew problem. This restricted length, over-the-block maze

router requires less wire-length comparing with normal maze router because of its ability to route over the block. In Fig. 2.5(a), U is an escaping point and A is a sink of the tree. Escaping point U slides to U' to legalize the inside tree. The restricted length, over-the-block maze is applied to reconnect A, and it will choose connection from A to V instead of from A to U' because of shorter wire-length, resulting in Fig. 2.5(b). Wire segment U' to W will be removed if no other part connects to U'.



(a) (b)

Figure 2.5: Restricted length, over-the-block maze routing find a shortest path to reconnect pin A

The implementation of block-aware maze routing is based on the normal maze routing. But its multiple points to multiple points search is from all points of the tree rooted at the current escaping point to T_0 or an escaping point of any inside tree. Furthermore, the length of the over-the-block path is checked every step in the search to make slew safe. The details of the algorithm are skipped here due to page limit.

2.3.5 Min-cost Slew Mode Buffer Insertion

After BOB-RSMT is fully constructed, we insert buffers in a free-location way, which allows buffers at any unblocked space. Comparing with fixed-location buffer insertion algorithm, free-location buffering can freely choose position for buffering, which will result in lower buffer cost. We assume the input slew of each buffer is fixed at the slew constraint. Free-location buffering with fixed input slew will give a shorter runtime but conservative result [28]. It uses a dynamic programming framework to propagate a set solutions from bottom up to the source of the net. Each solution is characterized as a triple (C, W, S), where C stands for downstream capacitance, W denotes the total cost of the solution, and S is the worst downstream accumulated step slew degradation calculated from (3.4). Consider to propagate a solution from a node v_j to its parent v_i through edge $e = (v_i, v_j)$. One solution γ_j at node v_j propagates to v_i to become a solution γ_i as $C(\gamma_i) = C(\gamma_j) + C_e, W(\gamma_i) = W(\gamma_j), S(\gamma_i) = S(\gamma_j) + S_{step}(v_i, v_j)$.

In addition to unbuffered propagation, a buffer can be placed at v_i to generate a buffered solution. If a buffer is placed, the buffered solution at v_i is becoming $C(\gamma_{i,buf}) = C_b, W(\gamma_{i,buf}) = W(\gamma_i) + W_b, S(\gamma_{i,buf}) = 0$.

When two sets of solutions propagated by both left and right children reach a branching node, these two set of solutions are merged. The merge is performed on each solution in left child with each solution in right child. Assume γ_l is one solution from left side and γ_r is one solution from right side to be merged. The merged solution γ_p will have $C(\gamma_p) = C(\gamma_l) + C(\gamma_l), W(\gamma_p) =$

$W(\gamma_l) + W(\gamma_r), S(\gamma_p) = \max\{S(\gamma_l), S(\gamma_r)\}$.

It is beneficial to prune useless solution at each node. As two solutions γ_{i_1} and γ_{i_2} are at same node, γ_{i_1} dominates γ_{i_2} only if $C(\gamma_{i_1}) <= C(\gamma_{i_2}), W(\gamma_{i_1}) <= W(\gamma_{i_2}), S(\gamma_{i_1}) <= S(\gamma_{i_2})$.

2.4 Experimental Results

We have implemented our algorithm in the C++ programming language. The experiments are conducted on an Intel Core 3.0GHz Linux machine with 32GB memory. We choose Gurobi Optimizer 4.60 as our solver for the integer linear programming.

RT1-RT5 and RC01-RC11 are benchmarks in our experiments. IND1-IND5 used in [6, 30] are not used in our experiments, because they require routing/buffering between adjoining blocks, which might be unfeasible for real designs. RT1-RT5 are randomly generated circuits used in [37]. RC01-RC11 are test cases used in [24]. Because these benchmarks are widely different in scale and do not carry timing and physical information, we first apply predetermined resistance and capacitance to all of them. We use different resistance and capacitance for horizontal and vertical wires respectively. If a congestion map is considered, we can assign each wire segment to a proper layer by pruning possible points in congestion.

For each benchmark, after FLUTE-3.1 finishes generating inside trees, we collect slew on every escaping point for all inside trees. The range of value

of collected slew is $[slew_{min}, slew_{max}]$. Then we test each benchmark under three slew constraints:

1. 20% slew: $slew_{min} + 20\%(slew_{max} - slew_{min})$
2. 50% slew: $slew_{min} + 50\%(slew_{max} - slew_{min})$
3. 80% slew: $slew_{min} + 80\%(slew_{max} - slew_{min})$.

These three tests of each benchmark can test the performance of our algorithm under tight, medium and loose slew constraints, respectively.

Table 2.5 compares the performance of our algorithm with some recently published OA-RSMT algorithms. Columns 4, 5, 6 list the over-the-block wire-length, outside-the-block wire-length and total wire-length of our algorithm under 20% slew constraint. Columns 7 to 12 are for same types of wire-length under 50% and 80% slew constraints. The row at bottom illustrates the average performance from all benchmarks listed above. We normalize the performance in such a way that the total wire-length of FLUTE-3.1 is 100. The outside-the-block wire-length from Huang [30] is 14.27% more than our BOB-RSMT algorithm under 20% slew constraint and 17.29% under 80% slew constraint. The free over-the-block routing resources reclaimed by BOB-RSMT are between 10% to 12% of the total wire-length. Even for the total wire-length, since BOB-RSMT can intelligently use over-the-block wires, it can reduce about 5% of total wire-length compared with [30] and [6]. The runtime of our proposed algorithm BOB-RSMT is divided into two parts:

solving ILP and block-aware maze routing, which are listed in the columns 2, 3, 6, 7, 10, 11 in Table 2.3. Our runtime is much shorter than both reported in [30] and [6].

Table 2.4 carries out the buffering results on *FLUTE*, approach in [28] and *BOB-RSMT*. For simplicity we only use one type of buffer, and total buffering cost is the number of buffers used. From the table we have minimum buffering cost associated with $20\%, 50\%, 80\%$ slew constraint respectively for all benchmarks. We use buffering on FLUTE as the baseline for our comparison. Buffering on FLUTE is performed without considering any block in the two-dimensional routing region. We implement the approach in [28] and the results are in columns 3, 8, 13 in the Table 2.4. The penalty parameter α for over-the-block routing wires in [8, 28] is set between 10 to 100, which increases if no solution can propagate to the source. Columns 5, 10, 15 in the Table 2.4 are the minimum buffering costs from BOB-RSMT. Columns after the buffering cost are the percentages of extra buffers used to overcome blocks by that approach. As we can see, buffering on BOB-RSMT only uses around 3% more buffers than FLUTE to propagate through blocks, while the approach in [28] uses more than 20%. The CPU runtime comparison between buffering on BOB-RSMT and the approach in [28] is in Table 2.3. Columns 5, 9, 13 illustrate the runtime for buffering on BOB-RSMT under three slew constraints while columns 4, 8, 12 are for approach in [28]. Buffering on BOB-RSMT is much faster because during the buffering stage, the tree structure of BOB-RSMT has no need to be changed to meet slew constraint, but in

contrast [28] needs to find LeastBlockedPath($[8, 26]$) during every step.

2.5 Summary

In this chapter, we study an important new class of RSMT problems, i.e., buffering-aware over-the-IP-block rectilinear Steiner minimum tree. We propose an effective and efficient algorithm which can reclaim the over-the-IP-block routing resources and is beneficial to buffering. With our proposed approach, we can reduce the outside-the-block wire-length for more than 14% and use about 19% less buffer cost than the approach in [28] to ensure slew correct RSMT with blocks. Our proposed algorithm BOB-RSMT can be used in both pre-routing and global routing stage to provide high quality routing solutions. One example is to pre-route certain persistent critical signals in large complex chips, such as a microprocessor, using higher metal layers. Since this is the first work of this kind, we expect more follow-up works to push the state-of-the-art of BOB-RSMT, which is crucial for large SOC designs with many IP-blocks.

Table 2.3: CPU runtime

Bench -marks	CPU (s)											
	20% slew				50% slew				80% slew			
	ILP	maze routing	C-SB buffering	BOB-RSMT buffering	ILP	maze routing	C-SB buffering	BOB-RSMT buffering	ILP	maze routing	C-SB buffering	BOB-RSMT buffering
RT1	0.02	0.1	270.09	0.03	0	0.04	281.62	0.02	0	0.04	285.03	0.01
RT2	0	0.13	1041.83	0.03	0	0.04	1056.27	0.07	0.01	0.1	1059.6	0.04
RT3	0.02	0.13	905.05	0.26	0	0.07	1023.39	0.19	0	0.07	1041.01	0.15
RT4	0.02	0.25	2859.06	0.64	0	0.07	2880.2	0.47	0	0.07	2896.48	0.43
RT5	0.03	2.61	> 7200	1.25	0.01	1.43	> 7200	1.03	0.01	0.35	> 7200	1.03
RC1	0	0.01	0.05	0	0	0.01	0.05	0	0	0	0.05	0
RC2	0	0	0.45	0.01	0	0	0.53	0	0	0.01	0.63	0
RC3	0	0.02	0.50	0.01	0	0.03	0.66	0.01	0	0.01	0.63	0.01
RC4	0.01	0.04	2.72	0.01	0	0.01	2.40	0.01	0.02	0	2.40	0.01
RC5	0	0.01	4.44	0.02	0.01	0.01	4.44	0.02	0.01	0.01	4.50	0.02
RC6	0.04	0.91	1652.4	0.2	0.01	0.08	1643.11	0.12	0	0.09	1634.05	0.13
RC7	0.04	3.43	> 7200	0.36	0.02	1.51	> 7200	0.29	0	0.56	> 7200	0.24
RC8	0.05	2.24	> 7200	0.05	0.01	0.76	> 7200	0.56	0.01	0.26	> 7200	0.55
RC9	0.09	5.08	> 7200	0.78	0.03	1.88	> 7200	0.55	0.02	1.02	> 7200	0.4
RC10	0.02	0.31	190.1	0.44	0	0.06	191.3	0.39	0	0.05	190.39	0.38
RC11	0.04	0.55	592.35	1.24	0.01	0.46	589.87	1.15	0.01	0.37	591.71	1.15

Table 2.4: Extra buffering cost comparison

Bench	20% slew					50% slew					80% slew				
-marks	FLU -TE	[28]	extra % of [28]	BOB -RSMT	extra % of BOB-RSMT	FLU -TE	[28]	extra % of [28]	BOB -RSMT	extra % of BOB-RSMT	FLU -TE	[28]	extra % of [28]	BOB -RSMT	extra % of BOB-RSMT
RT1	151	221	46.36	158	4.63	94	137	45.74	98	4.26	74	109	33.78	77	4.05
RT2	349	409	17.19	352	0.86	218	225	3.21	218	0.00	16	19	18.75	16	0.00
RT3	761	897	17.87	767	0.79	470	557	18.51	473	0.64	376	442	17.55	376	0.00
RT4	300	448	49.33	306	2.00	180	270	50.00	184	2.22	139	208	49.64	144	3.60
RT5	378	673	78.04	386	2.12	228	413	81.14	237	3.95	174	326	87.36	183	5.17
RC1	21	23	9.52	22	4.76	13	14	7.70	14	7.70	10	11	10.00	10	0.00
RC2	33	37	12.12	33	0.00	21	24	14.29	21	0.00	16	19	18.75	16	0.00
RC3	96	113	17.71	99	3.12	59	71	20.34	60	1.69	45	54	20.00	47	4.44
RC4	65	76	16.92	67	3.08	41	48	17.07	41	0.00	30	36	20.00	32	6.67
RC5	62	70	12.90	62	0.00	36	41	13.89	39	8.33	28	32	14.29	29	3.57
RC6	237	272	14.77	248	4.64	143	166	16.08	151	5.59	112	128	14.29	118	5.36
RC7	458	504	10.04	465	1.53	278	307	7.67	287	3.24	217	242	11.52	224	3.23
RC8	282	342	21.28	304	7.80	172	211	22.67	187	8.72	135	162	20.00	143	5.92
RC9	425	506	19.06	451	6.12	259	313	20.85	278	7.34	202	247	22.28	213	5.45
RC10	394	412	5.08	395	0.25	246	261	6.10	248	0.81	189	201	6.35	191	1.06
RC11	1662	1695	1.99	1670	0.48	1023	1044	2.05	1025	0.20	789	804	1.90	792	0.38
Ave.			21.89		2.64			21.71		3.42			22.90		3.06

Table 2.5: Comparisons between our proposed BOB-RSMT and OA-RSMT

Bench -marks	n	m	20% slew			50% slew			80% slew			FLUTE -3.1	Huang [30]	Ajwani [6]
			WL_i	WL_o	WL	WL_i	WL_o	WL	WL_i	WL_o	WL			
RT1	10	500	385	1449	1834	296	1522	1818	296	1522	1818	1817	2146	2191
RT2	50	500	1216	43469	44685	1216	43469	44685	1186	43507	44693	44685	45852	48156
RT3	100	500	263	7420	7683	276	7390	7666	282	7379	7661	7652	7964	8282
RT4	100	1000	1196	6647	7843	872	6957	7829	882	6947	7829	7827	9693	10330
RT5	200	2000	6702	36474	43176	7277	35720	42997	7491	35458	42949	42943	51313	54598
RC1	10	10	740	24550	25290	740	24550	25290	740	24550	25290	25290	25980	25980
RC2	30	10	5220	36998	42218	8190	34520	42710	8190	33020	41210	39920	41350	42110
RC3	50	10	530	53950	54480	1190	53290	54480	4480	48430	52910	52880	54160	56030
RC4	70	10	3030	52420	55450	4490	50960	55450	5420	50027	55447	55300	59070	59720
RC5	100	10	3590	69810	73400	3590	69810	73400	4750	68980	73730	73220	74070	75000
RC6	100	500	12983	65667	78650	14613	61980	76593	15049	62432	77481	77171	79714	81229
RC7	200	500	13141	97109	110250	13785	95162	108947	14244	93565	107809	106743	108740	110764
RC8	200	800	23674	88136	111810	25515	84049	109564	25184	83385	108569	108495	112564	116047
RC9	200	1000	25689	83972	109661	25689	83972	109661	26026	82192	108218	107729	111005	115593
RC10	500	100	8372	156348	164720	9400	155370	164770	9400	155370	164770	163980	164150	168280
RC11	1000	100	3016	229519	232535	3498	228232	231730	3498	228282	231780	231730	230837	234416
Ave.			9.95	91.38	101.33	10.99	89.98	100.97	11.69	88.74	100.43	100	106.03	108.17

Chapter 3

Timing-Driven, Over-the-Block RST Construction

3.1 Introduction

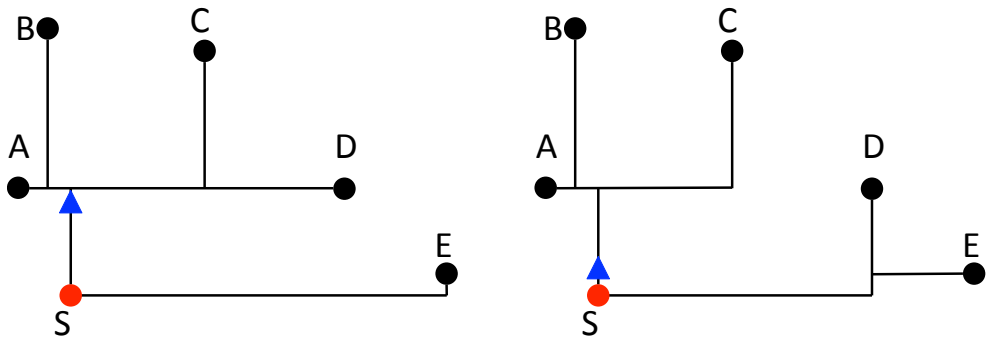
Chapter 2 discusses RSMT and related extensions, which produces good results regarding wire-length minimization, but they are not timing-optimal in deep sub-micron high-speed ICs. To help meet timing on critical paths, timing-driven RST is needed to optimize pin-to-pin delays on those paths. Approaches such as [20,21,31,46,53,58], focus on the minimum delay routing tree (MDRT) problem which minimizes a linear combination of delays at sinks. Other approaches(e.g. [9,16,32]) are able to optimize the required arrival time at the driver as a more practical target. Besides, timing optimization and obstacle avoidance are simultaneously considered in [38], etc. However, most of the above-mentioned timing-driven approaches have the following three problems:

1) In order to build an RST optimizing required arrival time at the driver, it is necessary to know the criticality at all sinks. The first problem is that most previous works (such as [9,38,42]) use simple estimation on arrival time and criticality for each sink, which is not accurate enough. For example

in [9], an optimally buffered 2-pin direct connection from root to one node is used to estimate the potential delay; similarly in [42], the require arrival time is calculated based on distance from root to merging point, neglecting the coupling from other part of the tree. Estimation cannot fully capture interconnection delay, including delay on wires and buffers, decoupling effect by buffers and load capacitance from un-buffered branch, which would result in a sub-optimal timing-driven RST. One the other hand, a buffered tree with topology close enough to the final constructed tree could provide criticality at all sinks accurately. We propose a pre-buffering approach in place of estimation so as to provide more accurate timing information. During pre-buffering, a timing-driven RST is iteratively built and buffered to offer criticality information for the next generation of timing-driven RST until the tree topology converges.

As is shown in Fig.3.1, if only estimation is used, it would conclude that sink E is critical, resulting in the topology in Fig.3.1(a). However, if we insert buffers on the topology in Fig.3.1(a) and re-calculate criticality, we will find that sink D is as critical as E. Based on that finding, the new topology would re-clusters D with E with a direct connection to root S. Upon this new topology, a new buffer insertion is applied to re-calculate criticality at each sink. In this example, we find the set of critical sinks is not changed anymore and thus the topology converges to Fig.3.1(b) which has a better WNS since the slack on D is improved.

2) From [55] and [29], it has been demonstrated that over-the-block

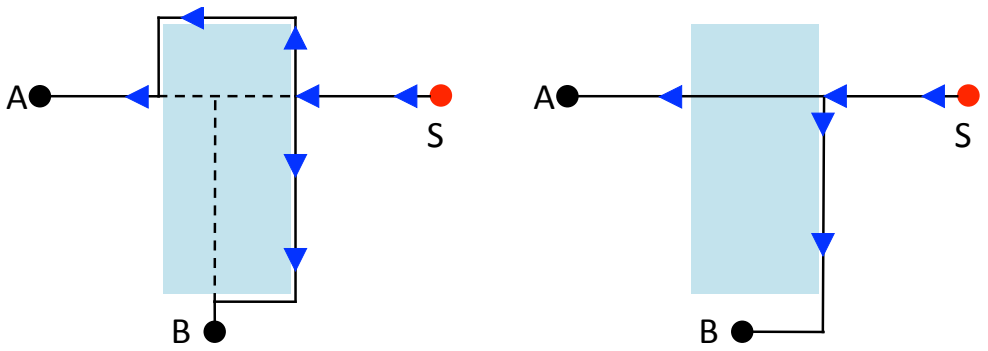


(a) (b)

Figure 3.1: (a) estimates only sink E is critical. (b) groups sink E and D as critical cluster.

RSMT (OB-RSMT) outperforms OA-RSMT in terms of wire-length. Over-the-block routing resources should be used in timing-driven RST construction as well to replace obstacle-avoiding detours with shorter over-the-block connection. In the meantime, certain slew constraints have to be satisfied for over-the-block routing to ensure the solution will not fail buffering. Fig. 3.2 compares obstacle-avoiding tree construction with over-the-block algorithm, in which the latter shifts part of the inside tree outside and keeps the remaining inside the block. As is shown in Fig.3.2(b), the algorithm reduces two buffers, some detouring wire-length and delay of paths in the tree.

3) Following topology generation and buffering, it has never been discovered or discussed that a buffer-location-based tuning can achieve considerable timing improvement without consuming additional buffering cost and noticeable wire-length. During the buffering, in order to obtain a legal buffer-



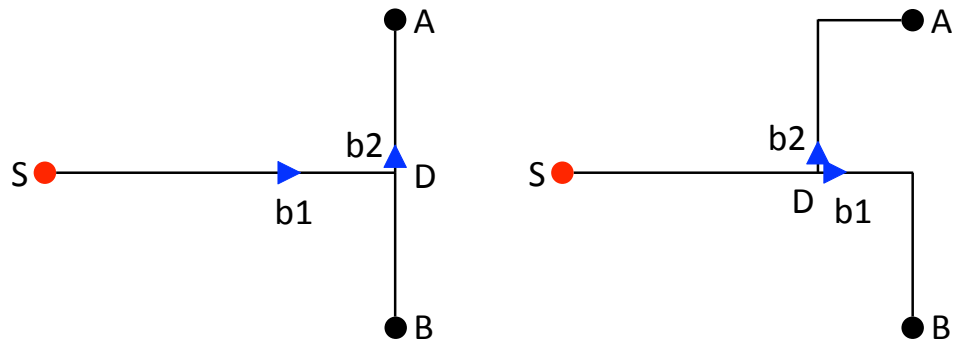
(a) (b)

Figure 3.2: (a) is an OA-RSMT with root S and two sinks A, B. (b) uses part of the over-the-block routing resources.

ing solution, some buffers are placed at positions without fully using up their power. The proposed post-buffering tuning algorithm could tune the locations of Steiner points based on the buffering information to further improve slack. In Fig. 3.3(a), we observe that buffer $b2$ is clamped under the Steiner point D to shield part of the downstream capacitance of D. We can change the position of the Steiner point (Fig. 3.3(b)) which makes the sequential buffers $b1$ and $b2$ parallel. The delay of the path from root S to A is notably reduced since the path becomes a decoupled direct connection and delay on buffer b_1 is taken away. However in a traditional flow, it is hard to accurately predict these better buffer locations via only topology generation and buffering.

This chapter makes the following major contributions:

1. We first propose a timing-driven, over-the-block RST construction algorithm which utilizes over-the-block routing tracks to reduce delay to



(a) (b)

Figure 3.3: (a) is a buffered RST with root S and two sinks A, B. (b) exhibits the tuned topology and new buffering.

critical sinks and shorten wire-length to non-critical sinks.

2. Our constructed RST satisfies the slew constraints everywhere with buffers placed at empty space.

3. During the tree construction, we use pre-buffering scheme to provide more accurate timing information, which helps explore better topologies for timing-driven RST.

4. We analyze the final buffered tree and relocate certain Steiner points to further improve the delay on paths to critical sinks.

5. We conduct our algorithm and observe significant improvements in WS, wire-length and buffering cost compared with existing works.

The rest of chapter is organized as follow. We first introduce basic concepts and our problem formulation in Section 3.2. Our timing-driven, over

the-block RST construction algorithm will be presented in Section 3.3, which includes three subsections. Section 3.3.1 discusses how to use pre-buffering to guide the tree construction. Section 3.3.2 discusses how to use over-the-block routing resources to reduce delays on critical paths without violating slew constraints. modify BOB-RSMT to ensure slew for over-the-block part. Section 3.3.3 introduces the post-buffering topology tuning algorithm which achieves considerable timing improvement without consuming noticeable wirelength and buffering cost. Experimental results will be shown in Section 3.4, followed by summary in Section 3.5.

3.2 Notations and Problem Formulation

In a two-dimensional routing region, we are given a net $N = \{s_0, s_1, s_2, \ldots, s_n\}$ with $n + 1$ pins, where s_0 is the unique source and the rest are sinks. $L = \{b_1, b_2, \ldots, b_m\}$ is a set of non-overlapping rectilinear blocks in a two-dimensional space R. For $\forall\ s_i \in N$, s_i is not inside the two-dimensional space occupied by L. Any area with high-density logic cells not allowed for buffering is also taken as buffering blockage into L.

Our algorithm constructs a timing-driven buffered tree $T(V, E)$ to connect all the pins in N, where V is the set of nodes and E is the set of horizontal and vertical edges. T might intersect with blocks in L, which confines a set of trees $S = \{T_1, T_2, \ldots, T_l\}$ inside blocks. We call trees in S *inside trees*. The outside-the-block part of T is defined as T_0. The buffered tree $T_b(V_b, E_b)$ is generated from T after we insert a set of nodes V' which corresponds to the

buffers chosen from buffer library B, and $V_b = V \cup V'$.

The Steiner tree has a unique path $P(s_0, s_i)$ from s_0 to each sink s_i. The presence of buffers along the path could separate the path into *stages*, each of which consists of a driver, a set of driven nodes as well as edges connecting the driver and the driven nodes. The total delay on a path is the summation of the delay on each stage along that path, which can be computed in many ways. As in this discussion, we adopt the Elmore model for wires and a switch-level linear model for gates. The models we adopt are simple and informative enough to guide our approach, yet our formulation is by no means restricted to these models. The delay of each stage in the path is expressed as:

$$t(d(u), u) = \sum_{e=(i,j) \in p(d(u),u)} r_e l_e (0.5 c_e l_e + C_u(j)) + R_b C_d(d(u)) + D_b \tag{3.1}$$

Total delay of the path is the summation over all stages in the path:

$$d(s_0, s_i) = \sum_{u \in V' \cap p(s_0, s_i)} t(d(u), u) \tag{3.2}$$

The slack of sink s_i is defined as $slack(s_i) = RAT(s_i) - d(s_0, s_i)$. WS is defined as $WS(T) = min\{slack(s_i) | 1 \leq i \leq n\}$, and the worst negative slack is determined by $WNS(T) = min\{0, WS\}$. Notations amongst the formulation are as follows:

- l_e = length of edge e,
- r_e = unit length wire resistance on a chosen layer for edge e,

- c_e = unit length wire capacitance on a chosen layer for edge e,
- R_b = chosen buffer or source output resistance,
- C_b = chosen buffer or source input capacitance,
- D_b = internal buffer or source delay,
- $d(u)$ = the driver of node u,
- $t(u, v)$ = delay from node u to node v,
- $C_d(v)$ = total capacitance of the sub-tree rooted at node v down to the nearest downstream buffer or sinks, including the sink or buffer input capacitance,
- $C_u(v) = C_d(v)$ if v is not a buffer or source; C_b if v is a buffer or source node,

For slew calculation, we adopt the PERI model [34]:

$$S(v_j) = \sqrt{S(v_i)^2 + S_{step}(v_i, v_j)^2} \tag{3.3}$$

$S(v_j)$ is the slew at any node v_j, calculated as the root-mean square of the *step slew* from v_i to v_j and *output slew* at node v_i. The output slew at v_i is described by a 2-D lookup table of input slew and load capacitance. The experimental results in [34] show the error of PERI is within 1%, which is indistinguishable from what is obtained using SPICE simulation. For simplicity,

we use Bakoglu's metric [12] for step slew calculation:

$$S_{step}(v_i, v_j) = \alpha * Elmore(v_i, v_j), \alpha = ln9 \tag{3.4}$$

The combination of Bakoglu's metric and the PERI model is shown to have error within 4% [34], which is, in general, accurate enough for RST construction purpose.

Our algorithm will construct a buffered RST T to connect all sinks and root while ensuring the slew rate on every point in the tree is within constraints. We use *slew mode* buffering as our buffering scheme as it is more predominantly used ([28, 44]) and saves buffering cost. The slew mode buffering satisfies the slew constraints on every point of the buffered tree with minimum buffering cost. Our buffered tree will have edges over the blocks but no buffers are allowed over the blocks. The object is to minimize the WNS of the tree with the lowest buffering cost.

3.3 Timing-driven Over-the-block RST

Our approach constructs a timing-driven, over the-block RST with slew constraints. First, the approach uses coupled buffering and topology generation to provide AT and criticality at each sink. Then, a timing-driven RST is constructed based on pre-buffering. Second, the topologies of over-the-block trees are optimized to meet the slew constraints while maintaining the delay to critical sinks. Then, buffering is performed on the constructed tree structure. Finally, the constructed tree is tuned based on buffering information followed

by buffering again. The overall algorithm of proposed approach is illustrated in Algorithm2.

Algorithm 2 *The overall algorithm*

Require: Set of pins N and blocks L
Ensure: Timing-driven over-the-block RST T
1: Construct timing-driven initial RST T with pre-buffering
2: Change the topology of T to meet the slew constraints
3: Perform buffering on T
4: Tune the topology of T based on buffering information
5: Perform buffering on T
6: **return** T

3.3.1 Initial Tree Generation with Pre-Buffering

Timing-driven RST requires the calculation of AT on each sink and might need RAT on internal nodes during the tree construction. Simple estimation of timing is inaccurate since there is no way to calculate the delay of un-constructed part of the tree or consider the final buffer distribution in the tree construction phase. Instead of using estimation, we apply pre-buffering to guide the tree construction.

Fig.3.4 depicts the proposed initial tree generation flow. We first generate a tree through any timing-driven RST algorithm. In this chapter, we use state-of-the-art critical-trunk-based RST algorithm [38] to generate this initial tree (not considering blockages in this stage). Then pre-buffering part will buffer the RST and analyze timing. We save these topology and buffering if they are best-so-far. We calculate the real AT based on the buffered tree to

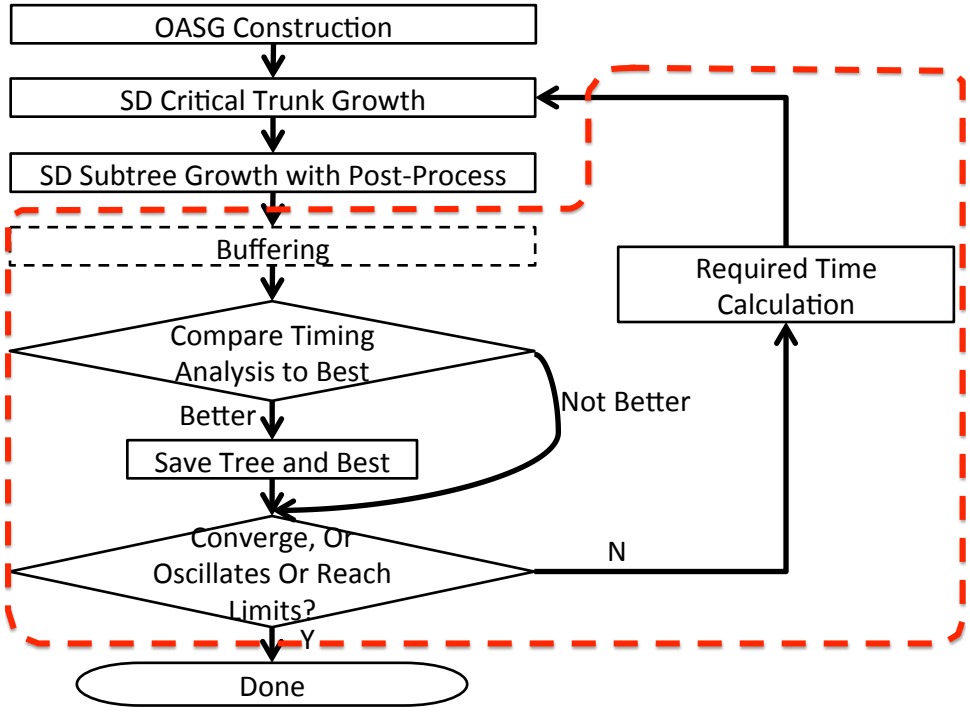


Figure 3.4: Flow of initial tree generation

substitute the pseudo time used in the tree topology generation algorithm as feedback information.

In the next iteration, all real critical sinks and critical trunks are re-determined because of the new timing information. In RST algorithm, we re-fix the critical trunks while the other two-pin nets are ripped up and re-routed by maze routing after the timing-driven critical trunk growth. Finally a post-process including rectilinearization and redirection is applied, which

produces another RST. We will iterate the whole procedure until the tree

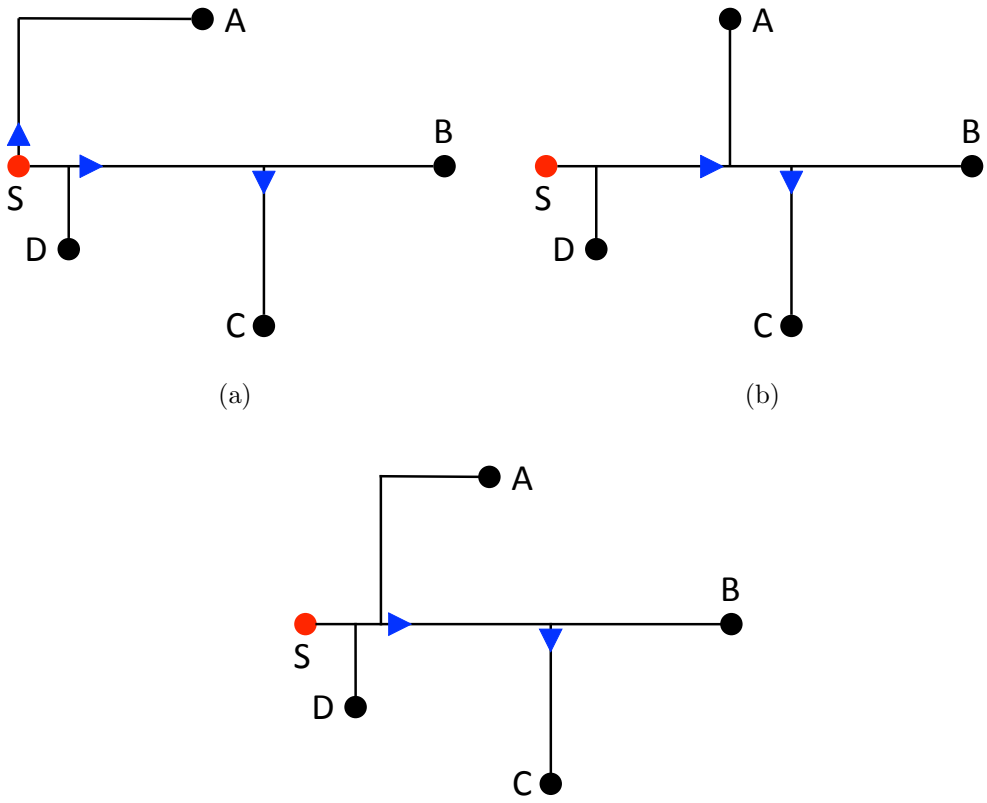


Figure 3.5: (a) is the initial critical trunk based tree with root S and sinks A,B,C,D. (b) reconstructs the tree according to the pre-buffering and timing information from (a). The tree topology converges in (c).

topology converges, or oscillates between several states, or the time limit is reached. Then we choose the best topology and WNS in our iterations as our initial tree. The new part of pre-buffering is indicated by dashed lines in Fig.3.4.

Example in Fig.3.5 shows that the topology and timing converge during the iterations. Initial structure in Fig.3.5(a) directly connects sink A to root as the RAT of A is small. In the next iteration, the topology generator decides to directly connect A to the trunk as in Fig.3.5(b), since according to Fig.3.5(a) the delay to A is small enough to meet the RAT, which in turn allows late branch. The late branch in Fig.3.5(b) leads to larger delay to sink A and eventually the topology converges to Fig.3.5(c) where the branch point of the path from root to A sits in the middle trunk leading to a star-like RSMT structure.

3.3.2 Buffering-Aware Over-the-Block Routing

We generate the initial tree without considering the blocks. The initial tree could cross over the blocks and break slew constraints even after buffer insertion. To prevent these violations, we change the topologies of over-the-block inside trees by approach similar to [55]. The objective in [55] is to minimize total wire-length only. Yet, in order to consider timing at the same time, we integrate criticality and slack into the objective function which minimize the wire-length of non-critical path as well as delay on critical path.

The initial tree confines a set of inside trees. For each inside tree, the ports, excluding the driver, on the boundaries of the block are called escaping points (EP). We use a mid-size hypothetical buffer at the driver and mid-size hypothetical buffers at each EP to determine if the tree has slew violation. Using mid-size hypothetical buffers instead of two extreme sizes will weaken the capability of utilizing more over-the-block routing resources, but the former turns out a more practical assumption and leads to less buffering cost as more solutions can propagate through this inside tree. If any inside tree violates the slew constraints, we apply three optimization primitives including parallel sliding, perpendicular sliding and EP merging [55] to fix the slew violations. Three optimization primitives are with different cost in our formulation since we consider timing as well.

For each inside tree t with slew violations, we first sort the illegal EPs per their slew violations. Next, in every iteration we choose the first illegal escaping point EP_1^t with the worst slew violation based on sorting. To improve

Table 3.1: Notation of variables in our formulation

X_{ij}	binary variable denoting the choice of PPS_{ij}^t, $X_{ij} = 1$ if it is chosen, otherwise $X_{ij} = 0$
E_{ij}	step slew reduction at EP_1^t if EP_i^t moves to PPS_{ij}
B_{ij}	output slew reduction on D^t if EP_i^t moves to PPS_{ij}
W_{ij}	estimated wire-length penalty of EP_1^t if EP_i^t moves to PPS_{ij}
C_i	estimated the timing criticality of EP_1^t

slew for EP_1^t, each escaping point from $\{EP_1^t, EP_2^t, \ldots EP_{|EP^t|}^t\}$ may slide to a different position by taking a combination of primitives.

The combination of optimization primitives provides each escaping point a set of possible points. Each possible point in the set is a point on the boundary edge where escaping point may move to, which in turn improves the worst slew. Moving every escaping point to certain possible point guarantees $slew_1^t$ to meet slew requirement. In the extreme situation where maximum slew constraint is zero, EP_1^t can still become legal escaping point after we merge one escaping point to another until only the driver is left. For any non-fixed $EP_i^t \in \{EP^t\}$, the j^{th} possible point associated with EP_i^t is denoted as PP_{ij}. PP_{ij} is stored in a 3-tuple format $\{E_{ij}, B_{ij}, W_{ij}\}$. E_{ij} and B_{ij} represent the step slew at EP_1^t and output slew reduction of the driver if EP_i moves to PP_{ij}. W_{ij} stands for the correspondingly estimated wire-length penalty. The possible point set associated with EP_i^t in the current iteration is denoted as PPS_i^t. $PPS_i^t = \{PP_{i1}^t, PP_{i2}^t, \ldots, PP_{ir}^t\}$, where r is the number of possible points inside.

In order to construct the inside tree under the slew constraint as well

as meeting slack constraints, $\forall EP_i^t \in EP^t$ we need to decide which possible point to choose. The simultaneous point choice problem can be formulated in an optimization problem as follows (notation in Table 3.1):

$$\text{min.} \sum_{i=1}^{|EP^t|} \sum_{j=1}^{|PPS_i^t|} X_{ij} W_{ij} (C_d(EP_i^t) C_i + \beta) \tag{5}$$

$$\text{s.t.}(S_{step1}^t + \sum_{i=1}^{|EP^t|} \sum_{j=1}^{|PPS_i^t|} X_{ij} E_{ij}^t)^2 +$$

$$(S^t(D^t) + \sum_{i=1}^{|EP^t|} \sum_{j=1}^{|PPS_i^t|} X_{ij} B_{ij}^t)^2 <= {slew_{spec}^t}^2 \tag{5a}$$

$$\sum_{j=1}^{|PPS_i^t|} X_{ij} = 1 \qquad \forall i \in \{1, 2, \ldots, |EP^t|\} \tag{5b}$$

The objective function (5) is to minimize the increase in delay on the critical paths and wire-length on non-critical paths. $W_{ij} C_d(EP_i^t)$ is the multiplication of resistance and total downstream capacitance, which estimates the amount of increase in delay for every sink downstream from EP_i^t. $C_i = \sum_{s_k} |slack(s_k)|$ is the weight for critical paths below EP_i^t, summing all absolute values of negative slacks of sinks downstream from EP_i^t. The weight β in the objective function selects solution with less estimated wire-length penalty on non-critical paths. The value of β is set remarkably smaller than $C_d(EP_i^t) C_i$ to avoid affecting critical paths. This objective function prefers less change on the critical paths while [55] can choose to increase the wire on critical path and exacerbate the WNS. Through the change of formulation, our new formulation considers the delay on critical paths and wire-length of

non-critical paths.One example is that Fig.3.6(c) is preferred to Fig.3.6(b) because the former reserves the timing for critical sink by moving escaping point on non-critical path to satisfy slew constraints. Constraint ($5a$) restricts that the total slew reduction on EP_1^t has to be able to pull $slew_1^t$ down below requirement. Constraint ($5b$) is used to limit only one position chosen for each escaping point.

3.3.3 Timing-driven Buffer-location-based Tuning

We apply the slew mode buffering to the timing-driven, over-the-block RST, which satisfies slew constraints with minimum buffering cost. In the slew mode buffering, each buffer is desired to drive to its limit, implying that the worst slew rate among all receivers (buffers or sinks) should reach the slew limit. Similar to the concept of slack in timing calculation, we define *slew margin* which means the worst input slew rate among all receivers does not reach the slew limit. The existence of slew margin is because the driver or Steiner points in the tree topology may enforce the buffering solution to place one buffer to shield capacitance from one side.

3.3.3.1 Slew Margin

In a RST, a Steiner point is the joint point for at least two sub-branches to merge at. Before propagating buffer solutions through the Steiner point, each sub-branch will have an unbuffered segment connected to the Steiner point, such as OB, Ob_1 in Fig.3.7(a). These segments do not require buffers

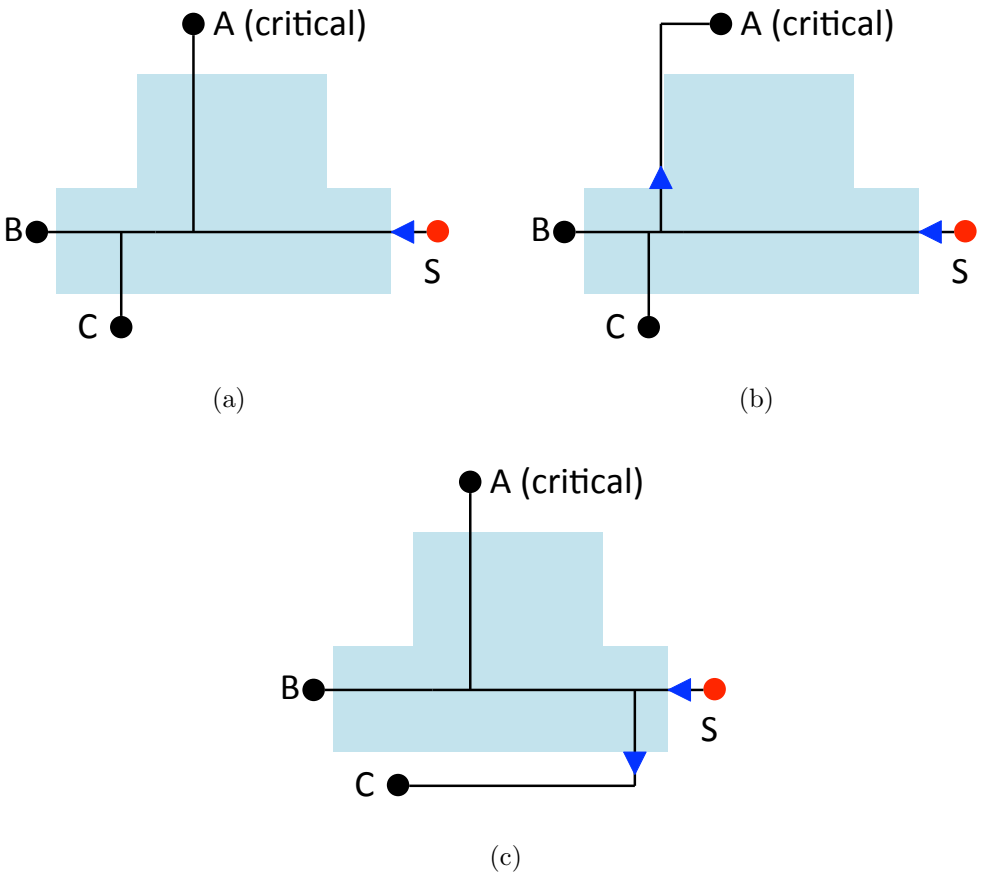


Figure 3.6: The root is S and three sinks are A, B, C. (a) is the initial timing-driven RST with slew violations. (b) fixes the slew violations with minimum wire-length penalty. (c) fixes the slew violations and considers the delay on critical path.

individually, but as a whole they may exceed the amount one large buffer can drive after propagating the Steiner point. The buffering tool has to place at least one buffer right below the Steiner point to shield one remaining segment to keep this solution legal. The buffering tool will place another buffer above the Steiner point to drive the unshielded parts along with the wire segment

above the Steiner tree (this buffer can be saved if root is above the Steiner point with ability to drive). For instance, in fig.3.7(b), S is driver and O is a Steiner point. The segment OB is shielded by inserting a new buffer b_2. The shielding buffer b_2 will not drive to its limit as we already know that the length of driven segment is less than the optimal reach length. Therefore, the stage below b_2 ends up with slew margin. In Fig.3.7(a), the slew limit we adapt is 70ps, and the stage driven by b_2 exhibits slew margin with maximum slew 60ps at sink B. We also notice that the stage driven by driver S also has slew margin since the maximum slew is 65ps at the input of buffer b_1.

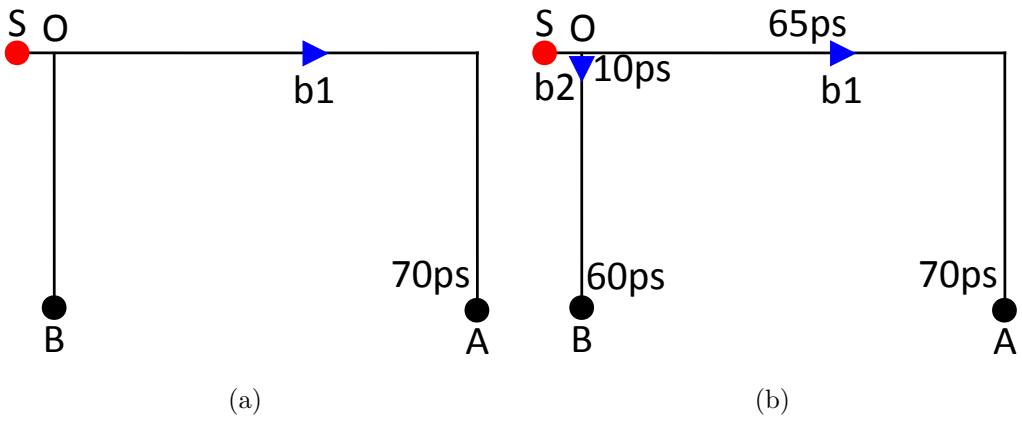


Figure 3.7: (a) bottom-up buffer solutions before merge at Steiner node O. (b) slew margin after propagation through Steiner node O

3.3.3.2 Buffer-location-based Tuning

Because the slew margin implies that the wire can be elongated to some extent without violating the slew constraints, the elongation of wires allows the change in topology without additional buffering cost. Per our approach,

there exists a way of changing topology to improve timing on critical path by elongating the wire with slew margin. As the slew margin occurs below the Steiner point, we extracts the simplified pattern with one Steiner point and two buffers demonstrated in Fig.3.8(a). Buffer b_1 sits right below the Steiner point for shielding and buffer b_2 stays above the Steiner point as in Fig.3.8(a). We analyze this simplified pattern to generalize the way of changing topology used in our topology-tuning algorithm. We annotate the stage driven by b_1 as $stage_1$, that driven by b_2 as $stage_2$ and that above b_2 as $stage_0$. Since $stage_1$ contains slew margin, we can calculate the elongation amount l to use up the slew margin. We denote the distance between b_2 and O as $l(b_2, O)$.

Observation 1 *If $l > l(b_2, O)$ and the input capacitance of buffers is negligible compared with wire capacitance, all slew constraints will be satisfied if we move the Steiner point to the location of b_2 and shift buffer b_1 up to the location right below the new location of the Steiner.*

Fig.3.8(b) shows this buffer-location-based tuning. Under the assumption of negligible input capacitance of buffer, the load and slew of $stage_0$ are not changed. The slew of the $stage_1$ is still within constraints owing to $l > l(b_2, O)$.

Observation 2 *If $l > l(b_2, O) + C_b/c_e$ and the input capacitance of buffers is not neglected, we can keep all slew constraints satisfied by moving the Steiner point to C_b/c_e above b_2 and shifting buffer b_1 up to the location right below the new location of the Steiner node. (C_b is the input capacitance of buffer b_1 and c_e is the unit capacitance for the wire segment above b_2)*

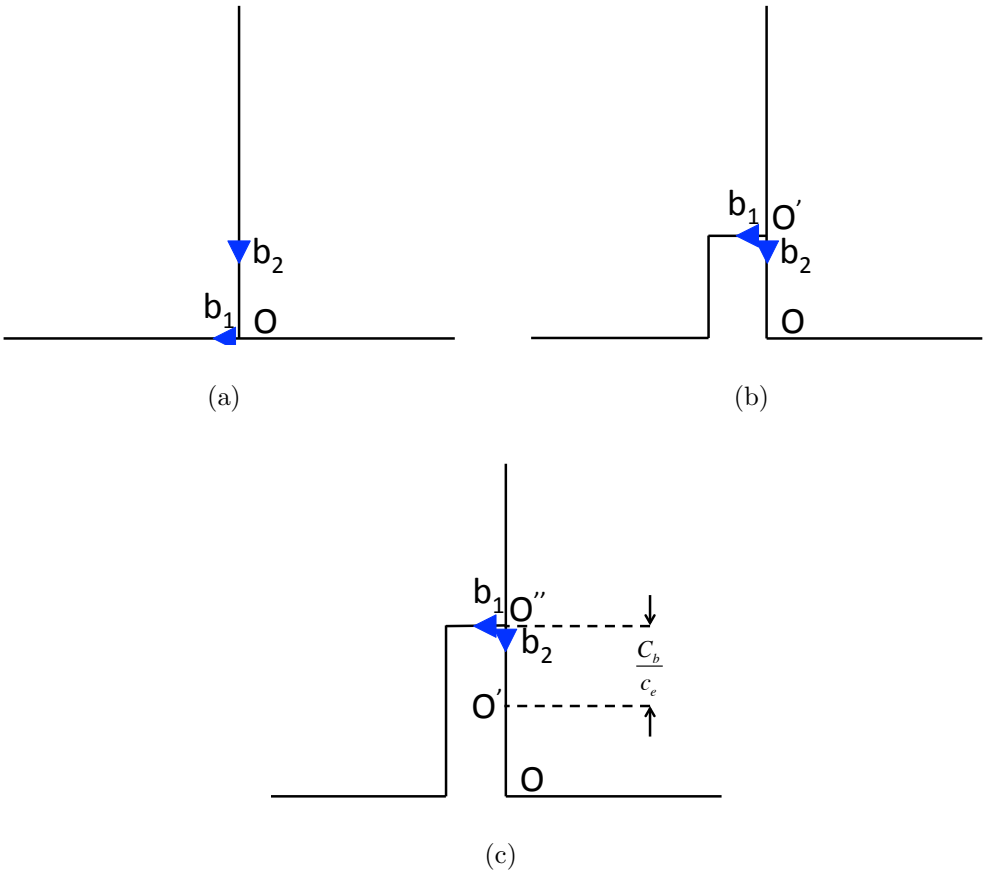


Figure 3.8: (a) depicts the pattern of slew margin. (b) shows buffer-location-based tuning if the input capacitance of buffers is negligible. (c) illustrates buffer-location-based tuning without neglecting the input capacitance of buffers.

Fig.3.8(c) illustrates the topology and buffering after the relocation of Steiner point O to C_b/c_e above b_2 and buffer shifting. Because the wire-length above b_2 is curtailed by C_b/c_e, the downstream capacitance for $stage_0$ is reduced by $C_b/c_e * c_e$ accordingly. Buffer b_1 is attached to $stage_0$ during buffer-location-based tuning, including C_b into the downstream capacitance. Therefore the

total downstream capacitance remains the same for $stage_0$. The amount of the downstream capacitance of $stage_2$ increases by $C_b/c_e * c_e$ as wire $O''O'$ is added below b_2. The input capacitance of b_1 is removed from $stage_2$ where the downstream capacitance is reduced by C_b. Hence the total downstream capacitance below b_2 stays the same. Under the assumption $l > l(b_2, O) + C_b/c_e$, the slew of $stage_1$ is still under slew constraints.

Algorithm 3 *Buffer-location-based Tuning*

Require: Buffered tree T
Ensure: Timing improved buffered tree T

```
1: Sort sinks in ascending order of slack
2: for each sink s_i with negative slack do
3:    node n = s_i
4:    while n! = s_0 do
5:       if n is Steiner point then
6:          if find buffer buffers b_1 right below n and b_2 above n then
7:             Calculate l based on slew margin
8:             if l > l(b_2, O) + C_b/c_e then
9:                T_copy = T
10:               Relocate n to C_b/c_e above b_2 and reconnect wires
11:               Shift buffer b_1 up to right below n
12:               if WNS(T) <= WNS(T_copy) then
13:                  T = T_copy
14:               end if
15:            end if
16:         end if
17:      end if
18:      n = Parent(n)
19:   end while
20: end for
21: return T
```

3.3.3.3 Algorithms

Our proposed algorithm searches for the pattern which satisfies all the above assumptions. The algorithm scans the buffered topology in a bottom-up fashion. Once a pattern analyzed in Section3.3.3.2 is detected, we perform the above-mentioned buffer-location-based tuning. The search starts from the worst negative slack sink among the set of sorted negative slack sinks. We evaluate the newly generated topology and commit the potential improvements. The algorithm is described in Algorithm3.

3.4 Experimental Results

We have implemented our algorithm in the C++ programming language. The experiments are conducted on an Intel Core 3.0GHz Linux machine with 32GB memory. We choose Gurobi Optimizer 5.10 as our solver for the integer linear programming.

RC01-RC12 are benchmarks in our experiments, same as those in [38]. We use two sizes of buffers in our experiment. The output resistances for two buffers are 450 ohms and 850 ohms, and the input capacitance are 3.8 fF and 1.9 fF respectively. Environment settings for wire and slew are calculated based on ITRS [3]. We use different resistance and capacitance for both horizontal and vertical layers. Each Steiner tree is placed on pre-selected layers. The slew constraint is set as 70 ps. Since the benchmarks do not comprise any timing information, to test the effectiveness of the slack optimization in our approach, we set RAT such that about 15% of the sinks are with negative

Table 3.2: Comparisons between TOB-RST-1, TOB-RST-2 and TOB-RST

Bench	Lin [38]			TOB-RST-1			TOB-RST-2			TOB-RST			
-marks	WNS (ps)	Buff	WL (um)	WNS (ps)	Buff	WL (um)	WNS (ps)	Buff	WL (um)	WNS (ps)	Buff	WL (um)	CPU (s)
RC1	-86	32	30220	-86	32	30220	-34	31	29370	-34	31	29370	0.52
RC2	-206	58	55700	-157	54	50880	0	52	48750	0	52	48750	0.89
RC3	-160	77	75730	-141	71	64270	-92	63	59530	-92	63	59530	0.82
RC4	-347	80	76340	0	84	79720	0	76	72920	0	76	72920	0.85
RC5	-305	95	92650	-177	102	97470	-108	96	96570	-108	96	96570	1.03
RC6	-722	134	130055	-722	134	130055	-521	123	118342	-423	123	119545	1.26
RC7	-605	179	185064	-574	174	182188	-249	162	178504	-162	162	179051	3.08
RC8	-418	189	185320	-220	191	190775	0	175	176920	0	175	176920	4.51
RC9	-787	182	177603	-517	186	180089	-126	168	162815	0	168	167240	7.70
RC10	-455	203	210040	-272	206	211910	-23	198	205650	0	198	209908	6.85
RC11	-1268	259	282338	-1142	265	287312	-1027	262	284077	-965	262	285290	11.41
RC12	-1221	885	1107538	-1008	912	1144662	-687	881	1101521	-245	881	1108324	27.38
Average	-548	1	1	-418	1.02	1.02	-239	0.96	0.97	-169	0.96	0.98	5.53

slack in a buffered minimum spanning tree interconnection.

We will evaluate pre-buffering, over-the-block routing and post-buffering tuning individually. We use the algorithm in [38] as baseline for our comparison since as far as we know it possesses state-of-the-art performance driven RST construction with buffering while others (such as [55] and [29]) are not timing-driven RST. We notate the timing-driven OA-RST constructed with pre-buffering as TOB-RST-1, the timing-driven RST with both pre-buffering, over-the-block routing as TOB-RST-2, and the final tree with pre-buffering, over-the-block routing and post-buffering tuning as TOB-RST.

3.4.1 Effectiveness of Pre-Buffering

First, to solely evaluate pre-buffering, we compare the performance of TOB-RST-1 with that of OA-RSMT generated by [38] in Table 3.2. Columns 5, 6, 7 in the table list the WNS, buffering cost and total wire-length of TOB-RST-1, while columns 2 to 4 present those for [38]. Since the required time of each sink is different in our experiments, the wire-length in column 2 is different from that of SD-OARST in [38]. As we can see, WNS is improved for most test cases, and the average improvement is 130 ps, while the change of buffering and wire-length is within 2%. The similarity of wire-length (buffering cost) demonstrates that the different set of critical sinks selected by pre-buffering benefits the slack with little impact on wire-length (buffering cost). In the experiments, the topologies of most benchmarks converge while only the topology of RC4 oscillates between two states and the better one of the two states is

returned. Also, all of the benchmarks converge or oscillate remarkably fast within four iterations at most.

3.4.2 Over-the-Block RST

To evaluate the effectiveness of over-the-block routing in TOB-RST-2, we compare TOB-RST-2 with TOB-RST-1. Columns 5 to 7 in Table 3.2 illustrate the WNS, buffering cost and total wire-length of TOB-RST-1 while the columns 8 to 10 are for TOB-RST-2. As shown in the table, over-the-block routing can improve WNS for all benchmarks. The average WNS improved from over-the-block routing is 179 ps with buffering cost and wire-length reduced by 6% and 5% respectively.

3.4.3 Post-buffering Topology Tuning

We compare TOB-RST with TOB-RST-2 to evaluate the effectiveness of post-buffering topology tuning. We only apply buffer-location-based tuning on critical paths with negative slack. Columns 11 to 13 in Table 3.2 present the WNS, buffering cost and total wire-length of TOB-RST. TOB-RST acquires about 70 ps improvements in WNS on average with less than 1% more wire-length. The buffering cost is the same since the post-buffering topology tuning does not consume buffering resources. We include total CPU runtime for TOB-RST algorithm in column 14 of Table 3.2, which contains total runtime of pre-buffering, over-the-block routing and post-buffering topology tuning. TOB-RST turns out to be fast since the maximum runtime is within one

minute.

3.5 Summary

In this chapter, we study a new class of RST problems, i.e., timing-driven over-the-block rectilinear Steiner minimum tree. We propose an effective and efficient algorithm which applies pre-buffering, over-the-block optimization and post-buffering tuning to optimize the slack on critical paths while saving wire-length on non-critical ones. Per our proposed approach, the generated topologies significantly improve WNS for all benchmarks along with 2% less wire-length and 4% less buffering cost than SD-OARST approach. Our proposed TOB-RST algorithm can be used in routing or post-routing stage to provide high-quality topologies to help close timing. This is the first work to solve timing-driven over-the-block RST problem crucial to high performance IC designs with multiple IP-blocks.

Chapter 4

Buffering-Aware Global Router with Over-the-Block Routing Resources Optimization

4.1 Introduction

In Chapter 2 and Chapter 3, rectilinear Steiner trees with wire-length and timing optimization are discussed separately. Tree construction is the most fundamental and crucial part of global routing problem, and with good tree construction algorithms, it could provide better routing solutions.

The CEDA-sponsored ISPD Global Routing Contests [4] and [5] attract attention from dozens of academic and industrial participants. Inspired by the competitions, many high-performance global routers are published, including but not limited to, FastRoute 3.0 [56], FastRoute 4.0 [52], BoxRouter 2.0 [17], NTUgr [15], NTHU-Route [25], NTHU-Route 2.0 [14], GRIP [51], FGR [47], MaizeRouter [40], Archer [45] and NCTU-GR [23].

Those global routers can be roughly divided into two categories: sequential and concurrent algorithms. Sequential works [56], [52], [15], [25], [14], [40], [45], [23] route the nets based on heuristic rip-up and reroute (RNR) techniques, which tend to run 2D global routing followed by layer assignment.

On the other hand, works such as [51] and [47] directly address the problem by running a full 3D global routing.

Meanwhile, extensively using IP-blocks to shorten turnaround time nowadays packs SOC designs with IP blocks or macros. To avoid routing over those blocks, obstacle-avoiding rectilinear Steiner minimum tree (OA-RSMT) problem has been actively studied over the years (e.g. [6,30,35,36]). However, completely avoiding those routing areas will result in significant underutilization of high-level metal layers which is the key to save power and close timing. To tackle that issue, new ideas of intelligently utilizing part of, instead of completely avoiding, the over-the-block routing resources with buffering awareness are proposed by [55] and [29] as BOB-RSMT [55] problem, as well as studied as scenic constraints in [39].

Since the guidance from two ISPD Global Routing Contests are similar, most published modern routers are aiming at the same problem: minimizing wire-length and via count in addition to alleviating congestion. However, the global routing problem has never been touched upon to not only consider wire-length, vias and overflows, but also properly use over-the-block routing resources. Studying this new problem is essential as to shorten the design cycle and improve the chip quality. If over-the-block routing resources are treated the same as that for out-the-block, long nets over the block will fail buffering, leading to additional manual work; whereas over-the-block routing resources are totally avoided, less remaining routing resources will significantly deteriorate the quality of the routing solution.

This chapter studies this new class of global router which tries to intelligently reclaim the "wasted" over-the-IP-block routing resources while minimizing overflows, wire-length and via count as in "basic" routers.

Our key contributions include:

1. We study the over-the-block global routing problem for the first time, providing global routing solution with overflows, wire-length, via count and buffering-awareness considered simultaneously.

2. We improve BOB-RSMT algorithm [55] by addressing its two limitations. Then we apply modified BOB-RSMT algorithm for our initial legal inside-tree generation.

3. For any block with overflow, in each iteration we evolve new topologies from inside trees confined within that block, with less cost associated with congestion, wire-length and via count,

4. We conduct Lagrangian-multipliers-based cost function to reflect the weighted impact from all generated topologies. It turns out topologies with less cost will have more impact on determining the cost of covered edges.

5. An RC-constrained A* search is proposed to help incrementally evolve new topologies with minimum cost while meeting slew constraints.

The rest of chapter is organized as follow. We first introduce basic concepts of inside trees, slew model and problem formulation in Section 4.2.

Our over-the-block routing algorithm will be presented in Section 4.3, which includes three subsections. Section 4.3.1 discusses how to modify BOB-RSMT to generate initial legal inside topologies. Section 4.3.2 illustrates the process of incrementally evolving new topologies according to Lagrangian-multiplier-based cost function and RC-constrained A* search. Experimental results are shown and analyzed in Section 4.4, followed by summary in Section 4.5.

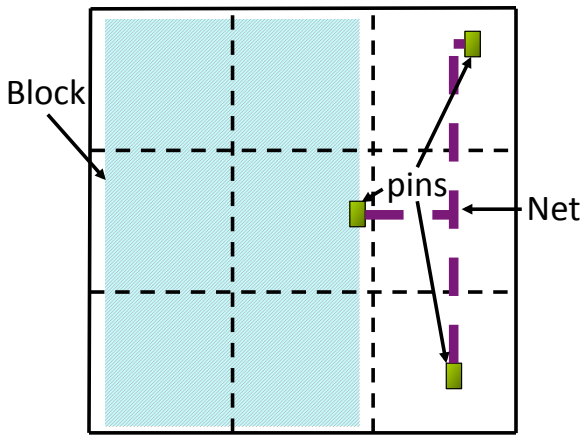
4.2 Preliminaries

4.2.1 Basic Over-the-block Concepts

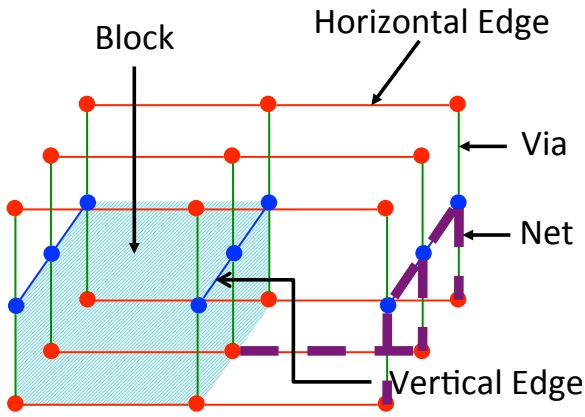
In global routing, the chip is partitioned into rectangular global routing bins where a 3D grid-graph $G = (V, E)$ is used to model the multi-layer design. As depicted in Fig. 4.1(a) and Fig. 4.1(b), each global routing bin is a vertex $v \in V$. The boundary between two adjacent global routing bins on the same layer is modeled as an edge $e \in E$ with a capacity c_e reflecting the maximum routing resources between the cells.

After placement, the chip is packed with IP blocks or macros which occupy the low metal layers and forbid any buffer-insertion. In our formulation, we set $B = \{b_1, b_2, \ldots, b_m\}$ as the set of blocks. Each block is modeled by a box in the 3D grid-graph G as the shadowed part in Fig. 4.1.

A set of multi-terminal nets $N = \{n_1, n_2, \ldots, n_k\}$ is required to be connected in the 3D graph G. The tree topology of each net n_i will enter and leave the blocks in the graph, which divides the whole tree topology into a set of outside trees TO_i and a set of inside trees TI_i.



(a)



(b)

Figure 4.1: 3D grid-graph G of three metal layers with each one divided into 3*3 global routing bins

For any inside tree t, the leaf nodes of t are on the boundaries of a block. Among all leaf nodes, one must be driving the signal and others are receiving. The leaf nodes that receive signals are escaping points, and the set

of escaping points for t is $EP^t = \{EP_1^t, EP_2^t, \ldots, EP_{|EP^t|}^t\}$.

We use the same model as in BOB-RSMT to check if any inside tree satisfies slew constraints. In our formulation, every inside tree is forced to be legal under this requirement to ensure signal integrity and buffering.

4.2.2 Problem Formulation

Matrices include wire-length, via cost and total overflow (TOF) are used to evaluate our routing solution. TOF is preferred to be zero since slightly overflowed global routing can still make detail routing considerably more difficult.

Our proposed buffering-aware global router will connect each net in N with the target of minimizing total wire-length in addition to reducing TOF. Over-the-block trees have to satisfy the slew constraints which ensure that every topology has feasible buffering solutions.

4.3 BOB-Router Algorithms

The overall flow of BOB-Router approach is depicted in Fig.4.2. The procedures in the "Main loop" frame consists of routing algorithm for inside trees while the rest part is composed of initial legal RSMT generation along with routing for outside trees.

In the BOB-Router problem formulation, any inside tree has to satisfy slew constraints to accommodate buffering. Due to this extra requirement,

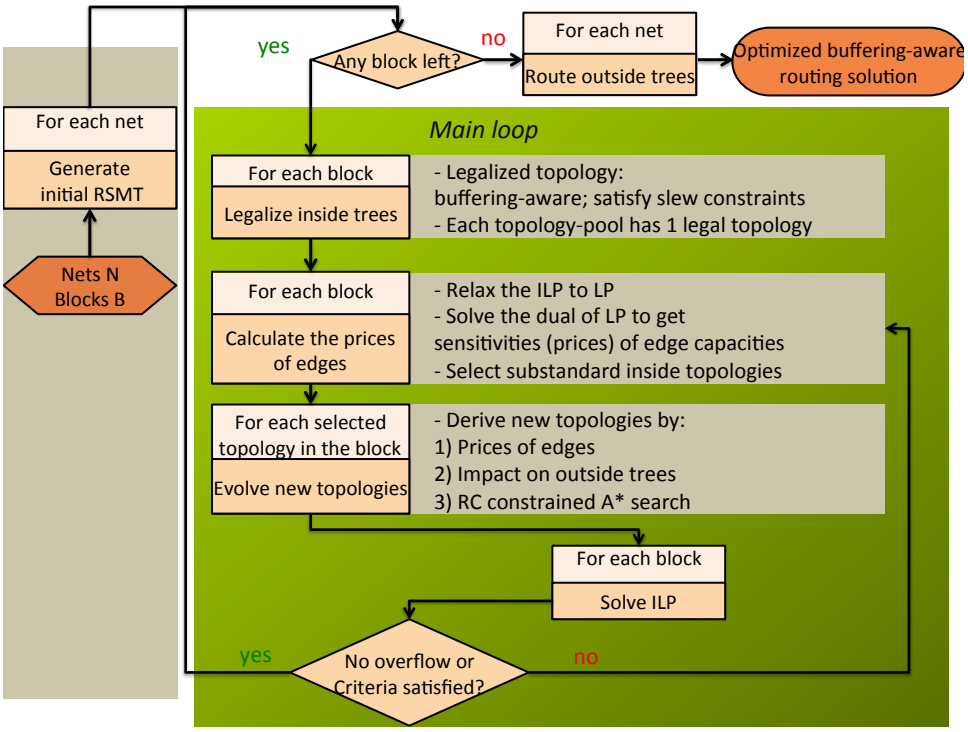


Figure 4.2: Overall flow of BOB-Router

routing for inside trees becomes more challenging than that for outside trees. Our BOB-Router will route inside trees ahead of outside trees, as we algorithmically emphasize on inside-tree routing which prefers topologies with least downside or even betterment on the cost of outside-tree routing.

To avoid simultaneously coping with wire-length, via count, overflow and slew constraints in inside-tree-routing problem, we decouple the slew constraints by legalizing all topologies first and making sure every following step

during the entire inside-tree routing will not violate the slew constraints. This statements could guarantee that the resulted inside trees are free from slew violation. The decoupling process includes two steps. First, since the initial inside trees could violate slew constraints, we apply an EP-movement-based legalization procedure modified from BOB-RSMT to legalize any illegal inside topology with minimum wire-length penalty. Second, the "evolve new topologies" step (shown in Fig.4.2) is the only step, after EP-movement-based legalization, that will change tree topologies. During this "evolve new topologies" step, in the inside-tree routing, we use an RC-constrained A* search to ensure that each operation during new topology evolution will not break the slew constraints.

4.3.1 Generate Legal Initial Topologies

We apply EP-movement-based legalization which is modified from BOB-RSMT algorithm to generate legal initial inside trees. BOB-RSMT approach efficiently generates a topology satisfying slew constraints, however, it has two limitations. First, movement of the driver for an inside tree is not considered. Second, when two branches at the opposite end of the driver move simultaneously, slew improvement may be underestimated.

To address those two limitations, we keep the optimization primitives but replace ILP with a greedy approach. Instead of evaluating each possible point and applying ILP to selection, we assess every *single-unit move* from all EPs and the driver, and select the best move. For example Fig.4.3(a)

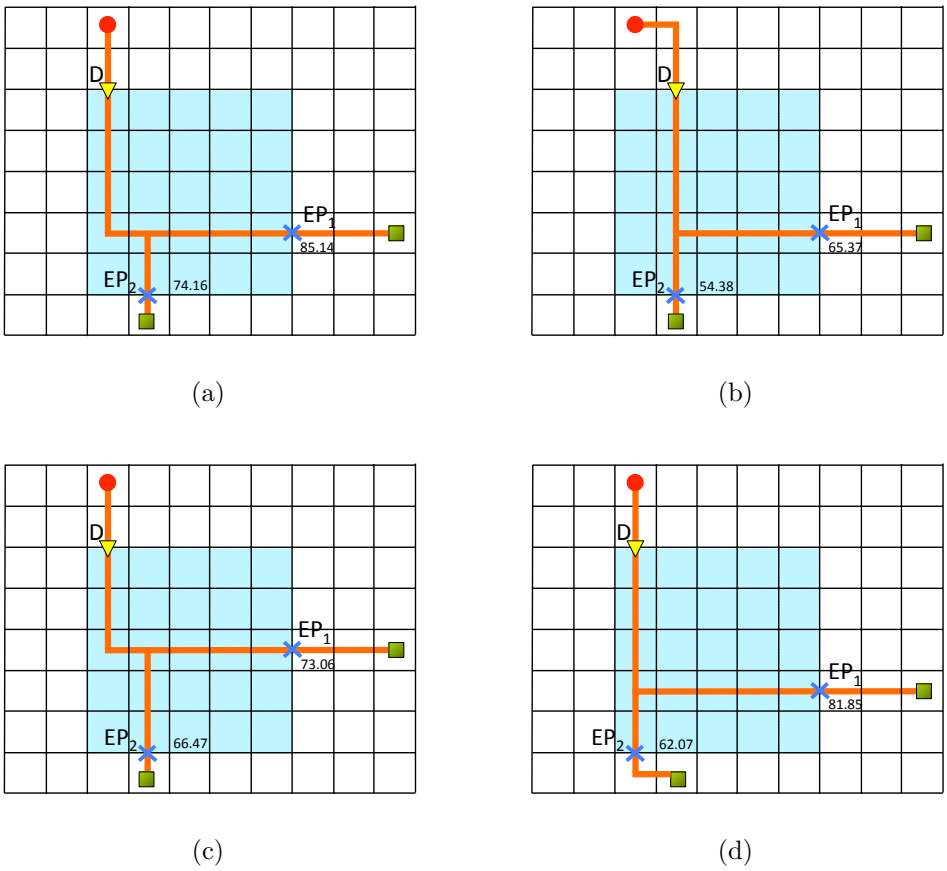


Figure 4.3: Best move selection (a) shows an illegal inside tree. (b), (c) and (d) exhibit and evaluate the best single-unit move from the driver, EP_1 and EP_2 respectively.

presents an illegal inside tree, while Fig.4.3(b), Fig.4.3(c) and Fig.4.3(d) give the resulted topologies assuming the best moves from the driver, EP_1 and EP_2 are selected respectively. In order to directly show amount of slew improvement in Fig.4.3, we set one for unit R and unit C, along with zero for buffer-output resistance, buffer-input capacitance and output slew in our slew model. As

we can see, the best slew improvement occurs in Fig.4.3(b), where the driver reduces worst slew by 19.88 slew units with single-unit move. However, in BOB-RSMT, this solution cannot be found since the move of the driver is not considered.

The other benefit from proposed BOB-RSMT-m is that it accurately catches the slew difference when multiple branches at the opposite end of the driver moving simultaneously. In this rare case, BOB-RSMT approach might disregard slew improvement from antenna clearance and overestimate slew improvement from branch sliding. As is shown in Fig.4.4, moving EP_1 to the right by one unit (Fig.4.4(b)) can improve the worst slew for 9.89 slew units while moving EP_2 in the same way (Fig.4.4(c)) will improve the worst slew for 3.30 slew units. If the ILP in BOB-RSMT is applied to choose both slides, the total improvement will be summed up to 13.19 slew units in an incorrect way. The actual improvement in Fig.4.4(e) is 11.98 slew units which consists of the slew improvement from moving EP_1 to the right by one unit and removing of antenna segment circled in 4.4(d).

4.3.2 Evolve More Legal Congestion-Aware Min-Cost Topologies

The initial-inside-tree legalization guarantees one legalized topology for each inside tree in any block. Placing these legal topologies simultaneously within each associated block could cause congestion problem. To resolve this issue, our approach uses the generated legalized topologies as seeds, giving birth to more legal congestion-aware topologies with less cost than current

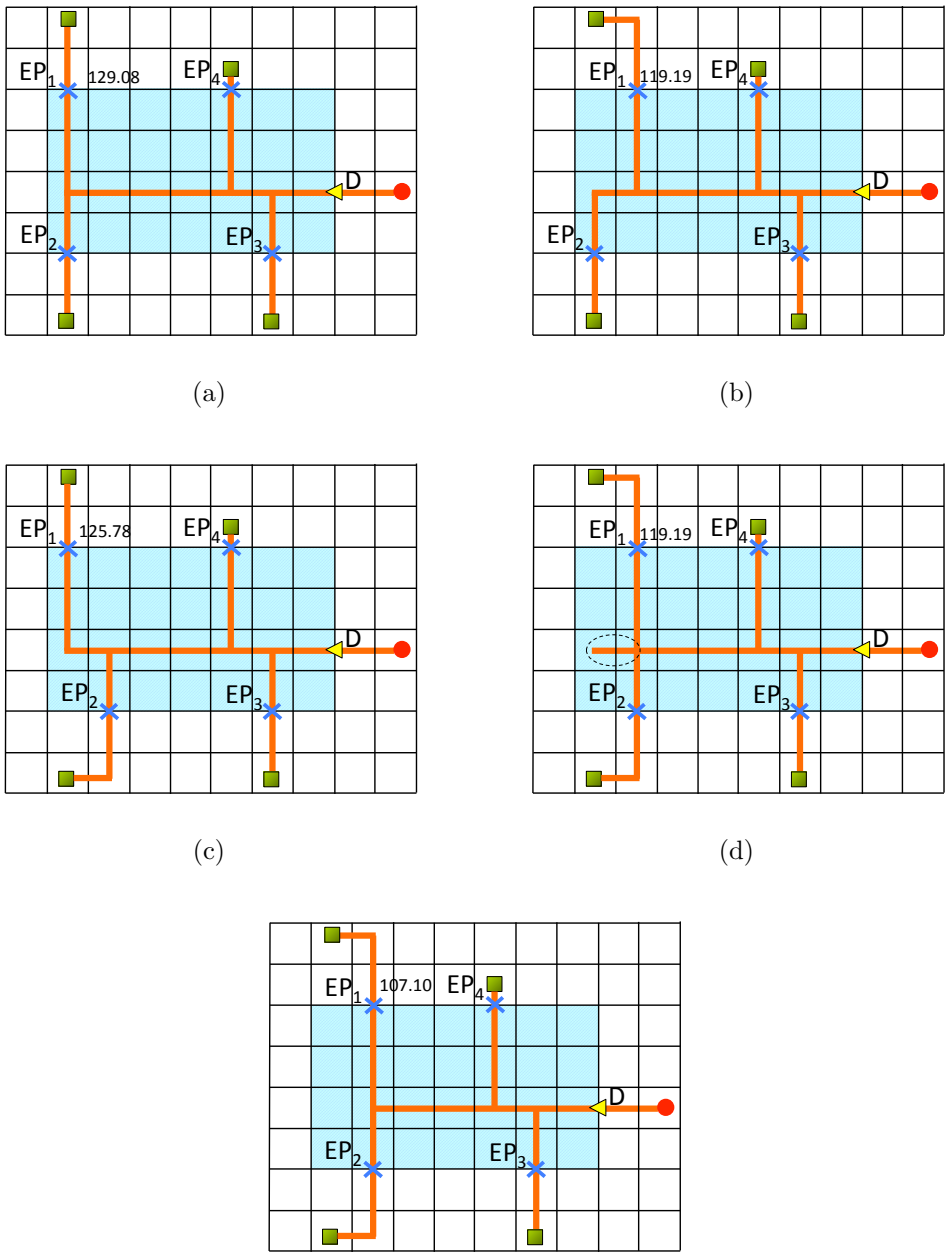


Figure 4.4: Slew calculation method in BOB-RSMT and BOB-RSMT-m. (a) shows an illegal inside tree. (b), (c) and (d) exhibit and evaluate the best single-unit move from the driver, EP_1 and EP_2 respectively.

topologies. Here, less cost means that the new generated topologies will have less combination of overflow, wire-length and vias. Finally, one topology will be chosen for each inside tree to achieve least overflow and cost.

Before introducing our method of evolving new topologies, we emphasize that we keep our topologies in Steiner tree structures instead of decomposing them into 2-pin nets in that (1) Steiner tree structures have more flexibility with unfixed Steiner points while 2-pin nets have to connect specified end points; (2) Steiner tree structure allows for tracking non-linear slew calculation over the entire tree, which is improbable for decomposed 2-pin nets. In normal global routing problem, it is non-trivial regarding how to come up with congestion-aware Steiner tree topologies with minimum cost. However, the Steiner tree topologies we demand for our inside trees have to satisfy additional slew constraints.

How to come up with congestion-aware Steiner tree topologies with minimum cost is always challenging in the global routing problem. More than that, the Steiner tree topologies we demand for our inside trees have to satisfy slew constraints. To solve all these problems, the flow of our proposed approach is:

- Assuming we have a topology pool for each inside tree, we use an ILP (same as [51]) to describe the topology-selection problem.
- We relax the ILP to a LP formulation. From the dual function of the LP, Lagrange multipliers related with the edge capacity constraints are

calculated to price the cost of each edge. The edge price will help evolve new topologies for selected topologies from high-congested areas to low-congested ones.

- We keep evolving new topologies within each round until certain criteria are met. In each round, we control the number of newly evolved topologies by a three-level topology-selection approach. Only topologies on current level will be evolved with new topologies. We advance to next level if current level stops optimizing our objective.

- An RC-constrained A* search is used to facilitate new topology generation without violating slew constraints, as well as minimize the cost of new topology.

- An ILP will be used to choose one topology among the topology pool for each inside tree to gain least overflow and cost.

Next, this chapter will talk about each step respectively.

Formulations First, we build an ILP formulation to describe the routing problem in each block. The ILP formulation contains no slew constraints, as

every topology presented in the ILP formulation is legal.

$$\min. \sum_{i=1}^{n} \sum_{t \in \zeta_i} X_{it} W_{it} + \sum_{i=1}^{n} M S_i \tag{4}$$

$$\text{s.t. } S_i + \sum_{i=1}^{n} X_{ij} = 1 \tag{4a}$$

$$\sum_{i=1}^{n} Q_i X_i <= C \tag{4b}$$

$$X_{it} \in \{0,1\} \qquad \forall i \in \{1,2,\ldots,n\} \qquad \forall t \in \zeta_i$$

$$S_i >= 0 \qquad \forall i \in \{1,2,\ldots,n\}$$

$TI = \{T_1, T_2, \ldots, T_n\}$ is the set of inside trees within block b, and ζ_i in the formulation is the collection of all topologies for T_i. For each Steiner tree topology $t \in \zeta_i$, parameter W_{it} represents the overall cost of the topology, including both wire-length and vias. Variable S_i denotes the routability of T_i; if S_i is positive, the inside tree T_i cannot be routed with available Steiner tree topologies. $(Q_i)_{et}$ indicates whether topology t contains edge e. $\sum_{i=1}^{n} Q_i X_i$ contains the amount of routing resources demanded on every edge in b, which is required to maintain under the edge capacity vector C.

In order to minimize overflow, MS_i is used in the objective function to penalize any unroutable inside tree T_i. Parameter M is a predefined large number which is greater than the wire-length of any possible Steiner tree in the chip. Solving ILP formulation (4) guarantees no overflow and minimizes total cost with maximum number of routed inside trees.

The ILP formulation has the following two purposes in our approach: i) to select one topology for each inside tree to check if overflow-free solution

could be achieved at the end of each iteration, ii) the dual problem of relaxed LP could provide cost of each edge.

Pricing the Edges Before solving the ILP and fixing the topologies in current iteration, more Steiner tree topologies, instead of the initial one solely, are wanted to effectuate least TOF and cost routing solution. We use sensitivity analysis on the edge capacity constraints to price each edge, which provides a guidance for the evolution of new Steiner tree topologies from current topologies.

Different edges on different layers have various values in the routing since some of them are in congested area while some are not. We calculate *price* to describe the potential overflow on each edge. To obtain the prices for edges, we first relax the ILP formulation (4) into an LP formulation by relaxing binary variables $\{X_{ij}\}$.

The relaxation on binary variables $\{X_{ij}\}$ splits the constraint of choosing only one topology for each inside tree into a set of fractional numbers indicating several potentially preferred topologies. A topology avoiding congested area and costing less wire-length and vias will be preferred and assigned positive X_{ij} which depends on the quality of the topology.

The price of each edge comes from the dual of this LP formulation exhibited in (6). The variable λ_i is the Lagrange multiplier associated with relaxed topology-selection constraint $(4a)$ for T_i and ρ_e is the Lagrange multiplier associated with relaxed capacity constraint $(4b)$ for edge e. Accord-

ing to complementary slackness theorem, for optimal primal variables $X_i^*, i \in \{1, 2, \ldots, n\}$ and optimal dual variable ρ_e^*, there exists $\rho_e^* * (\sum_{i=1}^{n} (Q_i)_e X_i^* - c_e) = 0$. When the ρ_e^* is positive, $\sum_{i=1}^{n} (Q_i)_e X_i^* - c_e = 0$ will be true, which means corresponding edge e has no "leftovers" in capacity. If the primal optimal solution exists, according to strong duality, the optimal dual variable ρ_e^* reflects how much improvement on the objective value we can make if the capacity of edge e increases by one. Therefore, we use the optimal dual variable ρ_e^* as the price for edge e. Compared with history-based cost in other routers, our price is more comprehensive because it considers all topologies we have and weights them according to their worth optimally. On the other hand, history-based cost may only consider certain recent topologies and do not have optimal way to weight these topologies. Our method will automatically weight each topology according to their quality which is represented by X_{ij}.

$$\max. \sum_{i=1}^{n} (-\lambda_i) + \sum_{e \prec b} (-\rho_e) c_e \tag{6}$$

$$\text{s.t. } \lambda_i + \sum_{e \prec t} \rho_e + W_{it} >= 0 \tag{6a}$$

$$\lambda_i >= 0 \qquad \forall i \in \{1, 2, \ldots, n\}$$

$$\rho_e >= 0 \qquad \forall e \prec b$$

Three-level Topology Selection If we evolve new topologies for each existing topology, the size of our topology pools will dramatically increase without corresponding speed of TOF mitigation. Therefore, we control the number of new evolved topologies in each iteration by only considering the most costly

Algorithm 4 *FindReroutes*

Require: Topology pool ζ, *level*, formulation variables S, O, P
Ensure: Selected topologies R

```
1: R = {}
2: if level = Level-one then
3:    for each S_i ∈ S do
4:       if S_i > 0 then
5:          R = R ⋃ ζ_i
6:       end if
7:    end for
8: else if level = Level-two then
9:    for each edge e in the block do
10:      if  O(e) < 0 then
11:         for each topology t ∈ ζ do
12:            if t contains e then
13:               R = R + t
14:            end if
15:         end for
16:      end if
17:   end for
18: else if level = Level-three then
19:   for each edge e in the block do
20:      if  P(e) < 0 then
21:         for each topology t ∈ ζ do
22:            if t contains e then
23:               R = R + t
24:            end if
25:         end for
26:      end if
27:   end for
28: end if
29: return  R
```

topologies.

We use a dynamic three-level topology-selection approach to determine

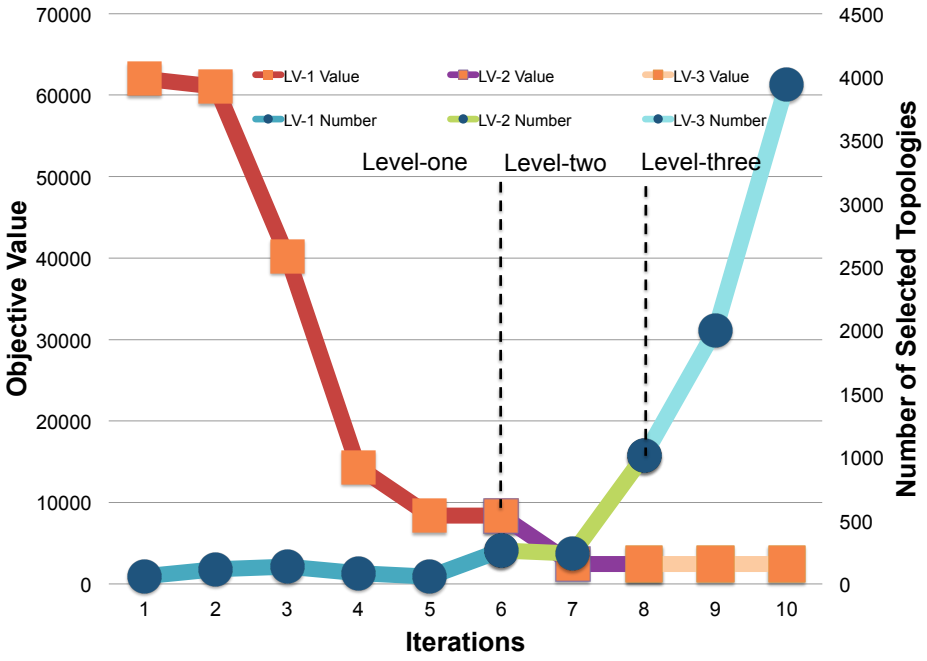


Figure 4.5: Progression of objective value and number of selected "to-be-evolved" topologies over optimization rounds for one block on ADAPTEC1

certain topologies for evolution in each iteration as depicted in Algorithm 4. Only if current stage fails to further improve TOF, will next stage be launched. The following level selects topologies in a more broad way which enables further TOF reduction.

- Level-one: After we find all inside trees with positive S_i, all topologies associated with these unroutable inside trees will be evolved.

- Level-two: If evolution of topologies from level-one is unable to keep optimizing the LP formulation, we assemble an inside-tree-routing solution by selecting the topology with largest X_{ij} for each inside tree. Then the overflow of each edge could be counted. In addition to the topologies from level-one, any topology containing overflowed edge(s) will be added.

- Level-three: If the topology evolution in level-two fails to keep optimizing the LP formulation, we evolve topologies covering edges with positive price in addition.

We gradually loosen our requirement for topology evolution, pushing the optimization with control over the number of processed topologies. Fig.4.5 evaluates the three-level topology-selection during optimization iterations for one single block on ADAPTEC1. It shows that the first iterations in level-one increases the size of topologies slowly. As optimization halts during any iteration, next level will be launched to reduce TOF.

RC-constrained A* Search After pricing and topology selection in every optimization iteration, we evolve new topologies with slew-aware rip-up and reroute. The pricey part will be ripped up and an RC-constrained A* search algorithm is applied to reroute disconnected parts without violating the slew constraints.

For any selected topology t, we find all wires with non-zero price, and

sort them by their prices in descending order. After sorting, we sequentially rip-up and reroute each wire. For one wire w on t, signaling from U to V, we remove w from t first. Then we calculate RC_p and C_p for each point $p \in t \backslash w$. RC_p and C_p are the maximum allowed RC and C connected to p without violation to the slew constraints. The maximum possible RC and C for all points on $t \backslash w$ will be:

$$RC^{max} = \max\{RC_p, p \in \{t \backslash w\}\} \tag{7}$$

$$C^{max} = \max\{C_p, p \in \{t \backslash w\}\} \tag{8}$$

Afterwards, an RC-constrained A* search is applied to reconnect V to the remaining part $t \backslash w$. We will only accept connections to point p with RC and C less than RC_p and C_p respectively. During RC-constrained A* search, any search path with RC exceeding RC^{max} or C exceeding C^{max} will be pruned away. The cost of each edge e in our A* search is the price of e plus one. The heuristic cost function we use is the 3-D Manhattan distance to the nearest point in $t \backslash w$, which clearly is a lower bound. This RC-constrained A* search guarantees least cost solution under slew constraints.

Net Ordering In traditional sequential routing, such as [14] and [15], net ordering plays important role because it will tremendously impact the quality of final routing solution. As shown in Fig.4.3.2, different net ordering leads to different overflow and wire-length.

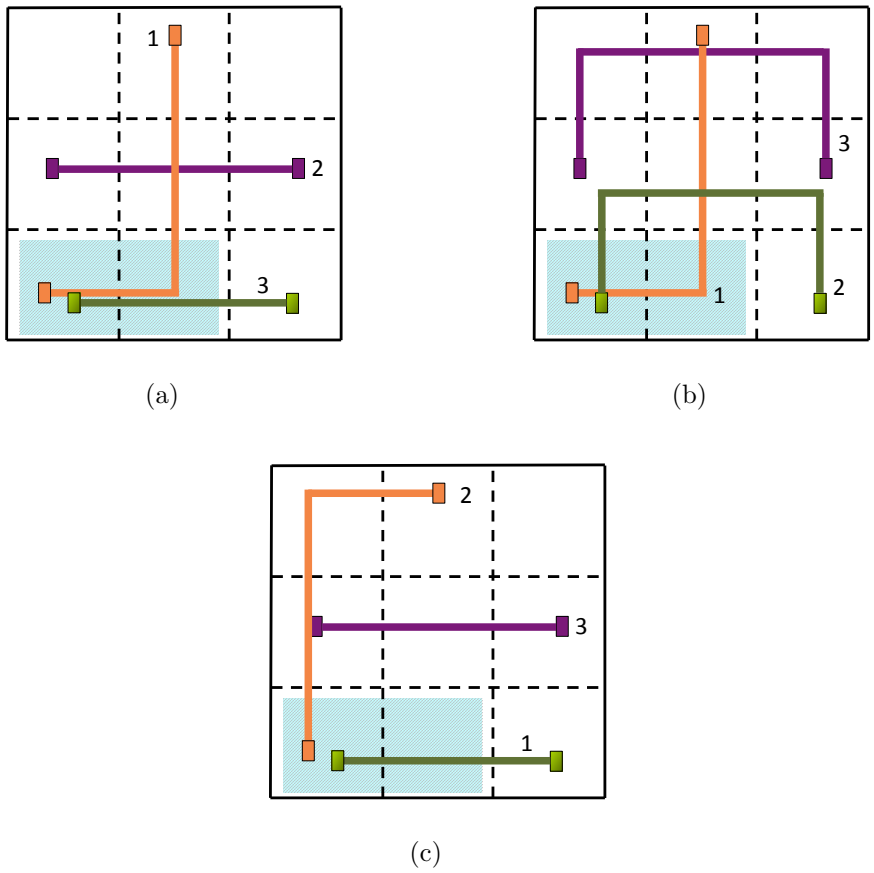


Figure 4.6: Impact of net ordering: (a) has overflows in shade area by sequencing orange, purple, green net. (b) has a different ordering of orange, green, purple but with detour of green and purple nets. (c) has the no overflow and detour by ordering green, orange, purple.

However, in concurrent algorithms, such as in [51] and [47], net ordering is not considered because all topologies are generated by solving one formulation. Our algorithm is not effected by net ordering in each iteration either. In each iteration, our algorithm pre-computes the edge prices which remains the same during the new topologies generation. Hence, each new

topology generated during the iteration is independent with net sequence in the current iteration.

4.3.3 Outside-tree Routing

After topologies of inside trees are fixed, capacities of all edges within blocks are set to zero before blockage-avoiding outside-tree routing, which will be solved by existing academic routers. Notice that even the capacities of all edges within blocks are set to zero, the existing router could still use them, this can be prevented by putting extra heavy penalty on over-the-block area.

4.4 Experimental Results

Table 4.1: Slew distribution of inside trees

Benchmarks	# nets	# inside trees	max slew	average slew
adaptec1	219794	57852	1713.8	36.9
adaptec2	260159	34769	494.4	28.5
adaptec3	466295	105137	23785.5	141.6
adaptec4	515304	86199	3986.7	65.8
bigblue1	282974	18763	380.1	22.1
bigblue2	576816	117259	69.9	4.0
bigblue3	1122340	79659	2025.1	22.1
bigblue4	2228930	234692	631.1	5.0

BOB-Router has been implemented in the C++ programming language. All experiments are conducted on an Intel Core 3.0GHz Linux machine with 16GB memory. We use 3D global routing benchmarks adaptec1 $\sim$ 4 and bigblue1 $\sim$ 4 from ISPD 2007 and 2008 Global Routing Contests for our ex-

Table 4.2: Comparisons between our proposed BOB-Router and OA-Router

Bench	over-the-block				outside-the-block				overall				OA-Router			
-marks	WL	Vias	TOF	cpu(s)	WL	Vias	TOF	cpu(s)	WL	Vias	TOF	cpu(s)	WL	Vias	TOF	cpu(s)
adaptec1	431886	138207	0	5690	2733837	1344218	199565	1421	3165723	1482425	199565	7111	3317320	1724765	450300	3463
adaptec2	261957	57838	265	4523	2615068	1258131	28847	1038	2877025	1315969	29112	5561	3371453	1836853	107498	4577
adaptec3	1235721	154123	1333	100210	8355049	2849048	639049	16527	9590770	3003171	640382	116737	10100613	3740726	1276779	18845
adaptec4	836840	105953	0	32718	8831370	2580484	329221	13202	9668210	2686437	329221	45920	11326871	3498262	438954	13455
bigblue1	98044	42090	0	55	3248498	1367350	22612	1637	3346542	1409440	22612	1692	3637249	1967568	70853	2232
bigblue2	258699	350385	0	520	3730497	2985365	3795	1131	3989196	3335750	3795	1651	4799773	3800398	5145	1346
bigblue3	522841	141885	0	2119	7800699	3847139	15148	2621	8323540	3989024	15148	4740	8961863	5267470	83416	8603
bigblue4	575639	731836	0	303	9358521	7489968	5266	2266	9934160	8221804	5266	2569	12363167	10444398	27939	5784
average	0.08	0.06	0.00	0.51	0.92	0.94	1.00	0.49	1.00	1.00	1.00	1.00	1.13	1.28	3.07	1.00

periments. Benchmarks from global routing contests are not annotated with blockage information explicitly. As far as we know, the block porosity information in the global routing benchmarks are derived from fixed macros in certain placement benchmarks. Owing to abutting blocks, it is arduous to directly retrieve geometric information of porosity areas from the routing benchmarks. Instead, we find the corresponding placement benchmarks, from which we are able to extract fixed macro geometric information. Besides, we remove nets containing pins inside blocks, which is beyond our formulation.

The wire resistance and capacitance for each metal layer are derived from ITRS [3], and we use 70ps as our maximal allowed slew.

We first evaluate the slew violation for each benchmark. Table 4.1 calibrates the slew numbers for all inside trees after RSMT topologies are generated by FLUTE and applied with a simple-layer-assignment heuristic. The heuristic will assign all inside trees on the lowest allowable pair of layers first. Then for all inside trees with slew violation, we will bring them to higher pair of metal layers according to extents of slew violations. From Table 4.1, we can see that some benchmarks with no slew problem initially, such as bigblue2, may encounter slew problem because it is possible that most of inside trees have been promoted to the highest pair of metal layers.

Since we eliminate all slew violation during initialization and keep slew under constraints, our final routing solution will not suffer from any slew problem. In Fig.4.7, we compare the slew distribution of inside trees from Table 4.1 with final routing solution for benchmark adpatec1. Initially, we observe the

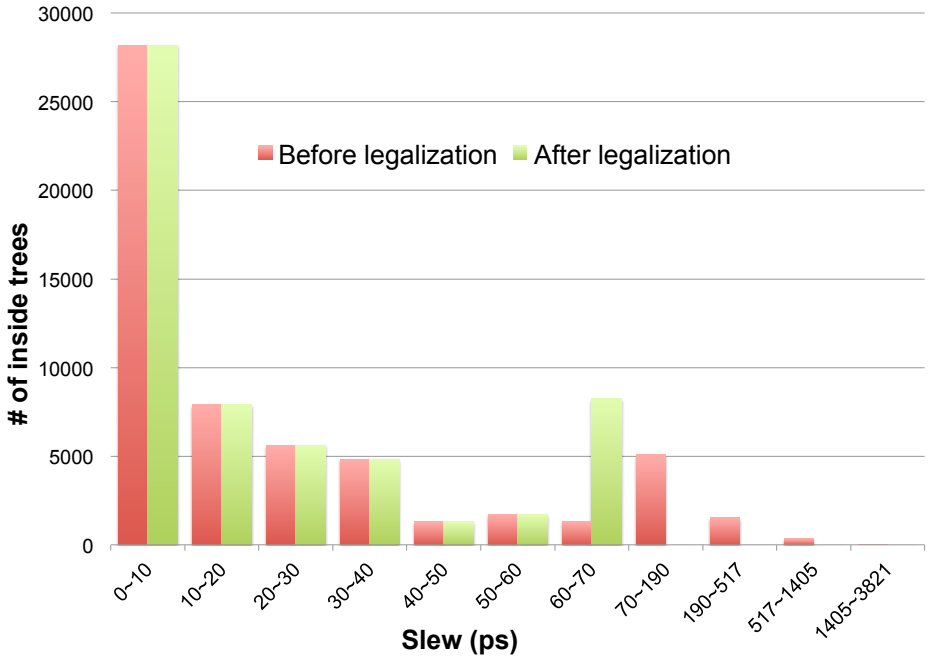


Figure 4.7: Slew distribution of all inside trees in adpatec1 initially and finally. Each y coordinates number of inside trees with slew in the slot between current and previous x

existence of inside trees with worst slew up to 1714ps. But after the benchmark is processed by BOB-Router, no inside tree has slew more than 70ps which is the maximum allowed slew rate in our slew constraints. The number of inside trees with slew between 60ps to 70ps is dramatically increased as most nets with slew violations originally are legalized to be just under 70ps.

Fig.4.8 analyzes the over-the-block overflow before EP-movement-based

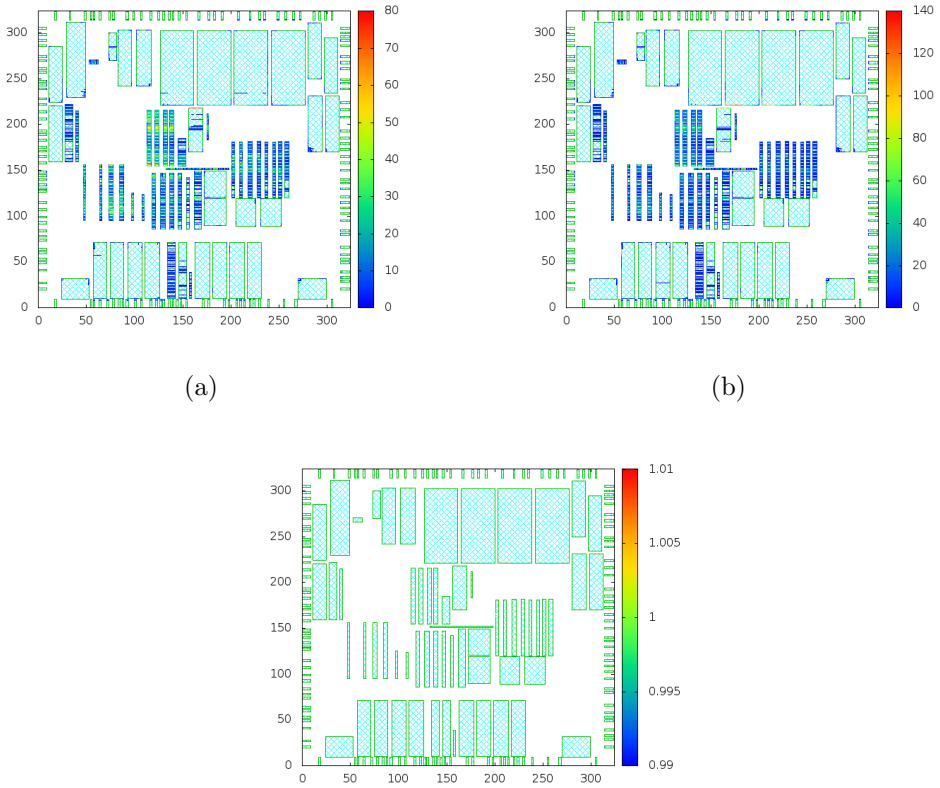


Figure 4.8: Over-the-block overflow analysis of a) before EP-movement-based legalization, b) after EP-movement-based legalization but before evolving new topologies c) after evolving new topologies and selecting new topology for each inside tree

legalization; after EP-movement-based legalization but before evolving new topologies; after evolving new topologies and selecting new topology for each inside tree, respectively. It shows that after EP-movement-based legalization, more overflows move to the edge of the blocks, which means some part of over-the-block routing moves out to help slew violation. Also, overflow slightly

improved after applying EP-movement-based legalization which is not designed to decrease overflow. 4.8(c) does not have overflow violation at all. This is because evolving new topologies and selecting the best topology for each tree eliminate the overflow violation completely.

If one router is not able to properly use the over-the-block routing resources, the safest way without breaking slew constraints thus involving manual work is to avoid the blocks completely by setting over-the-block routing capacity as zero (or large penalty). We compare our proposed BOB-Router with an obstacle-avoiding router (OA-Router) in terms of wire-length, via count and TOF. We modify NTHU-Router 2.0 [14] to be the OA-Router and the solver for outside-the-block routing for its good performance. The results are shown in Table 4.2. From the last row in the table, we can see that BOB-Router pushes about 8% of wire-length and 6% via count to the over-the-block part on average. The TOF of over-the-block routing is zero for most benchmarks. By using over-the-block routing resources, BOB-Router achieves about only 33% TOF, 88% wire-length and 78% via count of the OA-Router. We think more decrease of via count than wire-length is partially because BOB-Router performs full 3D routing for over-the-block part without layer assignment. Averagely, runtime of BOB-Router is same with OA-Router. The runtime of initial tree generation and legalization is negligible compared with solving LP and A* search. Solving LP and A* search divide the total runtime in an approximately even way. However, we notice that BOB-Router spends more time on bigger and tougher benchmarks, such as adaptec3. This

is because adaptec3 has non-zero overflow which requires maximum number of iterations and topologies are generated.

4.5 Summary

In the past few years, traditional global routing has been extensively studied, which in turn makes it hard even to improve performance by 1%. We propose a new formulation of global routing problem from a different perspective. Solving this new BOB-Routing problem could keep shortening design cycle and improving routing quality. With our proposed approach, we can generate slew-violation-free solution with 66% less TOF, 12% less wire-length and 22% less via count compared with the obstacle-avoiding approach.

To further explore this problem, it is worthwhile to fix the timing-critical nets by TOB-RSMT first, then route the rest nets for best overflow and wire-length.

Chapter 5

Conclusion

With technology keeps scaling into nanometer, interconnection optimization in VLSI design becomes more and more important yet challenging. Due to its importance, interconnection optimization, in particular global routing, has been studied for decades with many publications. Especially inspired by CEDA-sponsored ISPD Global Routing Contests, more related works have been done after 2008 as shown in Fig. 5.1. But, by looking into industrial VLSI design and interconnection, one important yet never been well-studied problem arouses excites and interests us. It is a huge waste as the over-the-block routing resources are not fully used currently. Lack of previous efficient methods will either cost huge manual work to use over-the-block routing resources or waste the routing resources which degrades the routing quality. In this dissertation, it studies this neglected yet important problem and provides a full set of solution for it.

In Chapter 2, the BOB-RSMT construction problem, as a fundamental part of global routing, has been studied. As we know, it is the first time slew values at the boundary of IP-blocks are selected as constraints for the RSMT construction. An effective and efficient algorithm which can reclaim the over-

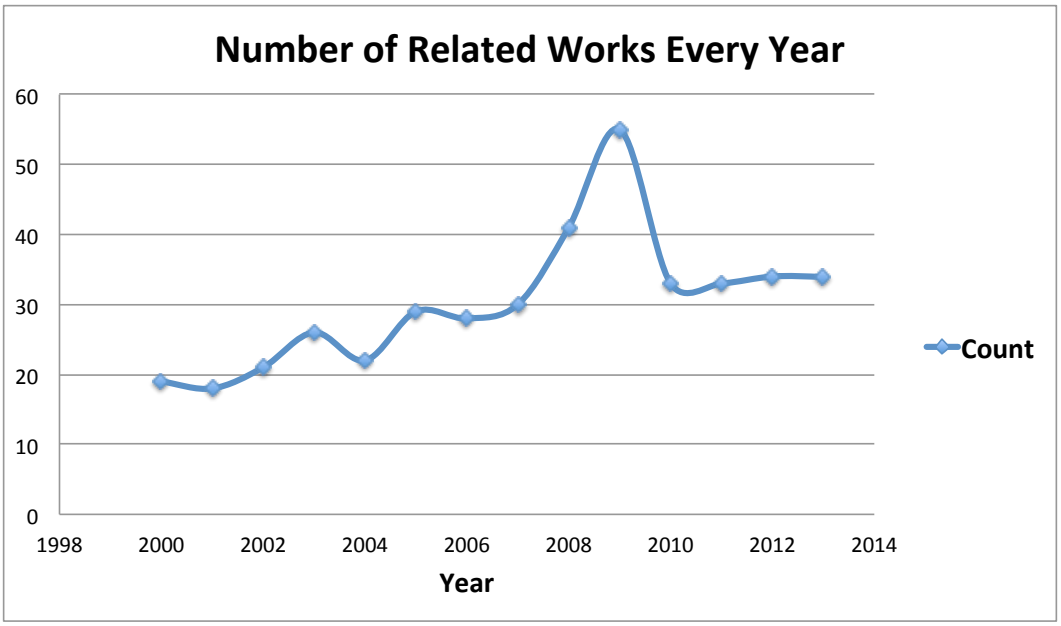


Figure 5.1: Number of routing related works which contains "VLSI routing" or "global routing" in title

the-IP-block routing resources and is beneficial to buffering is proposed to solve this new problem. With proposed approach, outside-the-block wire length and buffer cost are largely reduced and the constructed tree is buffering-friendly. This BOB-RSMT algorithm can be used in both pre-routing and global routing stage to provide high quality routing solutions.

More than wire-length-driven BOB-RSMT, timing-driven TOB-RSMT is studied in Chapter 3 for critical nets timing and wire-length co-optimization. The proposed timing-driven, over-the-block RST construction algorithm utilizes over-the-block routing tracks to reduce delay to critical sinks and shorten wire-length to non-critical sinks. It solves three common problems in timing-

driven RST construction: 1) accurately calculate criticality with buffered interconnect topology 2) timing-driven and over-the-block topology which saves wire-length on non-critical nets and timing on critical nets, and 3) post-buffering topology tuning further improves timing TOB-RST could provide topologies for critical nets during routing or ECO stages.

As RSMT algorithm providing topology for one single net, it consists a basis for the buffering-aware over-the-block routing problem. If one router is not able to properly use the over-the-block routing resources, the safest way without breaking slew constraints thus involving manual work is to avoid the blocks completely. As we know, previous routers never consider how to use over-the-block routing resources, not even consider obstacle-avoiding routing. BOB-Router is proposed to solve buffering-aware and over-the-block routing problem. It incrementally evolves new topologies for selected nets with minimum cost while meeting slew constraints. BOB-Router provides a solution with overflows, wire-length, via count and buffering-awareness optimized simultaneously.

Throughout these studies, a set of interconnection optimization algorithms are provided to optimize wire-length, via, timing, buffering cost together with over-the-block consideration. The future challenges of related works can be 1) using more accurate slew and delay models 2) considering critical nets during routing, and 3) improve the algorithm such that less runtime for routing problem. As interconnection optimization becomes more crucial and challenging with technology scaling, this dissertation provides a solution

for this practical and challenging problem.

Bibliography

[1] Gurobi Optimizer 4.52. http://www.gurobi.com/.

[2] http://en.wikipedia.org/wiki/Knapsack_problem.

[3] http://public.itrs.net/reports.html.

[4] ISPD 2007 Global Routing Contest and Benchmark Suite. `http://archive.sigda.org/ispd2007/contest.html`.

[5] ISPD 2008 Global Routing Contest and Benchmark Suite. `http://archive.sigda.org/ispd2008/contests/ispd08rc.html`.

[6] G. Ajwani, C. Chu, and W. Mak. FOARS: FLUTE Based Obstacle-Avoiding Rectilinear Steiner Tree Construction. In *Proc. ISPD*, pages 194–204, 2010.

[7] C. Alpert, Jiang Hu, Sachin S. Sapatnekar, and Paul Villarrubia. A practical methodology for early buffer and wire resource allocation. In *Proceedings of DAC*, pages 189–194, 2001.

[8] C. J. Alpert, G. Gandham, J. Hu, J.L. Neves, S.T. Quay, and S.S. Sapatnekar. Steiner tree optimization for buffers, blockages. and bays. In *Proc. IEEE Int. Symp. on Circuits and Systems*, pages 556–562, 2001.

[9] C. J. Alpert, M. Hrkic, J. Hu, A. B. Kahng, J. Lillis, B. Liu, S. T. Quay, S. S. Sapatnekar, A. J. Sullivan, and P. Villarrubia. Buffered Steiner Trees for Difficult Instances. In *Proc. ISPD*, pages 4–9, 2001.

[10] C.J. Alpert, G. Gandham, M. Hrkic, J. Hu, S. T. Quay, and C. N. Sze. Porosity aware buffered steiner tree construction. *IEEE TCAD*, 23(4):517–526, 2003.

[11] C.J. Alpert, M. Hrkic, J. Hu, and S. T. Quay. Fast and flexible buffer trees that navigate the physical layout environment. In *Proc. DAC*, pages 24–29, 2004.

[12] H. B. Bakoglu. Circuits, Interconnections, and Packaging for VLSI. Addison-Wesley, 1990.

[13] S. P. Boyd and L. Vandenberghe. Convex Optimization. Cambridge University Press, 2004.

[14] Y. Chang, Y. Lee, J. Gao, W. Wu, and T. Wang. NTHU-Route 2.0: A Robust Global Router for Modern Designs. In *IEEE TCAD*, volume 29(12), pages 1931–1944, 2010.

[15] H. Chen, C. Hsu, and Y. Chang. High-Performance Global Routing with Fast Overflow Reduction. In *Proc. ASPDAC*, pages 582–587, 2009.

[16] Chung-Kuan Cheng, Ting-Ting Y. Lin, and Ching-Yen Ho. New performance driven routing techniques with explicit area/delay tradeoff and simultaneous wire sizing. In *Proc. DAC*, pages 395–400, 1996.

[17] M. Cho, K. Lu, K. Yuan, and D. Z. Pan. BoxRouter 2.0: Architecture and implementation of a hybrid and robust global router. In *Proc. ICCAD*, pages 503–508, 2007.

[18] C. Chu and Y. Wong. FLUTE: Fast Loopup Table Based Rectilinear Steiner Minimal Tree Algorithm for VLSI Design. *IEEE TCAD*, 27(1):70–83, 2008.

[19] J. Cong, L. He, K. Khoo, C. K., and D. Z. Pan. Interconnect Design for Deep Submicron ICs. In *Proc. ICCAD*, pages 478–485, 1997.

[20] J. Cong, K. Leung, and D. Zhou. Performance-Driven Interconnect Design Based on Distributed RC Delay Model. In *Proc. DAC*, pages 606–611, 1993.

[21] Jason Cong, Lei He, Cheng-Kok Koh, and Patrick H. Madden. Performance optimization of VLSI interconnect layout. *Integration, the VLSI Journal*, 21(1-2):1–94, 1996.

[22] Jason Cong, Tianming Kong, and David Zhigang Pan. Buffer block planning for interconnect-driven floorplanning. In *Proceedings of IEEE ICCAD*, pages 358–363, 1999.

[23] K. Dai, W. Liu, and Y. Li. Efficient Simulated Evolution Based Rerouting and Congestion-Relaxed Layer Assignment on 3-D Global Routing. In *Proc. ASPDAC*, pages 570–575, 2009.

[24] Z. Feng, Y. Hu, T. Jing, X. Hong, X. Hu, and G. Yan. An O(nlogn) algorithm for obstacle-avoiding routing tree construction in the -geometry plane. In *Proc. ISPD*, pages 48–55, 2006.

[25] J. Gao, P. Wu, and T. Wang. A New Global Router for Modern Designs. In *Proc. ASPDAC*, pages 232–237, 2008.

[26] J. Hu, C.J. Alpert, S.T. Quay, and G. Gandham. Buffer insertion with adaptive blockage avoidance. *IEEE TCAD*, 22(4):492–498, 2003.

[27] J. Hu and S. S. Sapatnekar. Simultaneous buffer insertion and non-Hanan optimization for VLSI interconnect under a higher order AWE model. In *Proc. ISPD*, pages 133–138, 1999.

[28] S. Hu, C.J. Alpert, J. Hu, S.K. Karandikar, Z. Li, W. Shi, and C.N. Sze. Fast algorithms for slew-constrained minimum cost buffering. *IEEE TCAD*, 26(11):2009–2022, 2007.

[29] T. Huang and E. F.Y. Young. Construction of rectilinear Steiner minimum trees with slew constraints over obstacles. In *Proc. ICCAD*, pages 144–151, 2012.

[30] T. Huang and Evangeline F.Y. Young. An Exact Algorithm for the construction of Rectilinear Steiner Minimum Trees among Complex Obstacles. In *Proc. DAC*, pages 164–169, 2011.

[31] k. D. Boese, A. B. Kahng, B. A. McCoy, and G. Robins. Rectilinear Steiner Trees with Minimum Elmore Delay. In *Proc. DAC*, pages 381–386, 1994.

[32] A. B. Kahng and B. Liu. Q-Tree: A New Iterative Improvement Approach for Buffered Interconnect Optimization. In *Proc. IEEE Annual Symp. on VLSI*, pages 183–188, 2003.

[33] Richard M. Karp. Reducibility among Combinatorial Problems. In *Complexity of Computer Computations*, pages 85–103, 1972.

[34] C. V. Kashyap, Charles J. Alpert, F. Liu, and A. Devgan. Closed Form Expressions for Extending Step Delay and Slew Metrics to Ramp Inputs. In *Proc. ISPD*, pages 24–31, 2003.

[35] L. Li, Z. Qian, and Evangeline F.Y. Young. Generation of Optimal Obstacle-avoiding Rectilinear Steiner Minimum Tree. In *Proc. ICCAD*, pages 21–25, 2009.

[36] L. Li and Evangeline F.Y. Young. Obstacle-avoiding Rectilinear Steiner Tree Construction. In *Proc. ICCAD*, pages 523–528, 2008.

[37] C. Lin, S. Chen, C. Li, Y. Chang, and C. Yang. Efficient obstacle-avoiding rectilinear steiner tree construction. In *Proc. ISPD*, pages 127–134, 2007.

[38] Y. Lin, S. Chang, and Y. Li. Critical-trunk-based obstacle-avoiding rectilinear Steiner tree routings and buffer insertion for delay and slack optimization. *IEEE TCAD*, 30(9):1335–1348, 2011.

[39] W. Liu, Y. Wei, C. N. Sze, C. J. Alpert, Z. Li, Y. Li, and N. Viswanathan. Routing Congestion Estimation with Real Design Constraints. In *Proc. DAC*, pages 1–8, 2013.

[40] M. D. Moffitt. MaizeRouter: engineering an effective global router. In *Proc. ASPDAC*, pages 226–231, 2008.

[41] M.R.Garey and D.S.Johnson. The Rectilinear Steiner Tree Problem is NP-Complete. *Proceedings SIAM Journal on Applied Mathematics*, 32(4):826–834, 1977.

[42] T. Okamoto and J. Cong. Interconnect Layout Optimization by Simultaneous Steiner Tree Construction and Buffer Insertion. In *Proc. ASPDAC*, pages 44–49, 1996.

[43] Takumi Okamoto and Jason Cong. Buffered Steiner tree construction with wire sizing for interconnect layout optimization. In *Proc. ICCAD*, pages 44–49, 1996.

[44] P.J. Osler. placement driven synthesis case studies on two sets of two chips: hierarchical and flat. In *Proc. ISPD*, pages 190–197, 2004.

[45] M. Mustafa Ozdal and M. D. F. Wong. Archer: a history-driven global routing algorithm. In *Proc. ICCAD*, pages 488–495, 2007.

[46] M. Pan, C. Chu, and P. Patra. A Novel Performance-Driven Topology Design Algorithm. In *Proc. ASPDAC*, pages 244–249, 2007.

[47] J. A. Roy and I. L. Markov. High-Performance Routing at the Nanometer Scale. In *IEEE TCAD*, volume 27(6), pages 1066–1077, 2008.

[48] Prashant Saxena, Noel Menezes, Pasquale Cocchini, and Desmond A. Kirkpatrick. Repeater Scaling and Its Impact on CAD. *IEEE TCAD*, 23(4):451–463, 2004.

[49] Jack Y.-C. Sun. System Scaling and Collaborative Open Innovation. In *Symposium on VLSI Technology (VLSIT)*, pages T2 – T7, 2013.

[50] Jerry Wu, Yin-Lin Shen, Kitt Reinhardt, Harold Szu, and Boqun Dong. A Nanotechnology Enhancement to Moores Law. In *Applied Computational Intelligence and Soft Computing*, volume 2013(2), 2013.

[51] T. Wu, A. Davoodi, and J. T. Linderoth. GRIP: Scalable 3D Global Routing Using Integer Programming. In *Proc. DAC*, pages 320–325, 2009.

[52] Y. Xu, Y. Zhang, and C. Chu. FastRoute 4.0: Global Router with Efficient Via Minimization. In *Proc. ASPDAC*, pages 576–581, 2009.

[53] Jingyu Xua, Xianlong Hong, Tong Jing, Yici Cai, and Jun Gu. An efficient hierarchical timing-driven Steiner tree algorithm for global routing. *Integration, the VLSI Journal*, 35(2):69–84, 2003.

[54] Geoffrey Yeap. Smart Mobile SoCs Driving the Semiconductor Industry: Technology Trend, Challenges and Opportunities. In *IEEE International Electron Devices Meeting (IEDM)*, pages 1.3.1 – 1.3.8, 2013.

[55] Y. Zhang, A. Chakraborty, S. Chowdhury, and D. Z. Pan. Reclaiming Over-the-IP-Block Routing Resources With Buffering-Aware Rectilinear Steiner Minimum Tree Construction. In *Proc. ICCAD*, pages 137–143, 2012.

[56] Y. Zhang, Y. Xu, and C. Chu. FastRoute3.0: A Fast and High Quality Global Router Based on Virtual Capacity. In *Proc. ICCAD*, pages 344–349, 2008.

[57] Hai Zhou, D. F. Wong, I-Min Liu, and Adnan Aziz. Simultaneous routing and buffer insertion with restrictions on buffer locations. In *Proc. DAC*, pages 96–99, 1999.

[58] Qing Zhu, Mehrdad Parsa, and Wayne W. M. Dai. An Iterative Approach for Delay-Bounded Minimum Steiner Tree Construction. *Technical Report.*

Vita

Yilin Zhang was born in China in March 1986. He received a Bachelor of Science degree in Department of Electrical Engineering and Computer Science from Peking University (PKU) in China, 2008. He obtained Master degree in Department of Electrical and Computer Engineering from University of Texas at Austin (UT-Austin) in United States, 2010. He started his Ph.D. program at the University of Texas at Austin in 2008, with one year absence from 2010 to 2011, under the supervision of Professor David Z. Pan.

His research during his doctoral program includes rectilinear Steiner tree construction, buffering, static timing analysis (STA) and routing. He was the owner of three first-author papers, titled as "Reclaiming Over-the-IP-Block Routing Resources Using Slew Constrained Rectilinear Steiner Minimum Tree Construction", IEEE/ACM International Conference on Computer-Aided Design (ICCAD) 2012, "Timing-Driven, Over-the-Block Rectilinear Steiner Tree Construction with Pre-Buffering and Slew Constraints", ACM International Symposium on Physical Design (ISPD) 2014, "BOB-Router: A New Buffering-Aware Global Router with Over-the-Block Routing Resources Optimization", ACM/IEEE Asia and Pacific Design Automation Conference (ASPDAC) 2014, respectively. He also was the second-author in "O-Router: An Optical Routing Framework for Low-power on Chip Silicon Nano-Photonic Integration",

ACM/IEEE Design Automation Conference (DAC) 2009. Besides, he had two US patents as "An Efficient Ceff Model for gate output slew computation in early synthesis" (AUS820130764) and "A method to quickly prune impossible-to-win participants in a Tournament Pool" (AUS82013104) both in 2013.

He worked as a graduate research assistant of Prof. Pan from 2011 to 2014. Also, he worked as an intern in the Processor Design Tools group at Oracle in 2012 (Austin, TX) and research intern in IBM Research Center in 2013 (Austin, TX). He worked as a design engineer in Marvell in 2010.

He has served as a reviewer for several technology journals and conferences including IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems (TCAD), IEEE Transactions on Very Large Scale Integration Systems (TVSLI), IEEE International Symposium on Circuits and Systems (ISCAS), Optimization Theory & Applications in Engineering Sciences (OPTE), IEEE/ACM International Conference on Computer-Aided Design (ICCAD), ACM/IEEE Asia and Pacific Design Automation Conference (ASPDAC), ACM International Symposium on Physical Design (ISPD) and the IEEE/ACM Design Automation Conference (DAC).

Permanent address: zylime@gmail.com

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.