UNIVERSITY OF CALIFORNIA SAN DIEGO

Improved Detailed Placement and Routing Methodologies and Optimizations for Advanced Technology Nodes

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Electrical Engineering (Computer Engineering)

by

Bangqi Xu

Committee in charge:

2021

DEDICATION

I dedicate this thesis to my mother for her endless support and encouragement. I dedicate this thesis to my father, in loving memory. Without them, this thesis would not have been possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Foremost, I would like to thank my family for their endless love, support and encouragement. This adventure would not have been possible without their patience.

I would like to thank my advisor Professor Andrew B. Kahng for his continuous support and invaluable advice for conducting research as well as being a better person, throughout my Ph.D. study.

Besides my advisor, I would like to thank my thesis committee members, Prof. Chung-Kuan Cheng, Prof. Rajesh Gupta, Prof. Ryan Kastner and Prof. Farinaz Koushanfar, for their time to review and provide insightful comments on my Ph.D. studies.

I would like to thank my fellow labmates in the UCSD VLSI CAD Laboratory (Minsoo Kim, Dr. Seungwon Kim, Zhiang Wang and Mingyu Woo) and former lab members (Dr. Wei-Ting (Jonas) Chan, Ahmed Taha Elthakeb, Dr. Kwangsoo Han, Chia-Tung Ho, Dr. Hyein Lee, Dr. Jiajia Li, Hsin-Yu Liu, Uday Mallappa, Dr. Siddhartha Nath, Tushar Shah, Dr. Vaishnav Srinivas, Sriram Venkatesh and Dr. Lutong Wang) for inspiring discussions. I would also like to thank Prof. Seokhyeong Kang and Dr. Kambiz Samadi for their guidance and suggestions on my studies.

Last, but not least, I would like to thank my industrial collaborator Changho Han for his time for discussion and providing valuable suggestions. I would like to thank Dr. Wen-Hao Liu for mentoring me during my internship and on research projects. I would also like to thank Dr. Patrick Groeneveld and Dr. Stefanus Mantik for providing valuable feedback.

The material in this thesis is based on the following publications.

Chapter 2 contains the reprints of Changho Han, Kwangsoo Han, Andrew B. Kahng, Hyein Lee, Lutong Wang and Bangqi Xu, “Optimal Multi-Row Detailed Placement for Yield and Model-Hardware Correlation Improvements in Sub-10nm VLSI”, *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2017; Changho Han, Andrew B. Kahng, Lutong Wang and Bangqi Xu, “Enhanced Optimal Multi-Row Detailed Placement for Neighbor Diffusion Effect Mitigation in Sub-10nm VLSI”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38(9), 2019; and Andrew B. Kahng, Jian Kuang, Wen-Hao Liu and Bangqi Xu, “In-Route Pin Access-Driven Placement Refinement for Improved Detailed Routing Convergence”, to appear in *IEEE Transactions on Computer-Aided Design* *of Integrated Circuits and Systems*, 2021. The dissertation author is a main contributor to, and a primary author of, each of these papers.

Chapter 3 contains reprints of Andrew B. Kahng, Lutong Wang and Bangqi Xu, "The Tao of PAO: Anatomy of a Pin Access Oracle for Detailed Routing", *Proc. ACM/IEEE Design Automation Conference*, 2020. Chapter 3 also contains part of the draft of Andrew B. Kahng, Lutong Wang and Bangqi Xu, "TritonRoute-WXL: The Open Source Router with Integrated DRC Engine", in submission to *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021. The dissertation author is a main contributor to, and a primary author of, each of these papers.

Chapter 4 contains a reprint of Andrew B. Kahng, Lutong Wang and Bangqi Xu, "TritonRoute: The Open Source Detailed Router", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021. Chapter 4 also contains part of the draft of Andrew B. Kahng, Lutong Wang and Bangqi Xu, "TritonRoute-WXL: The Open Source Router with Integrated DRC Engine", in submission to *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021. The dissertation author is a main contributor to, and a primary author of, each of these papers.

My coauthors (Mr. Changho Han, Dr. Kwangsoo Han, Professor Andrew B. Kahng, Dr. Jian Kuang, Dr. Hyein Lee, Dr. Wen-Hao Liu and Dr. Lutong Wang, listed in alphabetical order) have all kindly approved the inclusion of the aforementioned publications in my thesis.

VITA

2015	B. S., Electrical and Computer Engineering, Shanghai Jiao Tong University, Shanghai, China
2015	B. S., Electrical Engineering, University of Michigan, Ann Arbor
2017	M. S., Electrical Engineering (Computer Engineering), University of California San Diego, La Jolla
2019	C. Phil., Electrical Engineering (Computer Engineering), University of California San Diego, La Jolla
2021	Ph. D., Electrical Engineering (Computer Engineering), University of California San Diego, La Jolla

All papers coauthored with my advisor Professor Andrew B. Kahng have authors listed in alphabetical order.

PUBLICATIONS

A. B. Kahng, J. Kuang, W.-H. Liu, and **B. Xu**, “In-Route Pin Access-Driven Placement Refinement for Improved Detailed Routing Convergence”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021), to appear.

A. B. Kahng, S. Kang, S. Kim, and **B. Xu**, “Enhanced Power Delivery Pathfinding for Emerging Die-to-Wafer Integration Technology”, *IEEE Transactions on Very Large Scale Integration Systems* (2021), to appear.

A. B. Kahng, L. Wang and **B. Xu**, “TritonRoute: The Open Source Detailed Router”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40(3) (2021), pp. 547-559.

A. B. Kahng, L. Wang and **B. Xu**, “The Tao of PAO: Anatomy of a Pin Access Oracle for Detailed Routing”, *Proc. ACM/IEEE Design Automation Conference*, 2020, pp. 1-6.

V. A. Chhabria, A. B. Kahng, M. Kim, U. Mallappa, S. S. Sapatnekar, **B. Xu**, ”Template-based PDN Synthesis in Floorplan and Placement Using Classifier and CNN Techniques”, *Proc. Asia and South Pacific Design Automation Conference*, 2020, pp. 44-49.

S. Dolgov, A. Volkov, L. Wang and **B. Xu**, “2019 CAD Contest: LEF/DEF Based Global Routing”, *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2019, pp. 1-4. (**Invited Paper**)

C. Han, A. B. Kahng, L. Wang and **B. Xu**, “Enhanced Optimal Multi-Row Detailed Placement for Neighbor Diffusion Effect Mitigation in Sub-10nm VLSI”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38(9) (2019), pp. 1703-1716.

T. Ajayi, V. A. Chhabria, M. Fogaa, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee, U. Mallappa, M. Neseem, G. Pradipta, S. Reda, M. Saligane, S. S. Sapatnekar, C. Sechen, M. Shalan, W. Swartz, L. Wang, Z. Wang, M. Woo and **B. Xu**, "Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project", *Proc. ACM/IEEE Design Automation Conference*, 2019, pp. 76:1-76:4. (**Invited Paper**)

A. B. Kahng, S. Kang, S. Kim, K. Samadi and **B. Xu**, "Power Delivery Pathfinding for Emerging Die-to-Wafer Integration Technology", *Proc. Design, Automation and Test in Europe*, 2019, pp. 836-841.

Y. Cao, J. Li, A. B. Kahng, A. Roy, V. Srinivas and **B. Xu**, "Learning-Based Prediction of Package Power Delivery Network Quality", *Proc. Asia and South Pacific Design Automation Conference*, 2019, pp. 160-166.

A. B. Kahng, L. Wang and **B. Xu**, "TritonRoute: An Initial Detailed Router for Advanced VLSI Technologies", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2018, pp. 81:1-81:8.

C. Han, K. Han, A. B. Kahng, H. Lee, L. Wang and **B. Xu**, "Optimal Multi-Row Detailed Placement for Yield and Model-Hardware Correlation Improvements in Sub-10nm VLSI", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2017, pp. 667-674.

J.-A. Carballo and **B. Xu**, "The Architecture Value Engine: Measuring and Delivering Sustainable SoC Improvement", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2016, pp. 24:1-24:5. (**Invited Paper**)

ABSTRACT OF THE DISSERTATION

Improved Detailed Placement and Routing Methodologies and Optimizations for Advanced Technology Nodes

by

Bangqi Xu

Doctor of Philosophy in Electrical Engineering (Computer Engineering)

University of California San Diego, 2021

Professor Andrew B. Kahng, Chair

In advanced technology nodes, aggressive device scaling along with fundamental physical (lithographic patterning, CMP, reliability, variability, etc.) and circuit (crosstalk, delay, etc.) limitations remain. As a result, ever-more complex design rules introduce challenges for the design automation tool flow, especially placement and routing (P&R). Moreover, as feature sizes shrink, there is increased difficulty of modeling the behavior of devices, as the proximity of devices significantly affects device performance.

The increasing complexity and difficulty lead to three challenges. First, turnaround times of both automated design tool flow and manufacturing increase due to (i) model-hardware miscorrelation and (ii) miscorrelation in different P&R tool stages. Second, direct application of academic works is limited because research works focus more on abstracted and simplified problems, while leaving the key elements of such abstraction and simplification as open questions. Third, the gap between academia and industry is widening because academic works tackle highly-dependent problems with independent and disjoint efforts. For example, the open literature is dominated by isolated research works on global routing and detailed routing, where the crucial correlation between these stages is ignored.

To address these three challenges, this thesis presents research works in three directions: (i) detailed placement optimization for correlation improvement; (ii) key elements of enablement for routing in advanced technology nodes; and (iii) an open source, end-to-end global-detailed routing tool that gives a first-ever academic routing flow for advanced technology nodes.

To improve correlation with detailed placement optimization, this thesis presents two works: (i) an optimal multi-row detailed placement optimization for neighbor diffusion effect mitigation; and (ii) an in-route, pin-access driven detailed placement refinement for detailed routing convergence improvement.

To enable academic research on routing, this thesis presents two works on key elements: (i) a geometry-based design rule check engine and (ii) a dynamic programming based pin access analysis engine.

To narrow the gap between academia and industry in routing, this thesis presents an end-to-end, complete routing flow for advanced technology nodes. Implementation of the routing flow along with the aforementioned design rule check engine and pin access analysis engine are open-sourced under a permissive license.

Chapter 1

Introduction

Over the past decade, the semiconductor industry has made remarkable progress in process technology, advancing from the $28nm$ node to the $5nm$ node. This progress has improved the power efficiency and performance of electronic devices, enabling new possibilities that span from mobile and wearable customer devices to cloud computing and machine learning applications. However, the progress in semiconductor process technology does not come for free. In order to push toward the limits of patterning, materials and nanofabrication, process technology has become increasingly complex, which translates to increase in design and automated tool flow complexity. This thesis presents several challenges that arise in advanced technology nodes, along with efforts to address these challenges.

1.1 Challenges

Increasing complexity in advanced technology nodes brings new stresses to both industry and academia. For industry, new technology nodes come with new device behavior and new design constraints; these challenge existing models and abstractions for devices, as well as models that underlie the automated design tool flow, especially in placement and routing (P&R). As a result, the benefit from new technology nodes – in terms of transistor density scaling – realized in actual products has slowed down from the traditional Moore's Law as shown in Figure 1.1. In order to harvest the last drops of benefit from new technology nodes, it is essential to understand the correlation challenges in existing models which have blocked these benefits up to now.

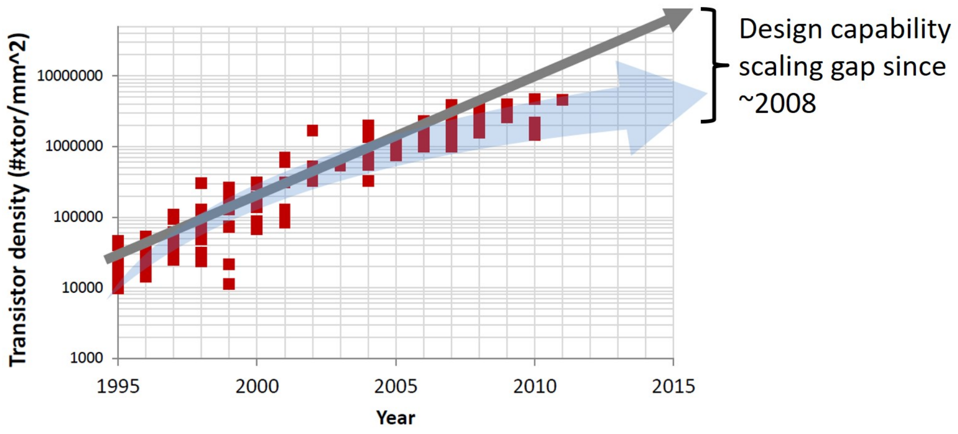


Figure 1.1: Gap between "available" density scaling (gray arrow) and "realizable" density scaling in MPU products (red squares). [45]

For academia, the ever-more complex design rules make research progress more challenging. Figure 1.2 shows the page count of the LEF/DEF (Library Exchange Format / Design Exchange Format) [117] reference over the past 15 years, which reflects the increasing complexity of design rules. As a result, academic research works tend to spend efforts on abstracted problems or specific subproblems of large research topics. This leads to two challenges. The first challenge is how to obtain valid abstracted problems. The second challenge is how to connect subproblems that are derived from the same large research topic.

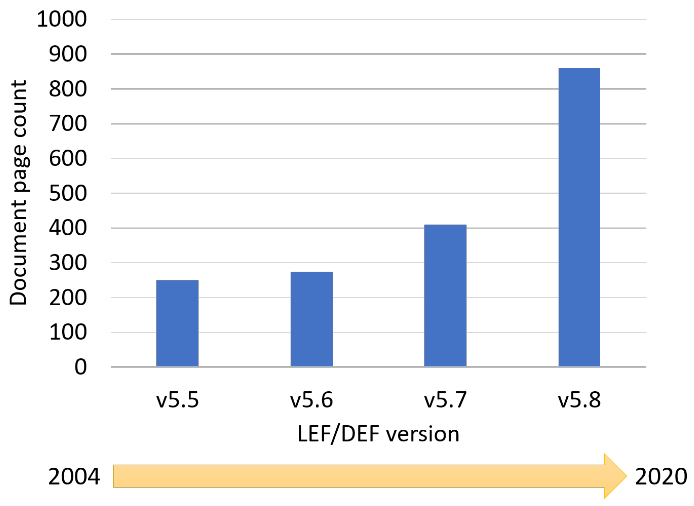
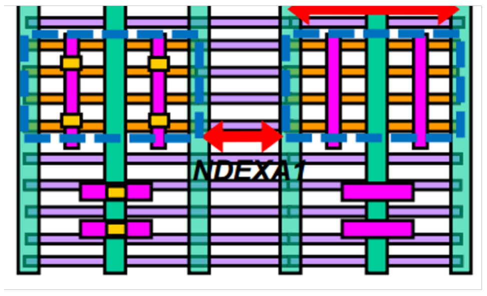


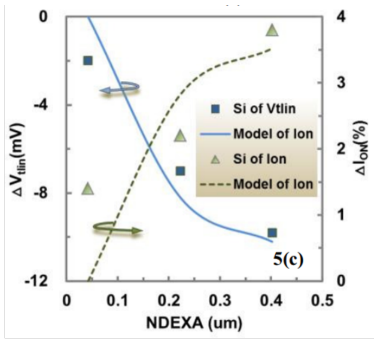
Figure 1.2: LEF/DEF document page count over the years.

1.1.1 Model-Hardware and Model-Model Correlation

New technology nodes challenge existing models used in design and design-to-manufacturing methodologies. One aspect of these model challenges is model-hardware correlation. In advanced technology nodes, device behavior no longer depends on independent geometry parameters [27]. Due to aggressive device scaling, lithography limitations and process complexity, *layout-dependent effect* (LDE) arises from the proximity of devices. An example of LDE is *neighbor diffusion effect* (NDE). Since neighboring diffusion area creates stress on the device channel, the threshold voltage and the drive strength (i.e., I_{on}) depends on the width of neighboring diffusion spacing (NDEXA) [11]. Figure 1.3 illustrates the actual layout of neighboring devices and the threshold voltage and drive strength as a function of NDEXA. If such effect exists only within a standard cell, the impact can be captured by library characterization. However, if such effect arises between devices from neighboring standard cells, the impact is not captured by library characterization, thus causing miscorrelation between timing model, power model and the actual silicon hardware (i.e., *model-hardware miscorrelation*).



(a)



(b)

Figure 1.3: Illustrations of (a) actual device layout; and (b) threshold voltage and drive strength as a function of the width of neighboring diffusion spacing (NDEXA).

Another aspect of model challenges is related to the conventional automated design tool flow. The conventional tool flow tends to use different models, even for the same purpose, due to the intrinsically separated nature of design tool stages. As the solution space becomes more restricted due to the increasing complexity of design rules, the models from different tool stages may not correlate well with each other (i.e., *model-model miscorrelation*). An example of such model-model miscorrelation is related to pin access. Conventionally, a placement tool uses models (e.g., pin density) to estimate pin accessibility. Such models usually only consider information pertaining to a given standard cell. However, when we consider routing information (e.g., routing tracks), two instances of the same standard cell may have very different pin accessibility according to the different track offsets of the two instances, as shown in Figure 1.4. In a conventional tool flow, placement is rarely revisited to cure pin access issues during the routing stage. Therefore, such model-model miscorrelation in pin access can cause incurable or hard-to-cure design rule violations in routing.

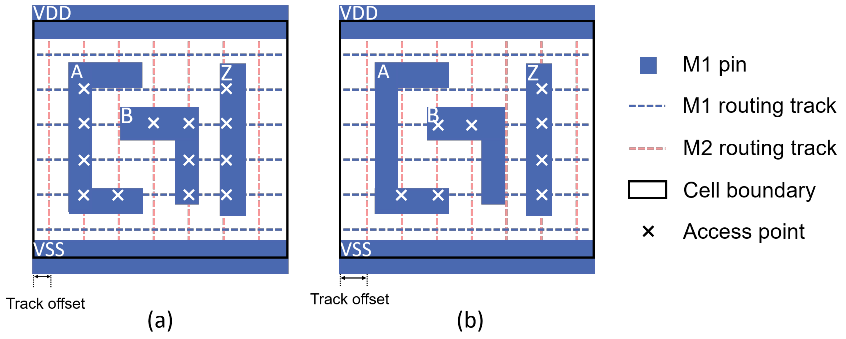


Figure 1.4: Illustration of pin access points with different routing track-placement site offsets.

1.1.2 Missing Building Blocks in the Academic Tool Flow

As noted above, more research efforts are spent on abstracted problems due to ever-more complex design rules. This is especially true for the VLSI routing domain. For example, many research works abstract the routing problem from “pin-to-pin” to “point-to-point”. Although such abstraction helps the researcher focus more on the routing problem itself, making design rule check (DRC) clean pin access in advanced technology nodes is not trivial, and is often left unaddressed in academic tool flows. Without a robust pin access analysis, even if the abstracted routing problem can be solved, it is still difficult to apply the academic tool flow in real-world design contexts due to the potentially dirty pin access. Moreover, leaving such fundamental aspects as pin access open will blur any understanding of “improvement” in academic routing methods.

Another missing piece in most academic tool flows is a DRC engine, which is greatly desired by routing tools. As the design rules become rich and complicated, it is nearly impossible to obtain

DRC-clean routing solutions using a "correct-by-construction" approach. Many modern design rules contain conditional checking which makes "correct-by-construction" impractical. Therefore, in order to achieve a DRC-clean routing solution, any routing tool will require a design rule check engine as one of its necessary building blocks.

1.1.3 Need for Correlated Global-Detailed Routing

A further challenge that arises from the increasing complexity in design rules is also related to routing. Considering the complexity of routing, the overall routing problem is usually divided into two stages – global routing and detailed routing. The idea behind the two-stage approach is to divide and conquer into manageable subproblems. This being said, the two-stage approach is not designed to separate, or even isolate, the global routing and detailed routing problems. However, in the open literature, we can barely find works that address the correlation between global routing and detailed routing, while the number of works that are solely focused on either global routing or detailed routing is considerable. As a result, global routing works tend to over-simplify the global routing problem, because aspects from detailed routing are ignored. Moreover, works that over-simplify the routing problem are usually not applicable in real-world industry contexts, thus making the gap between academia and industry larger.

1.2 This Thesis

To address the new challenges from advanced technology nodes and help narrow the gap between academia and industry, this thesis presents innovative optimization methodologies in detailed placement and routing as well as essential building blocks towards DRC-clean routing. Figure 1.5 illustrates the scope and the organization of this thesis.

To improve model correlation, this thesis presents two detailed placement optimization methodologies: (i) a multi-row detailed placement optimization for neighbor diffusion effect mitigation; and (ii) an in-route pin access-driven placement refinement for improved detailed routing convergence.

To fill in the missing pieces towards DRC-clean routing in academic tool flow, this thesis presents two essential building blocks: (i) a geometry-based design rule check engine; and (ii) a multi-level pin access analysis framework with design rule satisfaction.

To address the global-detailed routing correlation challenge and narrow the gap between academia and industry, this thesis presents TritonRoute-WXL – the open-source global-detailed routing tool that is capable of delivering DRC-clean routing solutions in advanced technology nodes.

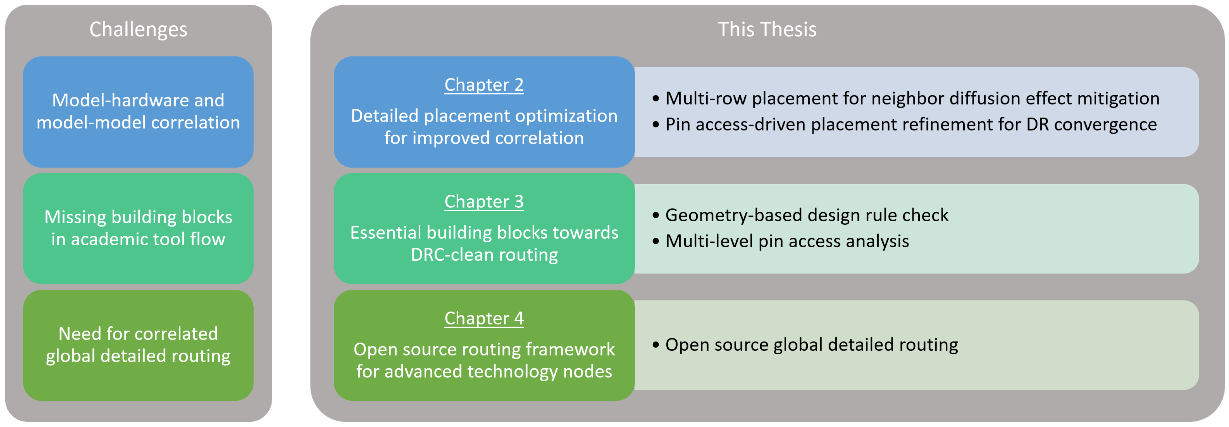

Challenges
Model-hardware and model-model correlation
Missing building blocks in academic tool flow
Need for correlated global detailed routing
This Thesis
Chapter 2
Detailed placement optimization for improved correlation
• Multi-row placement for neighbor diffusion effect mitigation
• Pin access-driven placement refinement for DR convergence
Chapter 3
Essential building blocks towards DRC-clean routing
• Geometry-based design rule check
• Multi-level pin access analysis
Chapter 4
Open source routing framework for advanced technology nodes
• Open source global detailed routing

Figure 1.5: Scope and organization of this thesis.

The remainder of this thesis is organized as follows.

- Chapter 2 presents two detailed placement optimization methodologies for correlation improvement. First, we present a detailed placement optimization methodology for neighbor diffusion effect mitigation and improved model-hardware correlation. Our dynamic programming-based methodology is capable of performing single-row and multi-row detailed placement optimization considering neighbor diffusion effect, HPWL and displacement constraint. Our optimization minimizes *diffusion step* to mitigate neighbor diffusion effect and thus improve model-hardware correlation and yield, using (i) multi-height cell movement, (ii) multi-height cell reordering and (iii) single-height cell movement across standard cell rows. Further, we extend our optimization to potential timing-aware optimization. Second, we present an in-route, pin access-driven detailed placement refinement methodology to improve correlation between placement and routing in terms of pin access, thus improving detailed routing convergence. Our dynamic programming-based methodology is capable of performing single-row detailed placement refinement considering pin access, wirelength, timing and displacement constraint. We further support electrically-equivalent (EEQ) cell swapping for application of our methodology in sub-7nm technology nodes.

- Chapter 3 presents two works that provide essential elements toward DRC-clean routing solutions in advanced technology nodes. First, we present a geometry-based design rule check engine to meet routing tool needs in advanced technology nodes. The design rule checking engine is capable of capturing design rule violations in both academic and foundry technology nodes. The proposed engine includes a violation filtering mechanism that distinguishes between *detailed-routing-fixable* and *non-detailed-routing-fixable* violations, providing accurate design rule violation feedback for ripup-and-reroute in detailed routing. Our engine matches results from a commercial design rule checking tool in sub-14nm foundry enablement. Moreover, the incremental capability in our DRC engine enables further optimization in detailed routing for improved detailed routing convergence. Second, we present a multi-level pin access analysis framework with design rule satisfaction for advanced technology nodes. Our pin access analysis framework consists of cell pin-based access point generation, cell boundary conflict-aware access pattern generation, and dynamic programming-based access pattern selection for instance clusters. Our pin access analysis framework is capable of providing DRC-clean pin access for academic contest benchmark testcases as well as testcases in sub-14nm foundry technology nodes.

- Chapter 4 presents TritonRoute-WXL – a complete, end-to-end routing flow for advanced technology nodes. TritonRoute-WXL is the academia-leading router that consists of an in-memory router database, a global routing engine, a track assignment engine, and a detailed routing engine with integrated pin access analysis engine and design rule check engine. The global and detailed routing engines adopt a region-based ripup-and-reroute methodology. Our router comprehends design rule constraints in industry-standard formats for advanced technology nodes. TritonRoute-WXL delivers unparalleled solution quality for academic contest benchmarks in various technology nodes. For foundry technology nodes, TritonRoute-WXL is capable of delivering DRC-clean routing solutions for sub-14nm technology nodes.

- Chapter 5 concludes the thesis and gives future research directions.

Chapter 2

Detailed Placement Optimizations for Improved Correlation

This chapter presents two detailed placement optimization methodologies for correlation improvement. First, we present a detailed placement optimization methodology for neighbor diffusion effect mitigation and improved model-hardware correlation. Our dynamic programming-based methodology is capable of performing single-row and multi-row detailed placement optimization considering neighbor diffusion effect, HPWL and displacement constraint. Our optimization minimizes *diffusion step* to mitigate neighbor diffusion effect and thus improve model-hardware correlation and yield, using (i) multi-height cell movement, (ii) multi-height cell reordering and (iii) single-height cell movement across standard cell rows. Further, we extend our optimization to potential timing-aware optimization. Second, we present an in-route, pin access-driven detailed placement refinement methodology to improve correlation between placement and routing in terms of pin access, thus improving detailed routing convergence. Our dynamic programming-based methodology is capable of performing single-row detailed placement refinement considering pin access, wirelength, timing and displacement constraint. We further support electrically-equivalent (EEQ) cell swapping for application of our methodology in sub-$7nm$ technology nodes.

2.1 Enhanced Optimal Multi-Row Detailed Placement for Neighbor Diffusion Effect Mitigation in Sub-10nm VLSI

In advanced technology nodes, device behavior no longer depends on independent geometrical parameters [27]. Due to aggressive device scaling, lithography limitations and process complexity, *layout-dependent effect* (LDE) arises from the proximity of devices, and significantly affects device performance. An important type of LDE is *neighbor diffusion effect* (NDE) [11], where the horizontal spacing between diffusion regions changes the performance of transistors. Figure 2.1(a) illustrates different diffusion spacing caused by diffusion height changes between four transistors. If the heights of neighboring diffusion regions are different, there is a diffusion *step*, e.g., transistor *T2* has a diffusion *step* to each of *T1* and *T3*.

More specifically, the drive strength (i.e., I_{on}) and the leakage power (i.e., I_{off}) of a transistor fin is a function of the horizontal spacing to the adjacent diffusion regions of the transistor fin. Since NDE changes the electrical characteristics of transistors, it affects the power, performance and area of designs [11]. For example, Figure 2.1(a) shows the transistor fins A and B with the spacings to their neighboring diffusion area, i.e., d_A and d_B, respectively. As d_A and d_B are different, I_{on} and I_{off} of the two transistor fins are different (e.g., $I_{off}(A) = f(d_A) \neq I_{off}(B) = f(d_B)$) due to the change in V_{th} [11]. For example, given a single inverter with a diffusion *step* next to the PFET and a diffusion *step* next to the NFET, the impacts to the two devices in combination result in higher leakage.

In this work, we use a bimodal assumption to simplify the NDE problem: for a given transistor, either of two leakage values holds, depending on whether the diffusion region on the nearest neighboring site of the transistor has *full height* (that is, same or larger height), or *less height*, compared to the transistor's diffusion height. The leakage difference for the above two cases is linear with #*steps*, e.g., a diffusion height difference of two steps results in $2\times$ leakage difference compared to that of one step. In a conventional place-and-route flow, intra-cell NDE (i.e., NDE effect within a standard cell) is captured by library characterization since the diffusion shapes within a cell are pre-determined. However, it is difficult to capture inter-cell NDE since neighboring diffusion shapes are determined by detailed placement. Thus, in general, library characterization always assumes existence of a full-height neighboring diffusion region on standard cell boundaries, which causes miscorrelation between the model (i.e., library) and the hardware (i.e., actual diffusion shapes at standard cell boundaries and their device leakage impacts) in a design. Minimizing diffusion *steps* in detailed placement is a key idea toward reduction of model-hardware miscorrelation.

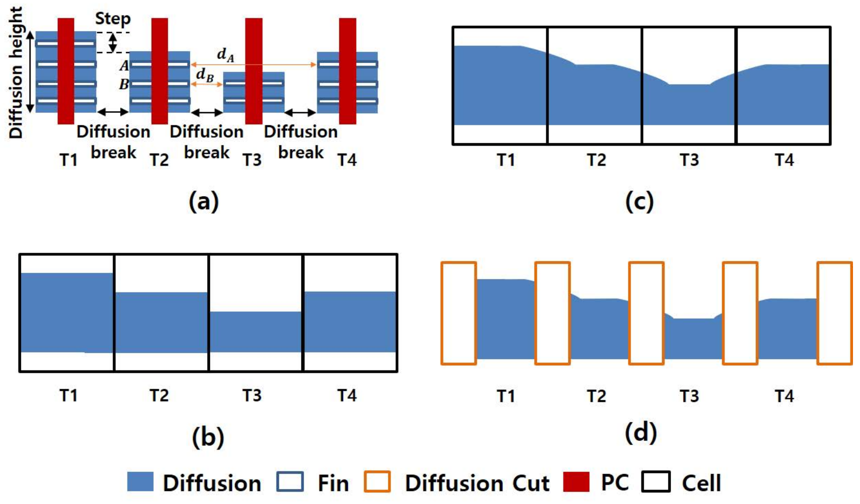


Figure 2.1: (a) Diffusion *step* and fin spacing; (b) desired pattern; (c) actual diffusion region showing corner rounding; and (d) diffusion breaks (after diffusion cuts applied).

With aggressive device scaling, the diffusion *step* not only causes NDE, but also induces a increase in the process complexity due to the limited resolution of conventional 193i lithography. In advanced nodes, the diffusion shapes of transistors are merged and patterned as a single polygon; the transistors are then separated by using diffusion breaks (which are achieved by applying diffusion cuts) [100], as shown in Figure 2.1(a). Figure 2.1(b) illustrates the desired pattern of a single polygon to generate the diffusion regions of four transistors. The actual pattern of the polygon (showing corner rounding in lithography) is shown in Figure 2.1(c). Figure 2.1(d) illustrates the final printed diffusion layout with diffusion cuts. At the boundaries of diffusion where diffusion *steps* exist, fin shapes and diffusion shapes are distorted due to the corner rounding phenomena. A distorted and/or sharp-angled end of a fin may cause an increase in electrical field, resulting in gate oxide breakdown [91]. Further, such distorted diffusion shapes change the diffusion height and fin length, which can cause dramatic shifts in threshold voltage (V_{th}), or even device failure in sub-10*nm* nodes.[1] This V_{th} shift has negative impact on design performance and quality. For example, V_{th} variation can cause setup time and/or hold time violations in a design. As a result, the

[1]According to our collaborator [68], there can be $> 150mV$ V_{th} shift in the 10LPE node.

maximum frequency that the design can achieve is reduced, or the design can even fail with hold time violations due to ultra-low V_{th} which cannot be recovered.

For a motivating study, we define a (inter-cell NDE-induced) *cell failure* to occur if the boundary transistor has a $> 100mV$ V_{th} shift compared to the average V_{th} for all transistors. According to [68], the failure rate of a transistor with a diffusion *step* is twice as high as a transistor without a diffusion *step* (base failure rate). The solid lines in Figure 2.2 show the yield vs. (initial) number of diffusion *steps* ($\sim$ #cells) with different base failure rates. We assume $\#steps$ is approximately proportional to #cells, which holds for testcases in Section 2.1.5. The dashed lines in Figure 2.2 show the projected yield for the same chip if we can reduce 90% of diffusion *steps*. In our preliminary study, more than 60% of standard cells (cell-boundary transistors) have inter-cell diffusion *steps*. For a relatively small design block *VGA* (85% utilization in an N7 (foundry $7nm$) design enablement, 69K cells and 50K diffusion *steps* initially), we assume a base failure rate of 1ppm and can achieve 3.6% yield improvement by removing 90% of diffusion *steps*. For a commercial design with multiple hundreds of millions of cells and diffusion *steps*, if we assume a more realistic 1ppb base failure rate, then we can achieve $\sim 3\%$ yield improvement by removing 90% of diffusion *steps*.[2] In light of this, minimizing diffusion *steps* helps to recover the yield of designs by reducing V_{th} (and thus speed) variation of transistors.

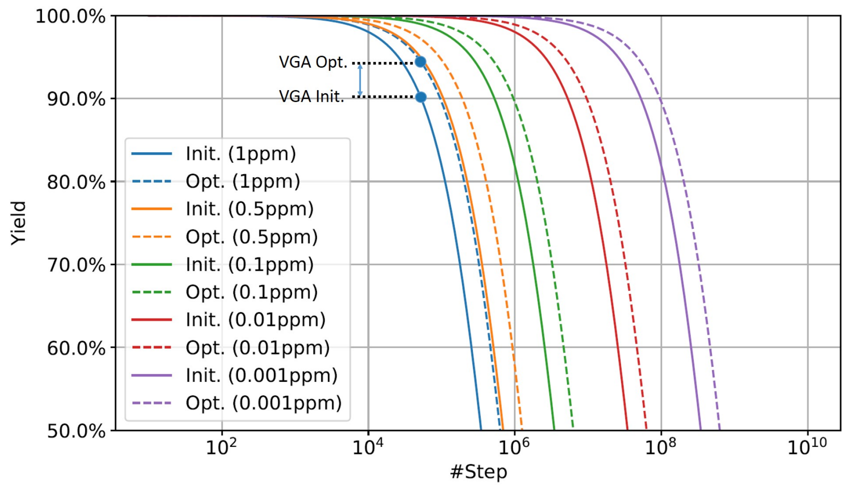


Figure 2.2: Initial (Init.) and projected (Opt.) yield assuming 90% inter-cell *step* reduction for various base failure rates.

[2]Based on guidance from our collaborator [68], after scaling to account for our small testcase sizes, we assume a base failure rate of 1ppm for each *step* in our experiments with small design blocks. See Table 2.4 in Section 2.1.5.

Current limitations and our approach. In order to reduce diffusion *steps*, special non-functional *filler cells* are instantiated between functional cells [74] as we elaborate in Section 2.1.2 below. However, opportunities for *step*-reducing filler cell insertion are limited given a fixed layout, and this approach (effectively similar to cell padding) is expensive in terms of area. Other works [26] [57][92][109] propose graph-algorithmic or dynamic programming methods to resolve complex design rules in advanced nodes. However, the solution spaces considered are typically limited due to the assumption of (ordered)-single-row placement.[3] Recent works [56][97] on multi-row detailed placement involve heuristic approaches, and no advanced-node rules are considered. Han et al. [36] propose an optimal single-row and double-row dynamic programming for detailed placement optimization, allowing cell reordering with support of double-height cells.

In this work, we extend our previous single-row and double-row detailed placement framework [36] with HPWL-awareness and with multi-row detailed placement optimization. Our main contributions are summarized as follows.

- We extend the optimal single-row dynamic programming-based approach [36] to an HPWL-aware version. The proposed approach minimizes and balances diffusion *steps* and HPWL cost. Our proposed algorithm is capable of all types of cell movements – i.e., cell variants, relocating, and reordering (specifically, *P-reordering* with $P > 2$).

- We propose a new multi-row dynamic programming, with support of movable, and fully-reorderable, multi-height cells, including reordering between multi-height cells. Inter-row cell moving within each optimization window (in multiple of rows) is intrinsically supported, and further improves solution quality.

- We propose metaheuristics to use both single-row HPWL-aware optimization and multi-row optimization to achieve better solution quality.

- We extend our formulation to a potential timing-aware optimization that leads to $6\times$ increase in *intentional* steps around timing-critical cells to improve the timing performance.

[3]Lin et al. [57] propose a *P-reordering* problem. However, only *2-reordering* (i.e., neighbor cell switching) is presented. We describe our methodology to handle the *P-reordering* problem in Section 2.1.2.

- We improve the solution quality over [36] by achieving up to 98% inter-cell diffusion *step* reduction compared to 90% achieved in [36], while consuming similar runtime.

The remainder of this work is organized as follows. Section 2.1.1 reviews related works. Section 2.1.2 describes the problem formulation and dynamic programming-based single-row detailed placement methodology. Section 2.1.3 describes the double-row detailed placement flow. Section 2.1.4 describes the multi-row detailed placement flow. In Section 2.1.5, we describe our experimental setup and results. Section 2.1.6 gives conclusions and directions for ongoing work.

2.1.1 Related Work

We classify relevant previous works on detailed placement into three categories: (i) detailed placement for advanced nodes, (ii) mixed cell-height placement, and (iii) NDE-aware detailed placement.

Detailed placement for advanced nodes. To support complex design rules introduced in advanced nodes, the objectives of detailed placement have changed from classical objectives (e.g., wirelength reduction [42][44][46][48][55][77]) in recent years. The works of [57][92][109] resolve triple-patterning issues. Yu et al. [109] propose shortest path and dynamic programming algorithms to solve the ordered single row (OSR) placement. Tian et al. [92] develop a weighted partial MAX SAT approach to solve the OSR problem. Lin et al. [57] propose a local reordered single row refinement (LRSR) and implement a 2-reordering (i.e., neighboring cell switching) approach using a unified graph model. Du and Wong [26] apply a shortest-path algorithm supporting flipping and 2-reordering to address the drain-drain abutment problem in FinFET-based cell placement. The works of [20][38] propose mixed integer linear programming (MILP)-based methods to comply with drain-drain abutment, minimum implant area and minimum oxide jog length rules, and to increase vertical M1 connections.

Mixed cell-height placement. Wu et al. [97] propose a pairing technique to handle double-height cells for detailed placement. Their method simply groups or inflates cells so that all cells become double-height cells, after which a conventional detailed placer can be used. Recently, Lin et al. [56] have proposed a *chain move* scheme along with a nested dynamic programming-based approach to support multiple cell-height placement. They first perform chain moves to save wirelength cost. On top of this, dynamic programming is applied to solve the nested shortest path problem. Other techniques [23] are developed to support non-integer-ratio (e.g., mixture of 8T and 12T cells) mixed cell-height placement.

NDE-aware placement. Ou et al. [75] perform NDE-aware analog placement by modifying and integrating a compact model for NDE into an existing analog placement algorithm. Oh et al. [74] develop special filler cells to mitigate NDE.

Han et al. [36] (which this work builds on) propose to resolve the NDE problem in the detailed placement stage. Inter-cell diffusion *steps* are minimized by trying to match the diffusion heights of neighboring cells. If two neighboring cells have different diffusion heights, special filler cells can be inserted to reduce diffusion *steps*. [36] proposes single-row and double-row dynamic programming optimizations that support cell relocating, reordering and flipping as well as double-height cells. They support reordering between single-height cells, and between a single-height cell and a double-height cell, but not between two double-height cells.

In summary, many works such as [26][57][92][109] propose graph or dynamic programming models to resolve complex design rules in advanced nodes. However, their solution spaces are limited by the assumption of (ordered)-single-row placement. Two recent works [56][97] on multi-row detailed placement give heuristic approaches, but no advanced node rules are considered. Our previous work [36] proposes dynamic programming-based methods to optimize single-row and double-row placements, systematically supporting cell reordering and double-height cells. However, the dynamic programming formulation cannot be extended to support more than two rows, and the formulation cannot support reordering between two double-height cells. Notably, our present work advances over [36], and is distinguished from previous approaches, in several ways. (i) We formulate an optimal (HPWL-aware) single-row and multi-row dynamic programming-based approach to minimize a cost function that includes diffusion *steps*. (ii) We support a richer set of cell movements than in previous works – i.e., flipping, relocating *and* reordering – via a systematic methodology to handle *P-reordering* with $P > 2$. Specifically, our multi-row approach intrinsically supports *inter-row* cell relocation. (iii) Our formulation supports multi-height cells with movable, and fully-reorderable, multi-height cells.

2.1.2 Single-Row Optimization

In this section, we describe the problem statement and our dynamic programming formulation for single-row detailed placement.

Single-Row Optimization Problem. *Given an initial legalized single-row placement, perturb the placement to minimize inter-cell diffusion* steps.

Inputs: A legalized single-row placement, available cell variants, and cost function of a diffusion *step*.

Output: Optimized single-row detailed placement with minimized overall cost (including inter-cell diffusion *steps*).

Constraints: Maximum displacement range, maximum reordering range, availability of cell flipping.

Filler Cell and Step Costs

Table 2.1: Cost for one diffusion *step*.

Spacing (sites)	0	1	2	3	4+
Cost	1	$+\infty$	1	1	0

Table 2.1 describes inter-cell diffusion *step* cost. For each pair of adjacent cells, if there are zero, two or three empty sites in between, the cost is equal to the number of inter-cell diffusion *steps*; if there are at least four empty sites in between, the cost is always zero. That is, with four or more empty sites we can always assume proper filler cell insertions resulting in no inter-cell diffusion *steps*. Figure 2.3 shows an example of filler cell insertion between two functional cells that have different diffusion heights at edges that face each other. If the two functional cells have fewer than four empty sites in between, filler cells can only match one of the diffusion heights. As a result, there always exists at least one diffusion *step* that affects one of the two functional cells. However, with a spacing of four or more sites, a legal diffusion height transition can always be achieved by one or more contiguous filler cell(s). Thus, the filler cell(s) can match both the diffusion heights of the two functional cells. In a relevant advanced technology, the minimum filler cell width is two placement sites due to process limitations. Therefore, adjacent functional cells must abut, or have at least two empty sites between them, in order to insert a

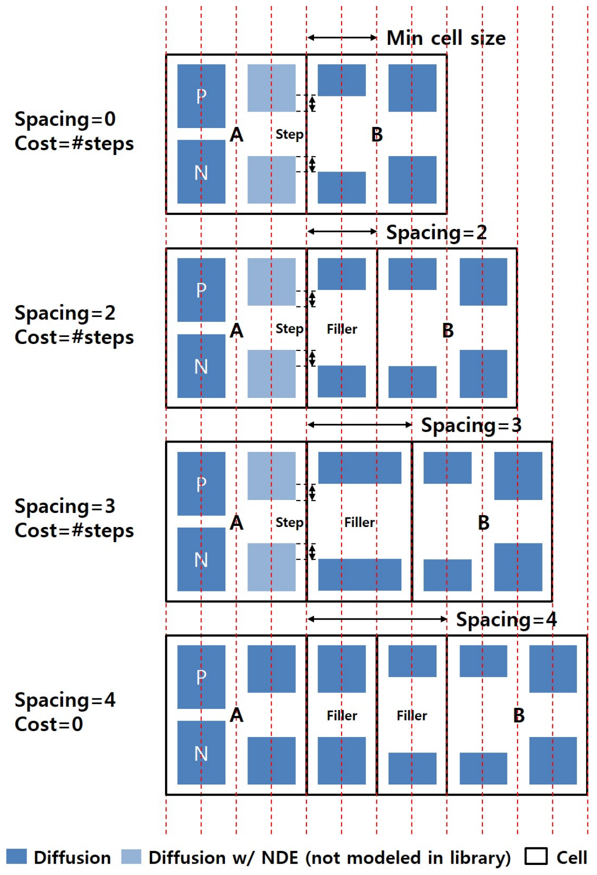


Figure 2.3: Filler insertion between cell A and B, given different spacings.

filler cell [68]. In our implementation, we avoid single-site spacings by assigning infinite cost to such scenarios, as indicated in Table 2.1. Even though our optimization does not explicitly allocate white space, the dynamic programming (presented later in Sections 2.1.2, 2.1.3 and 2.1.4) itself can utilize/change the local white space distribution by cell relocating and cell reordering within specified ranges. Filler cell cost is explicitly included in our dynamic programming cost calculation, such that our optimization is aware of both whitespace and filler insertion as it trades off between (i) abutting two cells without filler insertion at the cost of diffusion *steps*, and (ii) leaving four placement sites for a proper filler insertion in an effort to minimize the diffusion *steps* between neighboring cells.

Table 2.2: Notations.

Notation	**Meaning**
C	set of cells in a window of initial placement
c_k	k^{th} cell in the left-to-right ordered initial placement, i.e., k is the cell index
v	a cell variant
$w_{k,v}$	width of c_k with a variant v
$[-x_\Delta, x_\Delta]$	horizontal displacement range
x_k	absolute x coordinate of c_k in the initial placement, in units of placement sites
l	displacement from the initial placement, in units of placement sites
$[-r, r]$	reordering range
i	number of placed cells
j	position shift from the initial placement
s	placement status array
$d[i][j][v][l][s]$	minimum cost when i cells are placed with case (j,v,l,s)
The notations below apply only to multi-row optimization	
$[-y_\Delta, y_\Delta]$	vertical displacement range
y_k	absolute y coordinate of c_k in the initial placement, in units of rows
m	number of rows in an optimization window
b	row index in an optimization window
d_b	for the b^{th} row, d_b is the distance between the rightmost boundary of b^{th} row, and the rightmost boundary of all rows in the optimization window
D	distance array of d_b in an optimization window (i.e. $[d_0...d_{m-1}]$)
t_b	for the b^{th} row, type of the rightmost cell (e.g., 2-fin, 3-fin or 4-fin)
T	type array of t_b in an optimization window (i.e., $[t_0...t_{m-1}]$)
$\{D, T\}$	boundary condition
$[D][T]$	forming boundary condition $\{D, T\}$
$d[i][j][v][l][s][D][T]$	minimum cost when i cells are placed, forming boundary condition $\{D, T\}$

Notations

Table 2.2 shows notations used in our formulation. For each cell c_k, **cell index** k is its (left-to-right) sequentially ordered position in the initial placement. Given a set of cells (C) in a row of an initial placement, the leftmost cell is c_1, and the rightmost cell is $c_{|C|}$.

For each c_k, we define **cell variants** (v) which correspond to different cell orientations and cell layouts with the same functionality. To minimize #diffusion *steps*, we can use several variants of a cell with the same functionality, for which layouts have different diffusion heights. In our experiments below, $v = 0$ indicates the cell orientation in the initial placement, and $v = 1$ indicates the flipped (i.e., mirrored about the y-axis) cell orientation. $w_{k,v}$ is the width of cell c_k with variant v, in units of placement sites. Flipping a cell does not change the set of sites that the cell occupies.

We define the **displacement range** $[-x_\triangle, x_\triangle]$ as the constraint that a cell cannot move more than $x_\triangle$ sites from its initial placement. We use x_k to denote the initial right x coordinate of c_k, in units of placement sites. Thus, c_k can be placed with its right x coordinate in the interval $[x_k - x_\triangle, x_k + x_\triangle]$. We use l to denote the **displacement** (in sites) from the initial cell placement (i.e., $l \in [-x_\triangle, x_\triangle]$). For the cells on the boundary of the die, we make sure that the displacement range will not extend beyond the die boundary.

We support cell reordering with a **reordering range** $[-r, r]$, i.e., given r, in the placement solution c_k can have a new sequentially ordered position within the range $k - r, k - r + 1, \ldots, k + r$.

In our dynamic programming, we place one cell at a time from left to right, and the index i is used to indicate that i cells have been placed. Given a cell reordering range $[-r, r]$, cells c_k with $k < i - r$ are placed; those with $i - r \leq k \leq i + r$ may or may not be placed; and those with $k > i + r$ are not placed. For the $2r + 1$ cells such that $i - r \leq k \leq i + r$, we use a binary array s to denote the *placement status* of each cell. Here, s is a binary array of size $(2r + 1)$, i.e., $s \in \{0, 1\}^{2r+1}$. Each bit in the array indicates whether the corresponding cell is placed or not. For example, if we have six cells c_1 to c_6, $i = 4$ and $r = 1$, then s captures the placement status of the $(2 \cdot 1 + 1 = 3)$ cells c_3, c_4 and c_5. $s = [0, 1, 1]$ means that c_3 is not placed, while c_4 and c_5 are placed. Figure 2.4 illustrates six placement solutions with three legal states when $i = 4$. In this example, c_1 and c_2 must be placed and c_6 must not be placed. We note that the indices of s correspond to k (position in the initial placement), but not the final position. For example, $s[0]$ always represents the status for c_3, and $s[2]$ always represents the status for c_5, regardless of the actual sequence of positions, as shown in Figure 2.4(b). Also, when we have placed i cells, since cells with index $k < i - r$ must be placed, we must have placed $i - (i - r - 1) = r + 1$ cells that have cell index $i - r \leq k \leq i + r$. Thus, at all times, a legal status array s has exactly $r + 1$ elements equal to 1. In the above example, s always has $1 + 1 = 2$ elements equal to 1.

Given i, to identify the last placed cell c_k (that is, the i^{th} cell to have been placed), we define the **position shift** as j, where $k = i + j$. For example, in Figure 2.4(c), given $i = 4$, the position shift $j = -1$ indicates that the last placed cell is c_3, since $3 = 4 + (-1)$.

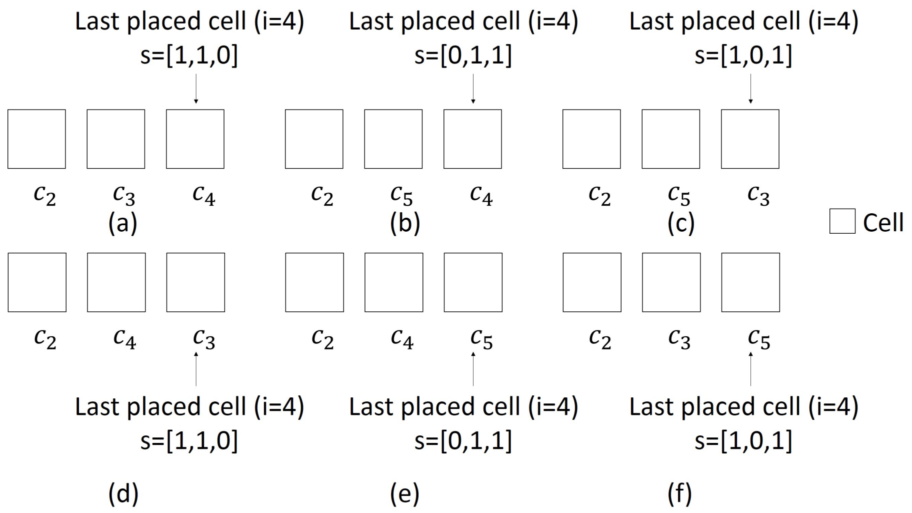


Figure 2.4: Illustration of six placement solutions with three legal states given $i = 4$ and $r = 1$.

At the heart of our dynamic programming recurrence, we use $d[i][j][v][l][D][T][s]$ to represent the minimum cost when i cells have been placed. Note that in single-row case, the dimensions of D and T are both zero. Therefore, the dynamic programming array can be reduced to $d[i][j][v][l][s]$. From this array, we can obtain the last placed cell c_k, where $k = i + j$. We can also tell the variant v in use, the displacement l, and the status s for cell c_k. We define the above as *case* (j, v, l, s), with i implicitly given, for simplicity. Therefore, we complete the row placement once we reach $i = |C|$, and we obtain the optimal solution by finding the minimum cost among all cases of $i = |C|$. In our implementation, we store a pointer for each entry in the DP array so that the optimized placement can be traced back from $d[|C|][j][v][l][s]$ all the way to $d[0][j][v][l][s]$.

Dynamic Programming Formulation

Algorithm 1 describes our dynamic programming (DP) procedure for single-row placement in detail. Line 2 initializes the DP solution array. Lines 3 – 13 describe the main algorithm. Starting with placing the first cell, the algorithm incrementally adds (places) cells next to the current partial placement solution. Procedure $getNext()$ returns a list of legal $next$ cells and the respective status of each of these cells. Along with legal (j', s') from Line 5, Line 6 checks all possible cases (v', l') considering placement legality and displacement constraints, as shown in Equation (2.1). Lines 7 – 9 update the minimum cost for the case (j', v', l', s') when we place the $i' = (i+1)^{st}$ cell. In Lines 14 – 17, we obtain the minimum cost among all legal cases when $i = |C|$, and Line 18 returns the minimum cost for the current row.

$$x_{i+j} + l + w_{i+j,v} \leq x_{i'+j'} + l' \tag{2.1}$$

The function $cost(^{i',j',v',l'}_{i,j,v,l})$ calculates the cost as a weighted sum of (i) diffusion *step* cost, (ii) displacement cost, and (iii) cell variant cost, as shown in Equation (2.2). The diffusion *step* cost is calculated as total #inter-cell diffusion *steps* between the i^{th} and $(i')^{th}$ placed cells. The displacement cost is equal to the absolute value of l'. In this work, we assume that the given initial placement solution has adequate quality in terms of various metrics, including but not limited to pin accessibility, global routability, etc. Thus, we simplify other optimization objectives as one "displacement minimization" objective. As noted above, in this work we assume two cell variants: original orientation and flipped orientation. We set the variant cost to one if a cell is flipped ($v' = 1$), and zero otherwise. Two weighting factors α and β (β can be seen as supplementing α by capturing an equivalence between cell flipping and displacement) are used to balance the three cost terms. We describe experiments regarding the impact of weighting factors in Section 2.1.5.

$$cost(^{i',j',v',l'}_{i,j,v,l}) = cost_{step} + \alpha \cdot cost_{disp} + \alpha \cdot \beta \cdot cost_{var} \tag{2.2}$$

Algorithm 2 details our methodology to obtain $next$ status. That is, given the binary status array for i, we construct the status array for $i' = i+1$. Line 2 initializes the list of next available $(cellIndex, status)$ combinations. In Line 3, we first shift the status array for i one bit to the left to obtain the cell placement status for $i' = i+1$. Then, Lines 4 – 9 check whether cell $c_{i'-r}$ must be placed as the $(i')^{th}$ cell. If we do not place $c_{i'-r}$ as the $(i')^{th}$ cell, then cell $c_{i'-r}$ will be placed out of its reordering range. Thus, we set $s[-r] = 1$ and return so that we make sure to choose $c_{i'-r}$ as the $(i')^{th}$ cell. Lines 10 – 16 check whether any binary indicator $s[m]$ is equal to zero. If so, $c_{i'+m}$ could be the next legally placed cell. In such a case, we add $(m, nextStatus)$ to the list.

Algorithm 1 Dynamic programming (single-row)

```
1:  **Initialize for all legal cases** $(j, v, l, s)$
2:    $d[0][j][v][l][s] \leftarrow 0, d[i][j][v][l][s] \leftarrow +\infty, (0 < i \leq |C|)$
3:  **for all** $i = 0$ to $|C| - 1$ **do**
4:    **for all** $d[i][j][v][l][s] \neq +\infty$ **do**
5:      **for all** $(j', s') \in$ getNext$(s)$ **do**
6:        **for all** $(v', l')$ **do**
7:          $i' = i + 1$
8:          $t \leftarrow d[i][j][v][l][s] + cost(^{i',j',v',l'}_{i\ ,j\ ,v\ ,l})$
9:          $d[i'][j'][v'][l'][s'] \leftarrow \min(d[i'][j'][v'][l'][s'], t)$
10:       **end for**
11:     **end for**
12:   **end for**
13: **end for**
14: $finalCost \leftarrow \infty$
15: **for all** $(j, v, l, s), i = |C|$ **do**
16:   $finalCost \leftarrow \min(d[|C|][j][v][l][s], finalCost)$
17: **end for**
18: **Return** $finalCost$
```

HPWL-Aware Optimization

We mitigate the wirelength impact of single-row *step* optimization by modifying the cost function. Specifically, we add a ΔHPWL cost component to the function $cost(^{i',j',v',l'}_{i\ ,j\ ,v\ ,l})$, as shown in Equation (2.3).

$$cost(^{i',j',v',l'}_{i,j,v,l}) = cost_{step} + \alpha \cdot cost_{disp} + \alpha \cdot \beta \cdot cost_{var} + \gamma \cdot cost_{\Delta HPWL} \tag{2.3}$$

We calculate the $cost_{\Delta HPWL}$ by summing up the ΔHPWL contribution of cell c_k over all nets incident to c_k, in the same way as in [48]. $cost_{\Delta HPWL}$ captures the impact of a cell's placement on bounding box sizes of incident nets. We use a new weighting factor γ to balance the four cost terms. We describe experiments regarding the impact of weighting factors in Section 2.1.5.

2.1.3 Double-Row Optimization

In this section, we describe the problem statement and the dynamic programming approach for *double-row detailed placement considering double-height cells as well as reordering, flipping and available cell variants*.

Algorithm 2 Procedure $getNext$ (single-row)

```
1: Inputs: $s$
2: Initialize $nextList \leftarrow \emptyset$
3: $s \leftarrow$ shiftLeft1Bit($s$)
4: if $s[-r] = 0$ then
5:     $s[-r] \leftarrow 1$
6:     $nextStatus \leftarrow s$
7:     $nextList \leftarrow nextList \cup \{(-r, nextStatus)\}$
8:     Return $nextList$
9: end if
10: for all $m \in [-r, r]$ do
11:     if $s[m] = 0$ then
12:         $nextStatus \leftarrow s$
13:         $nextStatus[m] \leftarrow 1$
14:         $nextList \leftarrow nextList \cup \{(m, nextStatus)\}$
15:     end if
16: end for
17: Return $nextList$
```

Double-Row Optimization Problem. *Given an initial legalized double-row placement with double-height cells, perturb the placement within each row to minimize inter-cell diffusion* steps.

Inputs: Legalized double-row placement, available cell variants, and cost function of a diffusion *step*.

Output: Optimized double-row detailed placement with minimized overall cost (including inter-cell diffusion *steps*).

Constraints: Maximum displacement range, maximum reordering range, availability of cell flipping.

Assumptions

We make the following assumptions with respect to this problem statement.

Assumption 1. *Cell rows can be fully separated from each other every two consecutive rows.* In the case of placement rows that contain only single-height cells, the assumption is correct by definition. However, for any cell row, a double-height cell that occupies sites in the row must span to either the upper neighboring row or the lower neighboring row, but not both. Figure 2.5(a) shows such separable pairs of cell rows, where rows 1 and 2 (with double-height cells A and B) do not interfere with rows 3 and 4 (with double-height cells C and D). By contrast, in Figure 2.5(b), row 2 has double-height cells E and F which interfere with both row 1 and row 3, violating our assumption. Given the interleaving of VDD/VSS power rails in modern libraries, our assumption is normally satisfied. In other words, all double-height cells in the current technology node tend to have the same power rail configuration. (In Figure 2.5(b), cell F has a different type of power rail design (VDD-VSS-VDD) than the other double-height cells (VSS-VDD-VSS).) We do not have such double-height library cells in the current technology node.[4]

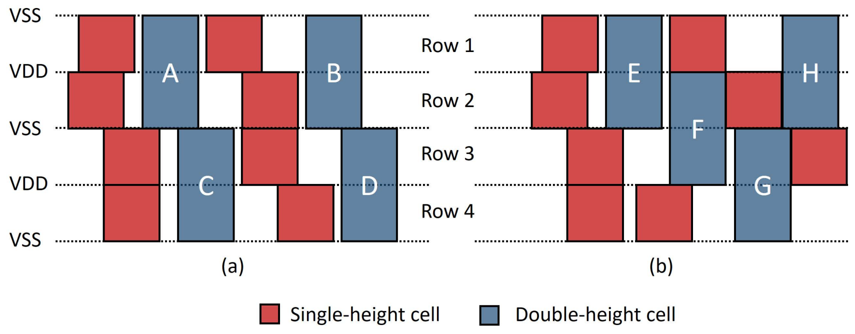


Figure 2.5: Illustrations of double-height cells in placement rows. (a) Separable pairs of cell rows, reflecting power rail design of double-height cells in current N10 libraries. (b) Non-separable pairs of cell rows.

Assumption 2. *The relative positions among double-height cells are fixed.*[5] For two double-height cells A and B, if A is initially to the left of B ($x_A < x_B$), then we require that in our final placement, c_A remains to the left of c_B. We note that we still allow reordering between a single-height cell and a double-height cell (thus, the double-height cells are *partially reorderable*) so as to maximize the *steps* reduction.

Formulation

Given the above assumptions, our approach can provide optimal placement solutions for two consecutive rows sharing common double-height cells as in Figure 2.5(a). Overall, double-row optimization uses single-row optimization as a basic building block. From each double-height cell, we invoke separate single-row optimizations that progress left-to-right in each of the two rows, and merge the solutions once

[4]Our collaborator [68] at a major advanced foundry indicates that all double-height cells have only one power rail configuration in the 10LPE node. Cells with height of four or more rows account for less than 1% of all instances, and thus our formulation can be easily adopted if we just assume that these very large (height $\geq$ four rows) cells are fixed.

[5]The double-height cell effectively breaks the two rows into separate optimization regions, wherein we invoke single-row optimization separately for the two rows. The prerequisite is that we know exactly what instance is the "next double-height cell", which requires that relative positions be unchanged for double-height cells. This assumption will be lifted below in Section 2.1.4.

Algorithm 3 Dynamic programming (double-row)

```
1:  Initialize $DHCellList \leftarrow getOrigDHOrdering()$
2:  Initialize costs for all legal CASES $(v, l, j_0, s_0, j_1, s_1)$
3:    $D[0][v][l][j_0][s_0][j_1][s_1] \leftarrow 0$
      $D[I][v][l][j_0][s_0][j_1][s_1] \leftarrow +\infty, (0 < I \leq |DHCellList| + 1)$
4:  for all $I = 0$ to $|DHCellList|$ do
5:    for all $D[I][v][l][j_0][s_0][j_1][s_1] \neq +\infty$ do
6:      for all legal $(v', l', j'_0, s'_0, j'_1, s'_1)$ do
7:        $I' = I + 1$
8:        $t \leftarrow D[I][v][l][j_0][s_0][j_1][s_1] + Cost(^{I',v',l',j'_0,s'_0,j'_1,s'_1}_{I\ ,v\ ,l\ ,j_0,s_0,j_1,s_1})$
9:        $D[I'][v'][l'][j'_0][s'_0][j'_1][s'_1] \leftarrow$
            $\min(D[I'][v'][l'][j'_0][s'_0][j'_1][s'_1], t)$
10:     end for
11:   end for
12: end for
13: for all $(v, l, j_0, s_0, j_1, s_1)$ when $I = |DHCellList|$ do
14:   $sol \leftarrow \min(D[I][v][l][j_0][s_0][j_1][s_1], sol)$
15: end for
16: Return $sol$
```

we encounter the next double-height cell. The merging is designed to preserve all optimal candidates, while enabling movable and partially reorderable double-height cells. Our development is similar to that of Algorithm 1, where we saw that given the minimum costs of all *cases* (j, v, l, s) for i, we could derive the minimum costs of all *cases* (j', v', l', s') for $i' = i + 1$. Now, let us extend the definition of $case$ to support double-row placement when double-height cells span the two rows, row 0 and row 1. We define $CASE$ $(v, l, j_0, s_0, j_1, s_1)$ given I, where I is the number of placed *double-height* cells. Subscripts 0 and 1 refer to row 0 and row 1, respectively. In Algorithm 1, we obtain the last placed cell c_k from i and j. Here, in double-row optimization, we know exactly the last placed double-height cell because of **Assumption 2**, and we would like to obtain i_0 and i_1 (number of cells placed in row 0 and row 1, respectively) since the formulation allows reordering between a single-height cell and a double-height cell, i.e., a single-height cell may be relocated to the left of the double-height cell even if that single-height cell was originally to the right of the double-height cell. Given the double-height cell's initial position k_0 in row 0 and k_1 in row 1, we have $i_0 = k_0 - j_0$ and $i_1 = k_1 - j_1$. The values of v, l, s_0 and s_1 can be obtained directly from *CASE*.

We give a precise description of our double-row dynamic programming in Algorithm 3. Line 1 obtains the double-height cell sequence from the initial (i.e., input) two-row placement. We note that two

Algorithm 4 $Cost$ (double-row)

```
1: Inputs: $I, v, l, j_0, s_0, j_0, s_0, I', v', l', j'_0, s'_0, j'_1, s'_1$
2: $k_0 \leftarrow getK(I, 0)$, $k_1 \leftarrow getK(I, 1)$
3: $k'_0 \leftarrow getK(I', 0)$, $k'_1 \leftarrow getK(I', 1)$
4: $i_0 \leftarrow k_0 + j_0$, $i_1 \leftarrow k_1 + j_1$
5: $i'_0 \leftarrow k'_0 + j'_0$, $i'_1 \leftarrow k'_1 + j'_1$
6: $d_0 \leftarrow optSR_0(^{i'_0, j'_0, v', l', s'_0}_{i_0, j_0, v, l, s_0})$
7: $d_1 \leftarrow optSR_1(^{i'_1, j'_1, v', l', s'_1}_{i_1, j_1, v, l, s_1})$
8: $totCost \leftarrow d_0 + d_1$
9: Return $totCost$
```

virtual double-height cells are added to “pad” the input at the start and at the end of the placement rows, respectively. Lines 2 – 3 initialize the DP solution array. The array only has entries for double-height cells, and records all solutions (costs) $D[I]$ when we have placed the I^{th} double-height cell. Lines 4 – 12 are the heart of the algorithm. Starting with the (left) virtual double-height cell, the algorithm incrementally places double-height cells and updates minimum costs from all *CASES* in $D[I]$ to all *CASES* in $D[I + 1]$ assuming we have placed I double-height cells. In Lines 13 – 15, we obtain the minimum cost among all legal *CASES* when we reach the ending (right) virtual cell ($I = |DHCellList|$), and Line 16 returns the minimum cost for the two rows.

Algorithm 4 describes the cost function in our double-row DP. Line 2 retrieves the double-height cell position in the initial placement for each of the rows. Line 3 gets the next double-height cell similarly. Line 4 obtains the numbers of cells (i_0 and i_1) that have been placed for the two rows. And, Line 5 obtains the numbers of cells (i'_0 and i'_1) that we must place by the time we reach the next double-height cell. For example, for row 0, we need to place cells starting from the *case* (j_0, v, l, s_0) with i_0, until we reach the *case* (j'_0, v', l', s'_0) with i'_0. The above can be achieved by $optSR$ – a modified version of the single-row dynamic programming. In $optSR$, we make sure that we do not place any double-height cells other than $c_{i'_0}$. Thus, Assumption 2 is maintained. In Lines 8 and 9, we return the two-row sum of costs.

We highlight the fact that in our implementation, given the starting *case* (j, v, l, s) with i, $optSR$ calculates all minimum costs of *case* (j', v', l', s') with i', where $k' = i' + j'$, within one functional call to our single-row DP. With this, the number of calls to single-row DP is proportional only to #*cases*, rather than to #*CASES*.

2.1.4 Multi-Row Optimization

In this section, we generalize from the single-row dynamic programming, and describe our approach for *multi-row detailed placement*, with support of fully-reorderable multi-height cells and inter-row cell relocating.

Multi-Row Optimization Problem. *Given an initial legalized multi-row placement, perturb the placement across the multiple rows to minimize inter-cell diffusion* steps.

Inputs: Legalized multi-row placement, available cell variants, and yield cost function.

Output: Optimized multi-row detailed placement with minimized overall cost (including inter-cell diffusion *steps*).

Constraints: Maximum horizontal displacement range, maximum vertical displacement range, maximum reordering range and availability of cell flipping.

Preliminaries

Similar to double-row optimization, we optimize m consecutive rows together (as a single optimization *window*) in multi-row optimization. In an optimization window, we move the cells according to our algorithm assuming that cells outside the window are fixed. Different windows are optimized separately. However, compared to the double-row optimization in Section 2.1.3, we do not require the relative positions among double-height cells to be fixed. Instead, a double-height cell can be reordered with another double-height cell as long as they are within the reordering range. Moreover, in contrast to Section 2.1.3's double-row optimization, where a cell cannot move outside its original cell row, here we allow a cell to move freely within a given vertical displacement range (in units of placement rows), enabling a larger solution space to minimize diffusion *steps*.

In single-row and double-row optimization, where only intra-row relocating and reordering are allowed, the initial cell ordering (c_k in Table 2.2) is defined within each row from the initial (input) placement. To enable a unified multi-row reordering range, with support of inter-row relocating and reordering, we redefine the original cell ordering as follows:

Definition. *Given an m-row initial (input) placement, cells in all m rows are left-to-right ordered according to their rightmost boundary, in a unified one-dimensional array, e.g., c_1, c_2, ..., c_k. If cells in the initial placement have the same x coordinate for their right boundary, we break ties using the y coordinate of their lower boundary.*

Figure 2.6 shows an example of sequential cell ordering for a two-row initial placement. We note that cells c_4 and c_5 could have their positions exchanged in the ordering, regardless of their left boundary. However, as mentioned, in our implementation tie-breaking is by descending order of y coordinate.

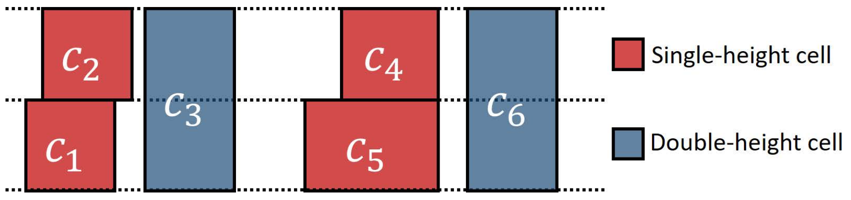


Figure 2.6: An example of multi-row cell ordering. Cells are sequentially ordered (c_1 to c_6) according to the x coordinate of their right boundary. Cells c_4 and c_5 have the same right boundary x coordinate, and thus could be switched in the ordering.

With the above redefined cell ordering, reordering range works the same way as in Section 2.1.2. The new sequentially ordered position is determined by the new x coordinate (in the final solution) of the right boundary of each cell. The difference between the original and the new sequentially ordered position should be always within the reordering range. In the multi-row optimization, given the above redefined cell ordering, our dynamic programming still seeks to place one cell at a time, from left to right. The left-to-right placement procedure then induces the following assumption:

Assumption. *The x coordinate of the right boundary of the $(i+1)^{st}$ cell must be greater than or equal to the right boundary of the partial placement consisting of i cells (i.e., placement boundary).*

Given the definition, the assumption does not reduce the solution space. For example, in Figure 2.7, assuming a partial placement of c_2 and c_1, if the 3^{rd} cell to be placed is c_3, and we would like its right boundary to be to the left of the placement boundary, then we can always get to such a partial placement solution from a partial placement of c_2 and c_3, followed by placement of c_1.

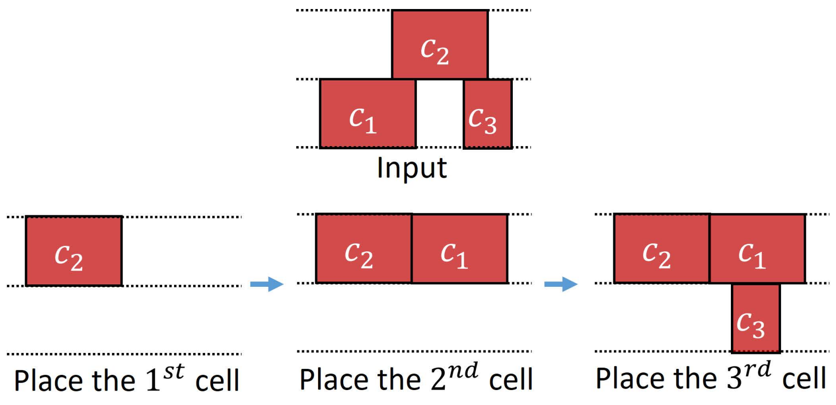


Figure 2.7: Illustration of the placement boundary assumption.

Formulation

Given the above assumption, our approach will find an optimal placement solution for a given optimization window of m rows containing multi-height cells. We illustrate the multi-row dynamic programming-based detailed placement in Figure 2.8(a). We use type array $T = \{t_0, ..., t_{m-1}\}$ to describe the type, i.e., 2-fin, 3-fin or 4-fin configuration, of the rightmost cell in each row. Initially, each entry of T is an initial virtual cell, indicating that the placement boundary for all rows is the left boundary of the die, and that there will be no diffusion *step* penalty applied to any type of cell immediately to the right of this boundary. We also use distance array $D = \{d_0, ..., d_{m-1}\}$ to describe the shape of the placeable region as shown in Figure 2.8(b). The subproblems solved in the DP are of form: place $|C| - i$ cells in the placeable region defined by a partial placement with i cells.

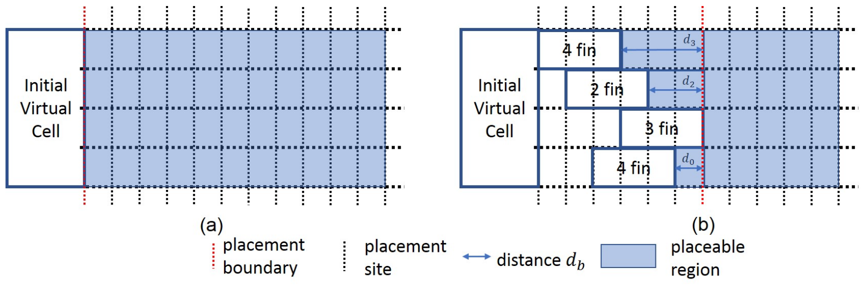


Figure 2.8: Illustration of DP in multi-row placement with $m = 4$.

We give a precise description of our multi-row dynamic programming in Algorithm 5. Note that the numbers of entries of distance array D and cell type array T are both $m - 1$ because the distance from the last placed cell to the placement boundary is always zero, and the cell type of the last placed cell can be retrieved by cell variant v. Lines 1 – 3 initialize the DP solution array. Lines 4 – 15 describe the main algorithm. Compared to single-row dynamic programming, we have one more iteration over all placement rows in an optimization window, subject to the maximum vertical displacement range constraint. Effectively, the multi-row DP array is different from single-row DP array in that it is capable of storing multiple intermediate placement solutions given the same cell ordering and horizontal displacement, as long as these solutions have different type (T) or distance (D) arrays. Also, Line 11 updates distance array D and cell type array T according to the choice of placement row b'. Lines 16 – 19 obtain the optimal solution among all legal cases when $i = |C|$, and Line 20 returns the optimal solution for the current optimization window.

Algorithm 5 Dynamic programming (multi-row)

```
1: **Initialize costs for all legal cases** $(j, v, l, b, D, T, s)$
2: $d[0][j][v][l][b][D][T][s] \leftarrow 0,$
3: $d[i][j][v][l][b][D][T][s] \leftarrow +\infty, (0 < i \leq |C|)$
4: **for all** $i = 0$ to $|C| - 1$ **do**
5:     **for all** $d[i][j][v][l][b][D][T][s] \neq \infty$ **do**
6:         **for all** $(j', s') \in \text{getNext}(s)$ **do**
7:             **for all** $(v', l', b')$ **do**
8:                 $i' = i + 1$
9:                 $t \leftarrow d[i][j][v][l][b][D][T][s] + cost(^{j',v',l',b'}_{i,j,v,l,b,D,T})$
10:                $d[i'][j'][v'][l'][b'][D'][T'][s'] \leftarrow$
                       $\min(d[i'][j'][v'][l'][b'][D'][T'][s'], t)$
11:                $Update(D, T)$
12:            **end for**
13:        **end for**
14:    **end for**
15: **end for**
16: $finalCost \leftarrow \infty$
17: **for all** $(j, v, l, b, D, T, s)$ **when** $i = |C|$ **do**
18:     $finalCost \leftarrow \min(d[|C|][j][v][l][b][D][T][s], finalCost)$
19: **end for**
20: **Return** $finalCost$
```

Multi-row optimization is not capable of being aware of HPWL change in y direction and across different optimization windows. Therefore, to prevent HPWL degradation, we add additional displacement costs if a cell is moved out of the original HPWL bounding box, with penalty coefficient $\gamma_{penalty}$, as shown in Equation (2.4).[6] The term $cost_{hpwl}$ is calculated as the distance between the current cell and the original HPWL bounding box, in units of placement sites.

$$cost = cost_{step} + \alpha \cdot cost_{disp} + \gamma_{penalty} \cdot cost_{hpwl} + \alpha \cdot \beta \cdot cost_{var} \tag{2.4}$$

2.1.5 Experiments

We implement our dynamic programming in C++ with OpenAccess 2.2.43 [126] to support LEF/DEF [117], and with OpenMP [122] to enable thread-level parallelism. We perform experiments in an N7 FinFET technology with multi-height triple-V_{th} libraries from a leading technology consortium. The fin height information is not disclosed in our enablement. Therefore, following guidance from [68], we randomly assign fin heights (2, 3, or 4 fins) to each cell with 1:3:6 ratio for 2, 3 and 4 fins, respectively, as our default fin height assignment methodology to match industrial designs at advanced nodes. For example, a double-height cell will have four random fin heights, i.e., for its left and right boundaries on the first row, and its left and right boundaries on the second row. Below, we further discuss the impact of alternative fin height assignment methods.

We generate the bimodal leakage values from the NDE-oblivious standard-cell Liberty file as follows [68]. Since NDE only affects the boundary transistors for each cell, given a leakage value of each standard cell from the Liberty file, we first approximate the boundary transistor leakage value by dividing the state-independent cell leakage by the cell width (in units of contacted-poly pitch), e.g., if a cell (width = 3) has a leakage value of three, then the boundary transistors have a leakage value of one. Then, for each diffusion *step*, 52% of boundary transistor leakage value is added to the cell leakage. In the above example, the cell has a new leakage value of 3.52 (resp. 4.04) when there exists one *step* (resp. two *steps*).

We apply our detailed placement optimization to an Arm Cortex-M0 core (*M0*) and four design blocks (*AES*, *JPEG*, *VGA* and *MPEG*) from OpenCores [121]. Design information is summarized in Table 2.3. We synthesize designs using *Synopsys Design Compiler L-2016.03-SP4* [127], and perform

[6]We pre-calculate all net bounding boxes (one-time effort) and only apply the HPWL penalty if a cell is placed outside of its nets' bounding boxes.

place-and-route using *Cadence Innovus Implementation System v15.2* [114]. We also apply our detailed placement optimization to winning solutions from the ICCAD-2017 multi-deck standard cell legalization contest [17]. All experiments are performed with 8 threads on a 2.6GHz Intel Xeon server.

In the following, we show (i) the scalability and sensitivity, i.e., impact of cell displacement range $x_\triangle$, reordering range r, enabling of cell flipping f, and #rows per window m for the multi-row implementation on runtime and quality of results (QoR in terms of *step* reduction); (ii) impact of the weighting factors, i.e., weighting factor α for cell displacement, weighting factor β for cell flipping, and weighting factor γ for HPWL on QoR; (iii) metaheuristics by combining single-row HPWL-aware and multi-row optimization; (iv) our main results with single-row, double-row and multi-row optimization for five design blocks and three fin height assignment methodologies; (v) performance improvement using *intentional steps*; and (vi) our results with multi-row optimization for ICCAD-2017 benchmarks [17].

Table 2.3: Design information.

Design	**#Inst**	**Clock period**
AES	∼12K	500ps
M0	∼10K	500ps
JPEG	∼54K	500ps
VGA	∼69K	500ps
MPEG	∼14K	500ps

Scalability/Sensitivity Study

In this subsection, we compare the impact of reordering range and displacement range on the single-row (SR), double-row (DR) and multi-row (MR) optimization. By default, we use $m = 2$ in MR optimization (see Figure 2.10 and discussion below). Following results of [36], cell flipping is enabled by default for maximum *step* reduction.

To assess the scalability of our approach, we sweep $(x_\triangle, r)$, i.e., maximum allowed cell displacement $x_\triangle$ (in placement sites) and maximum allowed one-sided reordering r, and study the impact on runtime. In this experiment, we sweep $x_\triangle$ from 0 to 15, and r from 0 to 2. A cell can freely move across 31 placement sites, and can have up to 5 different positions in a placement window, if we set $x_\triangle = 15$

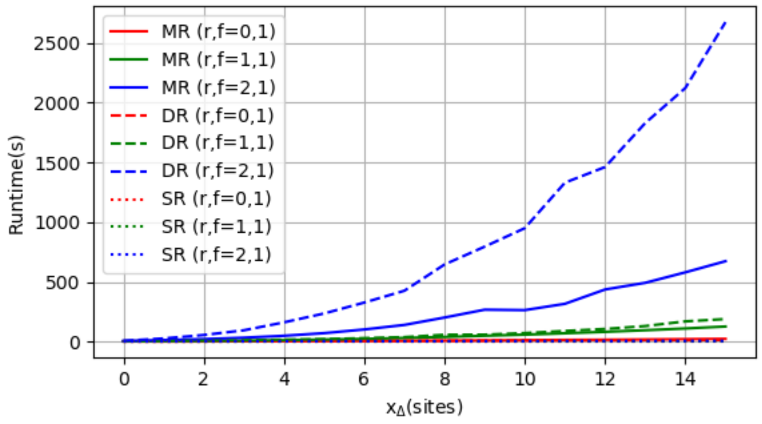


Figure 2.9: Sensitivity of runtime to (x_Δ, r, f) parameters.

and $r = 2$. We set $(\alpha, \beta) = (0, 0)$ as these parameters do not have any impact on the complexity of our formulation. We use design block *AES* for this study.[7]

Our study results are shown in Figure 2.9. We find that the runtime generally grows quadratically with the number of available placement sites per each cell. However, for cell reordering, there is a dramatic increase in runtime as r goes up, e.g., we observe $12\times$ runtime increase going from $r = 1$ to $r = 2$.

Also, compared to DR [36], our new MR implementation with $m = 2$ rows per window is much more efficient in terms of runtime. To investigate the impact of m (#rows in a window) in MR, we compare the sensitivity of #*steps* in Figure 2.10 for $m = 2$ and $m = 3$. Runs with $m = 4$ are not feasible due to much larger memory consumption. We find that $m = 2$ actually gives better #*steps* than $m = 3$ using our N7 library, because all multi-height cells have VSS power rails for their cell boundaries, such that all multi-height cells are aligned per two cell rows. Given the above observation, we use $m = 2$ for MR in all of the following experiments.

To assess the sensitivity to (x_Δ, r), Figures 2.11, 2.12 and 2.13 show #diffusion *steps*, HPWL and RWL respectively, as we sweep (x_Δ, r). Since our algorithm only optimizes #diffusion *steps* when $(\alpha, \beta) = (0, 0)$, here we see HPWL and RWL that correspond to a best-case (minimized) #*steps* normalized to initial design.

[7]To investigate the stability of our sensitivity studies and observations, we also use (i) an alternative *AES* design implementation with slightly different layout, and (ii) design block *M0*. Results for (i) and (ii) are consistent with the results that we report here.

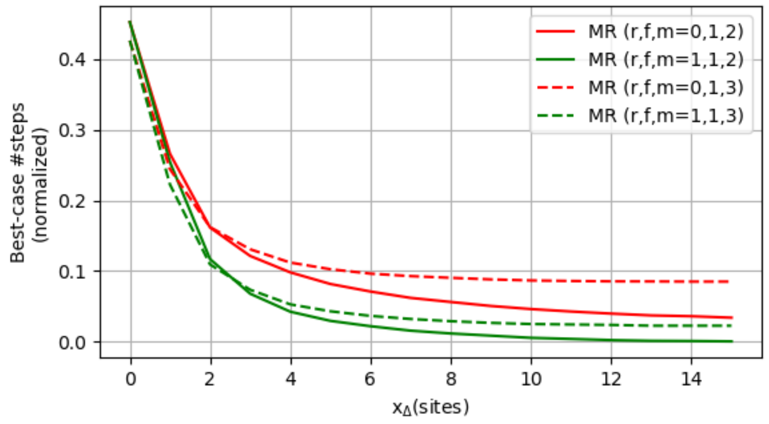


Figure 2.10: Sensitivity of #*steps* to m in MR optimization.

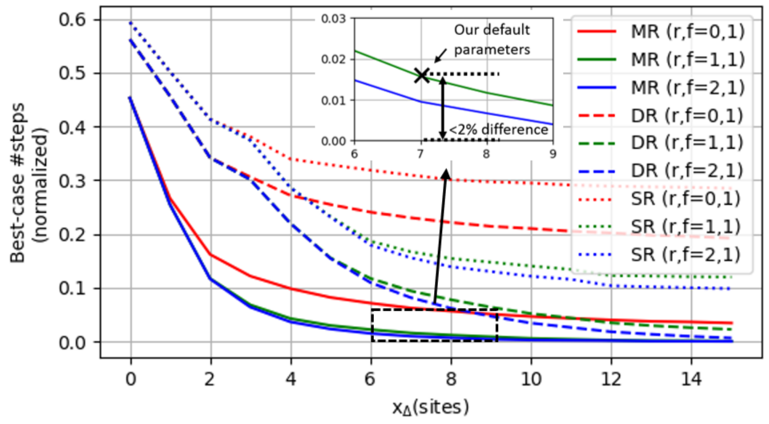


Figure 2.11: Sensitivity of #*steps* to (x_Δ, r, f) parameters.

We see from Figure 2.11 that SR can only reduce #*steps* by up to 80%, while DR and MR are able to reduce #*steps* by up to 99% given larger displacement range. Also, MR is consistently better than DR, especially given a smaller displacement range. Along with the runtime benefit of MR, we believe that the new MR implementation surpasses both the solution quality and the runtime efficiency of DR [36].

Moreover, for $f = 1$, there is only $\sim 0.6\%$ benefit of using $r = 2$ over $r = 1$, at the cost of 12$\times$ the runtime; this suggests that $r \geq 2$ may not offer significant benefit in reducing #*steps*. In Figure 2.12 and Figure 2.13, HPWL and RWL increase linearly as x_Δ goes up. Based on these studies, to balance solution quality and runtime we apply $(x_\Delta, r) = (7, 1)$ in all of the following experiments.

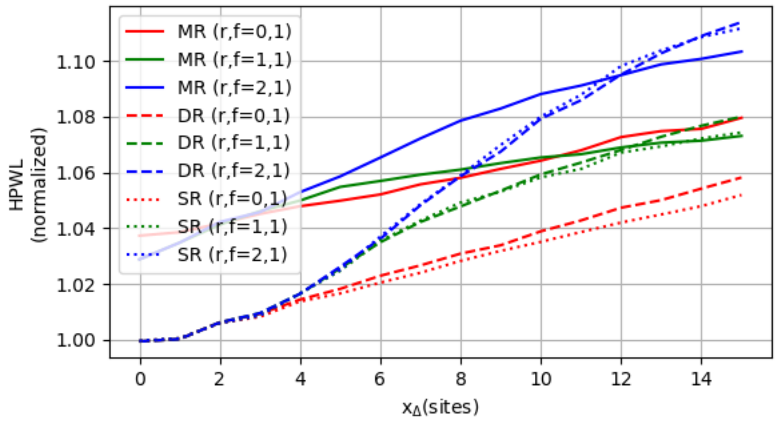


Figure 2.12: Sensitivity of HPWL to (x_Δ, r, f) parameters.

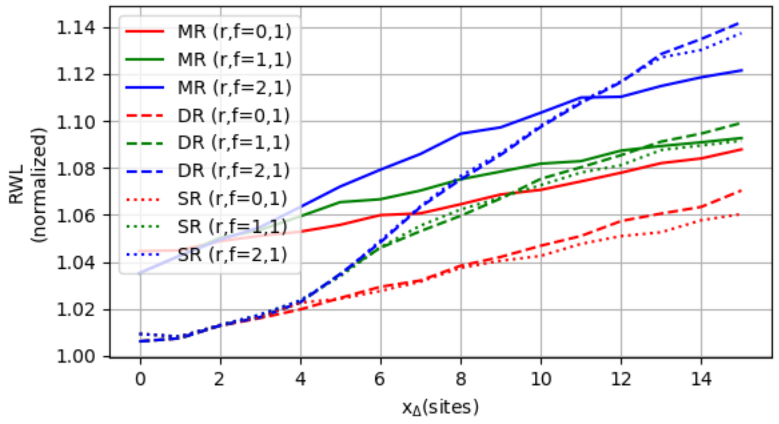
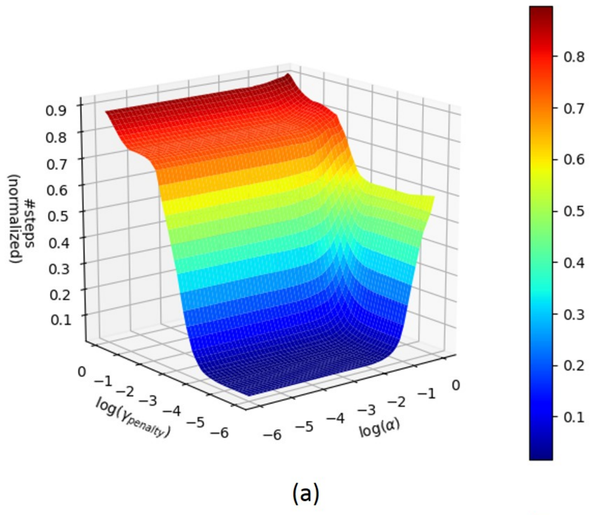


Figure 2.13: Sensitivity of RWL to (x_Δ, r, f) parameters.

Study of Weighting Factors

In the following subsection, our default flow is MR optimization, with two rows per window. We investigate impacts of the weighting factors $(\alpha, \gamma_{penalty})$ for cell displacement and HPWL penalty $(\gamma_{penalty})$ on HPWL and #*steps*. We sweep α and $\gamma_{penalty}$ from 0 to 1. We perform this experiment using design block *AES*. The results are shown in Figure 2.14. We can see that a non-zero displacement weight (α) and a non-zero HPWL penalty $(\gamma_{penalty})$ save HPWL while preserving most of the *step* reduction benefits. Therefore, we apply $\alpha = 0.01$ and $\gamma_{penalty} = 0.00001$ in all following experiments.

For the single-row optimization, we also study the impact of the HPWL weighting factor γ on HPWL and #*steps*. We sweep γ from 0.00001 to 1 with a step size of $10\times$. We perform this experiment using design block *AES*, with results shown in Figure 2.15. The tradeoff between HPWL and #*steps* is clear when γ is in the range of $[0.00001, 0.01]$. We use $\gamma = 0.0001$ for the HPWL-aware single-row optimization.



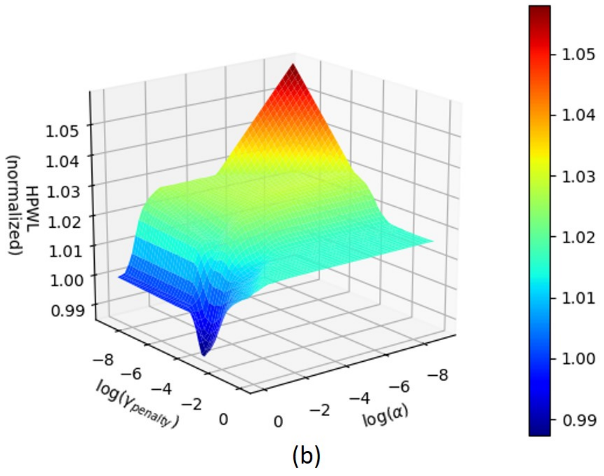


Figure 2.14: Impacts of weighting factors $(\alpha, \gamma_{penalty})$ on the tradeoff between HPWL and #*steps*.

Main Results

We apply our multi-row dynamic programming-based optimization to all our design blocks using the aforementioned parameter settings, i.e., $(x_{\Delta}, r, f) = (7, 1, 1)$ and $(\alpha, \beta) = (0.01, 1)$. Table 2.4 shows

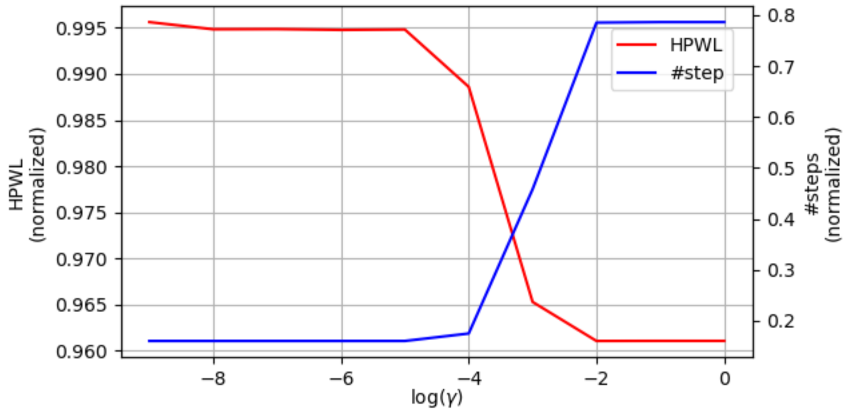


Figure 2.15: Impact of weighting factor γ on the tradeoff between HPWL and #*steps*.

the *step* reduction, runtime and estimated yield improvement for all five design blocks using multi-row optimization. We also report the impact on other metrics, i.e., routed wirelength (RWL), worst negative slack (WNS) and leakage power as reported by the place-and-route tool [114].

Table 2.4: Experimental results for all design blocks using multi-row optimization.

Design	**Type**	**Fin Height Distribution**			**#*Steps***		**RWL (μm)**		**WNS (ns)**		**Leakage (mW)**		**Runtime**	**Est. Yield**
		2 fin%	3 fin%	4 fin%	Init	Final (Δ%)	Init	Final (Δ%)	Init	Final	Init	Final (Δ%)	(s)	**Impr.** %
AES	rand	10.0	30.4	59.6	7973	152 (-98.1%)	31873	32995 (+3.5%)	-0.013	-0.021	16.1	15.8 (-2.1%)	162.1	+0.71
	Vt	48.3	47.8	3.9	6816	143 (-97.9%)	31874	32944 (+3.4%)	-0.013	-0.020	16.6	15.8 (-4.9%)	81.5	+0.66
	drive	47.5	46.6	5.9	7215	236 (-96.7%)	31874	32888 (+3.2%)	-0.013	-0.018	16.1	15.8 (-2.0%)	109.9	+0.69
M0	rand	10.1	30.4	59.4	6588	243 (-96.3%)	27670	28728 (+3.8%)	-0.043	-0.070	18.9	18.6 (-1.9%)	174.4	+0.22
	Vt	49.3	48.6	2.1	5379	152 (-97.2%)	27674	28588 (+3.3%)	-0.043	-0.111	19.5	18.6 (-4.5%)	74.1	+0.52
	drive	46.3	45.6	8.0	6211	398 (-93.6%)	27669	28718 (+3.8%)	-0.043	-0.051	19.1	18.6 (-2.6%)	64.5	+0.58
JPEG	rand	10.0	30.0	60.0	34760	656 (-98.1%)	101000	107699 (+6.6%)	-0.319	-0.278	96.3	94.3 (-2.1%)	776.5	+3.50
	Vt	48.2	48.6	3.3	29452	387 (-98.7%)	100997	106972 (+5.9%)	-0.319	-0.274	98.8	94.4 (-4.4%)	403.2	+2.78
	drive	44.0	44.5	11.5	36173	1291 (-96.4%)	101003	108103 (+7.0%)	-0.323	-0.290	97.2	94.4 (-2.9%)	398.2	+3.30
VGA	rand	10.0	30.1	60.0	50766	6179 (-87.8%)	208155	217492 (+4.5%)	-0.137	-0.080	208.3	205.1 (-1.5%)	713.3	+4.56
	Vt	48.8	49.6	1.6	40743	3685 (-91.0%)	208155	216603 (+4.1%)	-0.137	-0.069	213.4	205.5 (-3.7%)	536.8	+3.48
	drive	42.1	42.8	15.1	57273	10871 (-81.0%)	208155	217664 (+4.6%)	-0.137	-0.129	208.2	205.1 (-1.5%)	491.1	+4.24
MPEG	rand	9.9	30.5	59.6	9994	1367 (-86.3%)	38896	40594 (+4.4%)	-0.005	-0.018	33.2	33.1 (-0.2%)	137.3	+0.87
	Vt	49.6	49.4	1.0	7824	753 (-90.4%)	38882	40383 (+3.9%)	-0.011	-0.026	33.2	33.1 (-0.3%)	68.6	+0.70
	drive	43.1	43.1	13.8	10931	2145 (-80.4%)	38901	40649 (+4.5%)	-0.005	-0.030	33.2	33.1 (-0.3%)	99.5	+0.86

We also investigate the impact of fin height assignment methodologies. We apply three methodologies – (i) *rand* randomly assigns fin heights according to probability ratio 1:3:6 for 2, 3, and 4 fins, respectively (see Section 2.1.5 above); (ii) *Vt* assigns fin heights according to their V_{th} property, with HVT (resp. NVT and LVT) cells having probability ratio 1:1:0 (resp. 1:1:1 and 0:1:1) for 2, 3, and 4 fins; (iii) *drive* assigns fin height according to their drive strength, with X0 (resp. X1 and others) cells having probability ratio 1:1:0 (resp. 1:1:1 and 0:1:1) for 2, 3, and 4 fins. The three methodologies generate different fin height distributions, and thus help confirm the robustness of our optimization in broader scenarios. The results are shown in Table 2.4. For all designs with the default (*rand*) random fin height distribution, we achieve up to 98.1% reduction in #*steps* at the cost of around 3.5% RWL increase. The results also show that our optimization has negligible impact on WNS and that we can slightly improve the leakage. In addition, we perform a preliminary yield estimation assuming 2ppm failure rate for each *step*, and 1ppm failure rate after we remove the *step* (recall Footnote 2). Based on this assumption, we can see a yield improvement of up to 4.56% for a design block of 69K instances. We note that the yield improvement is expected to grow markedly with the die size. A larger design with many millions of instances may see more benefits.

For V_{th} and drive distribution, the results show similar *step* reduction percentage, demonstrating the robustness of our optimization. Figure 2.16 shows the layouts of placements before and after MR optimization.

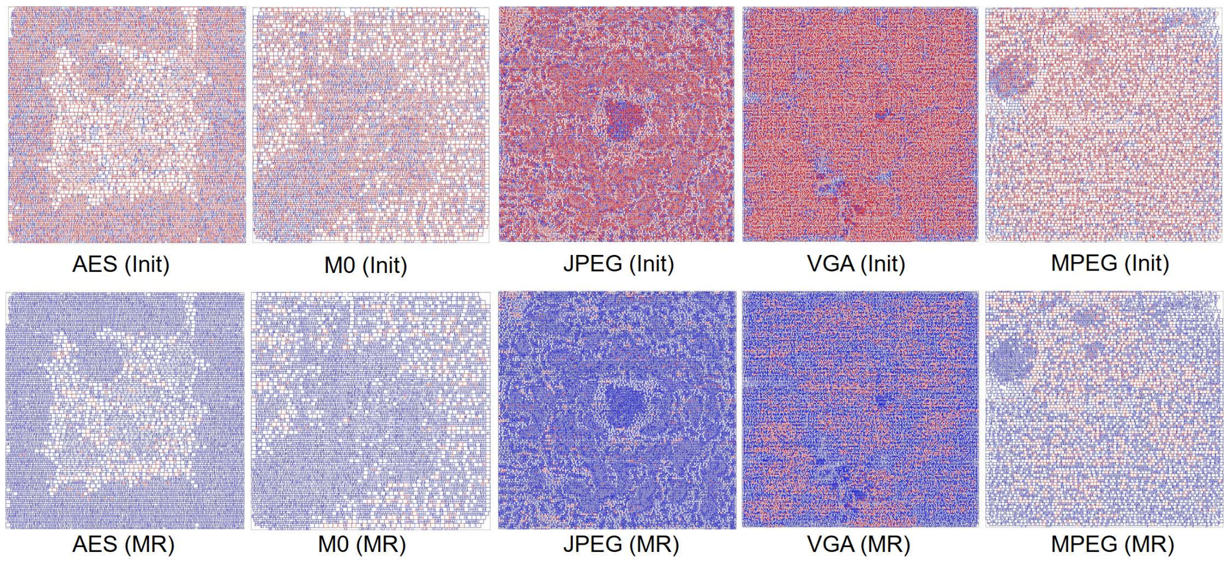


Figure 2.16: Layouts of placements before (Init) and after (MR) our MR optimization. Red color indicates cell instances with diffusion *steps* and blue color indicates cell instances without diffusion *steps*.

We also investigate the improvement achieved by our multi-row optimization over single-row, double-row optimization and previous works. We compare multi-row (MR) optimization to (i) single-row (SR) optimization (also to match [26][57]), (ii) ordered double-row (ODR) optimization (to match [56]), and (iii) double-row (DR) optimization. For (i), we use the proposed methodology in Section 2.1.2 and fix the locations of all multi-height cells. We note that our SR implementation is equivalent to [26][57], supporting neighboring cell swapping and cell flipping with the adaptation of NDE. In SR, we use the same displacement range and reordering range as in DR, while using the default HPWL weighting factor $\gamma = 0.0001$ (HPWL weighting factor is not considered in the work of [36]). For (ii), we simply run our DR optimization with zero reordering range to achieve an ODR equivalent to [56]. For (iii), we use the proposed methodology in Section 2.1.3. The comparisons of #*steps*, routed wirelength (RWL) and runtime are shown in Tables 2.5, 2.6 and 2.7, respectively. For design blocks with fewer double-height cells, SR performance is competitive with that of ODR. However, for design blocks with more double-height cells, ODR is significantly better (up to 21% more *step* reduction) than SR due to movable double-height cells. The results show that DR effectively reduces the diffusion *steps* by around half compared to SR, and by around 40% compared to ODR. On average, DR has 11.6% more *step* reduction than ODR, and 17.7% more than SR, with respect to the initial number of diffusion *steps*. This suggests the importance of supporting movable and reorderable double-height cells, as there will be substantial benefits.

Table 2.5: Comparison of diffusion *steps* with SR (to match [26][57]), ODR (to match [56]) DR, MR and metaheuristics (Meta). DH% = % of double-height cells.

Design	DH%	Init	**SR (to match [26][57])**	**ODR (to match [56])**	**DR**	**MR**	**Meta**
AES	4.3%	7973	1395 (-82.5%)	1869 (-76.6%)	750 (-90.6%)	152 (-98.1%)	131 (-98.4%)
M0	8.4%	6588	1672 (-74.6%)	1742 (-73.6%)	842 (-87.2%)	243 (-96.3%)	179 (-97.3%)
JPEG	8.3%	34760	9731 (-72.0%)	8341 (-76.0%)	4555 (-86.9%)	656 (-98.1%)	473 (-98.6%)
VGA	24.8%	50766	27170 (-46.5%)	16405 (-67.7%)	11816 (-76.7%)	6179 (-87.8%)	5652 (-88.9%)
MPEG	23.0%	9994	5101 (-49.0%)	3444 (-65.5%)	2402 (-76.0%)	1367 (-86.3%)	1215 (-87.8%)
Avg.	–	-0.00%	-64.9%	-71.9%	-83.5%	-93.3%	-94.2%

Table 2.6: Comparison of routed wirelength (RWL) with SR, ODR, DR, MR and metaheuristics (Meta).

Design	Init	**SR**	**ODR**	**DR**	**MR**	**Meta**
AES	31873	32517 (+2.02%)	32637 (+2.40%)	32898 (+3.22%)	32995 (+3.52%)	33065 (+3.74%)
M0	27670	28201 (+1.92%)	28271 (+2.17%)	28470 (+2.89%)	28728 (+3.82%)	28805 (+4.10%)
JPEG	101000	104562 (+3.53%)	104657 (+3.62%)	105550 (+4.50%)	107699 (+6.63%)	108173 (+7.10%)
VGA	208155	212186 (+1.94%)	212905 (+2.28%)	214169 (+2.89%)	217492 (+4.49%)	216856 (+4.18%)
MPEG	38896	39640 (+1.91%)	39799 (+2.32%)	39950 (+2.71%)	40594 (+4.37%)	40512 (+4.15%)
Avg.	+0.00%	+2.26%	+2.56%	+3.24%	+4.57%	+4.66%

Table 2.7: Comparison of runtime (seconds) with SR, ODR, DR, MR and metaheuristics (Meta).

Design	SR	ODR	DR	MR	Meta
AES	32	8	59	162	348
M0	22	8	51	174	214
JPEG	325	50	344	776	2153
VGA	493	51	386	713	1658
MPEG	30	11	86	137	234

Metaheuristics

We have also explored several metaheuristics to assess (i) the *step* reduction achievable by invoking multiple optimization iterations, as well as (ii) potential improved tradeoffs between *step* reduction and degradation from initial placement (in terms of HPWL). First, we investigate the maximum *step* reduction versus the number of iterations. To explore the maximum benefits of *step* reduction, we invoke the multi-row optimization several times. Since the multi-row optimization is for every two rows, e.g., row 1 and 2 in a window, row 3 and 4 in the next window, etc., we can shift the window by one row and run again if we can further improve the solution quality. In our experiments, we alternatively align/unalign the optimization window with double-height cells, with aligned window in the first iteration to encourage the movement of double-height cells. We show the normalized number of *diffusion steps* and HPWL versus the number of optimization iterations (up to 8) in Figure 2.17. Compared to one iteration, the second iteration removes 45 out of 152 remaining steps after the first iteration, while the remaining six iterations only reduce 13 more *steps*, at the cost of increased HPWL.

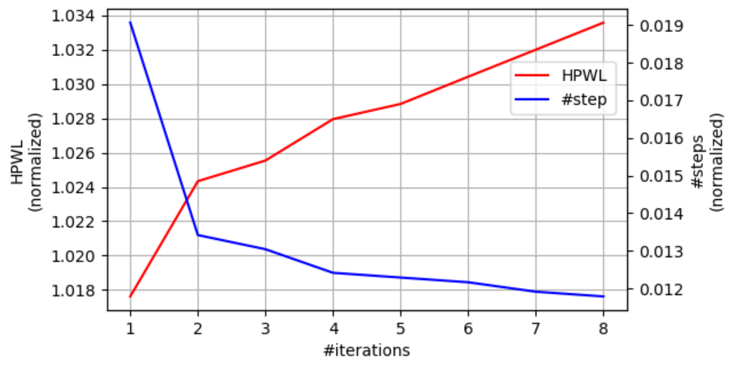


Figure 2.17: #*steps* (normalized) and HPWL (normalized) vs. #iterations in metaheuristic optimization.

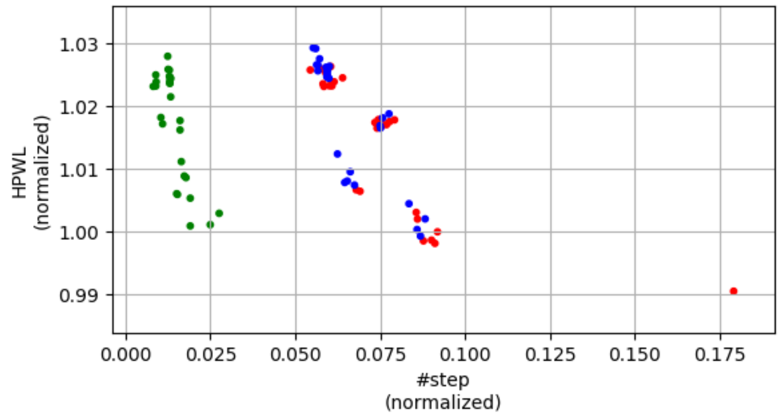


Figure 2.18: #steps vs. HPWL in metaheuristic optimization. Red (resp. green and blue) dots represent metaheuristic iterations that start with configuration A (resp. configuration B and configuration C).

Given the above observation, we seek to obtain a better tradeoff between *step* reduction and HPWL. Since our multi-row optimization is not HPWL-aware, we propose to invoke both single-row and multi-row optimization with a total "budget" of four iterations, to find the best four-iteration sequence. We explore all possible optimization sequences comprised of the following three configurations – (A) single-row HPWL-aware; (B) multi-row aligned with double-height cells; and (C) multi-row unaligned with double-height cells. We report the optimized number of *steps*, along with HPWL, in Figure 2.18. We can see that the configuration for the first iteration largely determines the optimized number of *steps*. The first iteration should be (B) to obtain better *step* reduction. Also, the optimization should finish with (A) for better HPWL. We report the metaheuristic results in Tables 2.5, 2.6 and 2.7.

Performance Improvement Using Intentional Steps

Similar in spirit to [47], we explore the possibility of improving design performance with *intentional steps* – i.e., using filler cells that create an *intentional step* to the neighboring timing-critical functional cell so as to improve the timing of that functional cell.[8] In the cost function, we use a third

[8]An *intentional inter-cell step* may increase/decrease the drive strength of the function cell. E.g., a *step* adjacent to a PFET may decrease the drive strength while a *step* adjacent to an NFET may increase the drive strength. Here, instead of using a filler cell to match diffusion heights for both the NFET and the PFET of the function cell (to reduce #*steps*), we create a filler-induced *intentional step* by matching the diffusion height for only the PFET, thus increasing the drive strength for the NFET. We note that exact timing and power impacts and tradeoffs will vary with STI processes.

weighting factor δ to represent the benefit of an *intentional step* to a timing-critical cell. We sweep δ from 0 to -2 with a step size of -0.1. We select 5% of all cells as timing-critical cells and perform optimization using all design blocks. The results are shown in Figure 2.19. We use $orig.opt$ to represent the results with $\delta = 0$, and $time.opt$ to represent the results with $\delta = -0.3$. Compared to $\delta = 0$, we achieve up to 5$\times$ increase in #*filler-induced steps* incident to timing-critical cells when $\delta = -0.3$, at the cost of slightly increased #*non-filler-induced steps* to non-timing-critical cells. This translates to up to 2.13 *steps* per timing-critical cell after $time.opt$, compared to 0.42 *steps* after $orig.opt$. Overall, we can still decrease total *steps* by more than 70%, showing the effectiveness of our algorithm. We note that as we add more *intentional steps* to timing-critical cells, we leave a smaller solution space for non-timing-critical cells. Thus, $time.opt$ generates more *steps* to non-timing-critical cells. We furthermore observe that as δ decreases, the #*intentional steps* that we can achieve approaches a limit, as shown in Figure 2.20. This may help set expectations for benefits that might be derived from a more comprehensive, timing-aware flow (which we leave for future work).

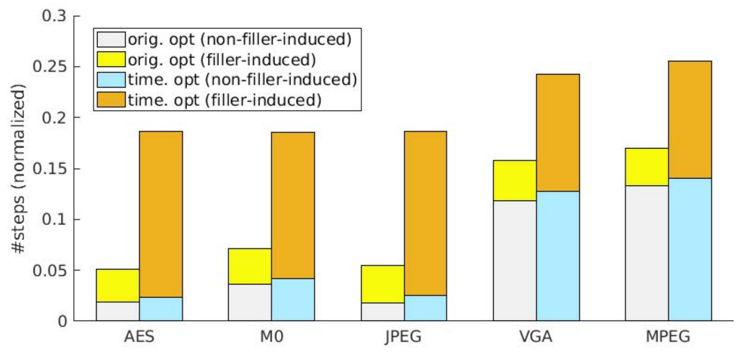


Figure 2.19: Comparison of #filler-induced *steps* and total #*steps* for all design blocks before ($orig.opt$, $\delta = 0$) and after ($time.opt$, $\delta = -0.3$) using *intentional steps*.

ICCAD-2017 Benchmark Results

We apply our multi-row dynamic programming-based optimization to winning solutions from the ICCAD-2017 contest [17] only considering row and site alignments, but not considering constraints, including maximum cell movement, cell edge spacing, pin access, pin shorts and fence regions from the contest. The input legalized placements for all benchmark testcases are from the first-place team's

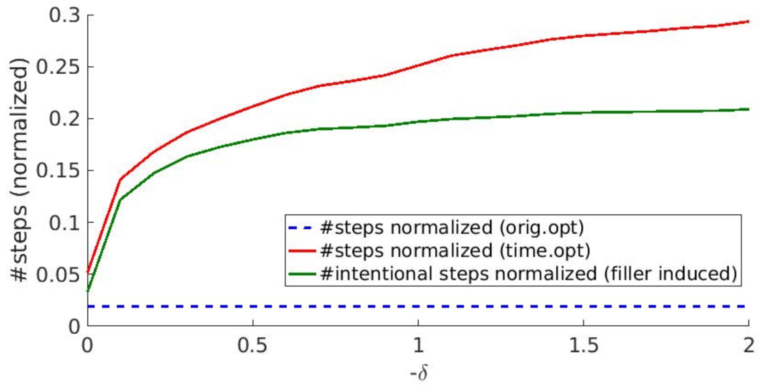


Figure 2.20: Sensitivity of filler-induced *steps* to δ. Testcase: *AES*.

solutions in ICCAD-2017 contest, except *pci_bridge32_a_md1* and *pci_bridge32_a_md2*, for which we use the second-place team's solutions (because the first-place team's solutions for these two testcases have cells placed outside of the die boundary). We keep the same P/G alignment as in the input placement. We apply *rand* fin height assignment methodology with the above-mentioned 1:3:6 ratio for 2, 3 and 4 fins, respectively. The results are shown in Table 2.8. For all ICCAD-2017 benchmark testcases, we achieve up to 96.8% reduction in #*steps*.

2.1.6 Conclusion

In this work, we present an *optimal* dynamic programming-based single-/double-row detailed placement methodology to minimize diffusion *steps* in sub-10nm VLSI, for improved yield and mitigation of NDE. Our work achieves several improvements as compared to previous works: (i) optimal dynamic programming with support of a richer set of cell movements, i.e., flipping, relocating and enhanced reordering; (ii) optimal double-row dynamic programming *with support of movable and reorderable double-height cells*; and (iii) a novel performance improvement technique using *intentional steps*. The proposed techniques achieve up to 98% reduction of inter-cell diffusion *steps*, with scalable runtime and high die utilization in an N7 node enablement.

Table 2.8: Design information and experiment results for ICCAD-2017 benchmark [17]. Distribution of single-height, double-height, triple-height and quadruple-height cells are shown in columns 1×H, 2×H, 3×H and 4×H, respectively.

Design	**#Inst**	**Cell types %**				***#Steps***		**Runtime**
		1×H	2×H	3×H	4×H	Init	Final (Δ%)	(s)
des_perf_b_md1	∼11K	94.80	5.20	0.00	0.00	57806	3781 (-93.46%)	361.3
des_perf_b_md2	∼11K	90.47	6.02	2.01	1.50	70733	7494 (-89.41%)	232.8
edit_dist_1_md1	∼13K	90.31	6.12	2.04	1.53	74351	6019 (-91.90%)	420.9
edit_dist_a_md2	∼13K	90.31	6.12	2.04	1.53	76657	8074 (-89.47%)	417.8
fft_2_md2	∼ 3K	89.62	6.56	2.18	1.64	22040	3789 (-82.81%)	53.2
fft_a_md2	∼ 3K	89.57	6.59	2.19	1.65	10960	606 (-94.47%)	136.4
fft_a_md3	∼ 3K	93.42	2.19	2.19	2.19	11631	372 (-96.80%)	78.1
pci_bridge32_a_md1	∼ 3K	90.39	6.07	2.02	1.52	17284	1429 (-91.73%)	83.8
des_perf_1	∼11K	100.00	0.00	0.00	0.00	73202	3516 (-95.20%)	488.7
des_perf_a_md1	∼11K	95.66	4.34	0.00	0.00	64624	3060 (-95.26%)	307.3
des_perf_a_md2	∼11K	96.99	1.00	1.00	1.00	64346	4793 (-92.55%)	315.9
edit_dist_a_md3	∼13K	93.88	2.04	2.04	2.04	78560	11100 (-85.87%)	258.9
pci_bridge32_a_md2	∼ 3K	85.51	7.08	4.05	3.37	21435	6235 (-70.91%)	71.2
pci_bridge32_b_md1	∼ 3K	90.39	6.07	2.02	1.52	14988	1070 (-92.86%)	68.1
pci_bridge32_b_md2	∼ 3K	96.97	1.01	1.01	1.01	13812	488 (-96.47%)	135.0
pci_bridge32_b_md3	∼ 3K	94.94	1.01	2.02	2.02	14929	1193 (-92.01%)	84.2

2.2 In-Route Detailed Placement Refinement for Improved Detailed Routing Convergence

In advanced technology nodes, detailed routing is a critical challenge. With smaller feature sizes, more and more complex design rules are introduced with each new technology enablement. Such complex design rules make detailed routing ever-more challenging. This is especially true with respect to *pin access*, where the detailed router aims to achieve DRC-clean connection to complicated pin shapes with comprehension of not only design rules, but intra-cell and inter-cell pin shape interactions as well.

Pin accessibility, which measures ease or difficulty of pin access, is an important measurement of routability. Pin accessibility assessment and modeling have been widely studied in recent works. Xu et al. [104] propose a dynamic hit point scoring strategy to dynamically assess pin accessibility based on the number of pin access points described in [102]. Seo et al. [85] propose a metric (i.e., Inaccessibility of Cell) to describe pin accessibility considering the number of access points and interference among access points based on a pre-defined threshold. Ding et al. [21] define a pin access region for each standard cell pin and propose a pin access penalty function based on distance and visibility between two pin access regions. Yu et al. [108] propose a deep learning-based pin pattern recognition methodology for pin accessibility prediction and optimization. In this work, we adopt the concept of *pin access pattern*, i.e., combination of pin access points, as in [50].

Many recent works focus on pin access-aware detailed placement optimizations. [21] proposes a two-phase pin accessibility-driven detailed placement refinement. The first phase performs cell flipping and swapping adjacent cells, and the second stage performs cell movement. Taghavi et al. [90] propose a local congestion metric considering local pin accessibility and apply a suite of detailed placement techniques, MILOR, to mitigate local congestion. Chow et al. [14] propose a two-step global-local move detailed placement algorithm to minimize wirelength considering cell density. In summary, these works attempt to optimize pin accessibility during placement stage with modeled pin accessibility information – in similar fashion as in a conventional physical design tool flow.[9]

In a conventional physical design tool flow, the placement tool uses models including cell density, pin density, etc. to estimate the pin accessibility. However, such estimation can be inaccurate, if not misleading. Figure 2.21(a) and Figure 2.21(b) have the same placement solution attributes (i.e., same cell and pin density) while the relative locations between pin shapes and track locations differ. The pin accessibility in Figure 2.21(a) is better than the pin accessibility in Figure 2.21(b) because of more on-grid access points, especially for pin A near the left cell boundary. A placement solution with poor pin accessibility, or poor pin accessibility correlation with routing, can cause a considerable amount of initial detailed routing design rule violations (DRCs).[10] Although detailed routers have the capability to iteratively fix DRCs, a large number of initial detailed routing DRCs can lead to (i) many iterations and long runtimes needed for DRC convergence, if the detailed router can converge on DRC at all; and (ii) longer routed wirelength due to detours needed to fix DRCs.

In this work, we propose an *in-route*, pin access-driven local placement refinement using a hybrid wirelength model with timing awareness (i.e., an objective function with weighted sum of timing and

[9]We adopt the widely-studied ordered-row placement in this work for scalability consideration.

[10]According to [66], the detailed routing can be divided into two steps. The initial detailed routing step handles the major routing rules; then, the detailed routing refinement step fixes the remaining complicated DRCs.

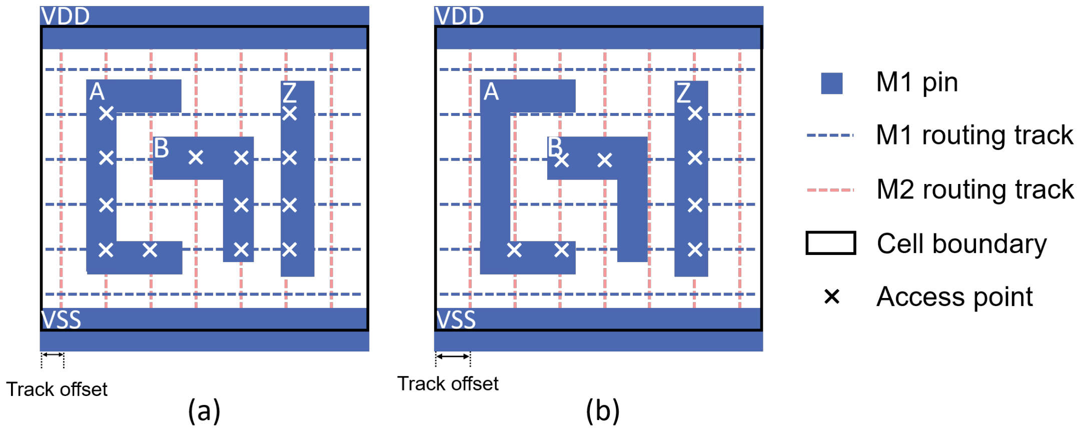


Figure 2.21: Pin access points with different routing track-placement site offsets.

wirelength components). With application of our optimization, a leading commercial P&R tool's detailed routing runtime can be significantly reduced as we attain improved initial detailed routing DRCs and final routed wirelength without timing degradation. To our knowledge, ours is the first detailed placement framework that comprehends *exact pin access*, precisely as the detailed router understands this. Our main contributions are summarized as follows.

- We propose a dynamic programming based (DP) approach to minimize a cost function that is aware of pin accessibility, wirelength and timing while also considering EEQ cell swapping for advanced technology nodes.
- We propose a pin accessibility model that comprehends interactions between neighboring standard cell instances.
- With integration of our optimization, the commercial P&R tool's detailed routing runtime can be reduced by up to 31.82% (avg. 15.06%). Moreover, our optimization results in up to 10.13% initial detailed routing DRC reduction, across 19 industry designs using various technology nodes.

The remainder of this work is organized as follows. Section 2.2.1 describes our problem formulation. Section 2.2.2 details our pin access-driven detailed placement optimization flow. Section 2.2.3 presents our experimental setup and results. Section 2.2.4 gives conclusions.

2.2.1 Pin Access-Driven Placement Optimization

We now present our problem statement and formulation for pin access-driven detailed placement optimization.

Pin Access-Driven Optimization Problem. *Given an initial legalized placement with pin access, perturb the placement to minimize overall cost.*

Inputs: A legal initial placement with pin access and a cost function considering pin accessibility, wirelength and timing.

Output: Optimized detailed placement with minimized overall cost (including pin accessibility).

Constraints: Maximum displacement range and placement legality.

We make the following observation and assumption with respect to this problem statement.

Observation. *Each cell row can be separated from each other from its neighboring cell rows in terms of pin accessibility.*

In a technology library that contains only single-height standard cells, the observation is obvious since the pin access points are well separated by VDD/VSS power rails inside the standard cell. For technology libraries that contain multi-height standard cells, the observation follows from the fact that shapes of standard cells are usually large. Hence, the pin accessibility inside the multi-height cell is usually decent, and instances of multi-height cells are unlikely to have pin access conflicts with their neighboring cell instances.

Assumption. *The locations of multi-height cell and clock-related instances are fixed.*

In advanced technology nodes, there are cells that span more than one standard cell row. Such multi-height cells usually are complex functional cells (e.g., flip-flops). Displacement of flip-flops in a well-optimized placement solution can introduce dramatic degradation in timing, especially because flip-flops are on critical timing paths. Similarly, in a well-optimized design, clock buffers are placed to satisfy various clock-related constraints (e.g., clock skew).

Notations

Table 2.9 shows the notations we use in our formulation. For each cell instance c_i, **cell instance index** *i* indicates it is the i^{th} left-to-right cell instance in the cell row from the initial placement. We use C to denote the set of cell instances in a row of the initial placement.

Table 2.9: Notations.

Notation	**Meaning**
C	set of cells in a row of the initial placement
c_i	i^{th} cell in the left-to-right ordered initial placement, where i is *the cell index*
$[-x_\Delta, x_\Delta]$	displacement range
x_i	absolute x-coordinate of c_i in the initial placement, in units of placement sites
l	displacement of a cell from its location in the initial placement, in units of placement sites
$d[i][l]$	dynamic programming table entry indicating that the i^{th} cell is placed with displacement l from its initial location

In order to honor the initial placement solution, we define a **displacement range** with x_Δ, in units of placement sites, to limit the amount of placement perturbation. We use x_i to denote the absolute x-coordinate, in units of placement sites, of c_i in the initial placement. Therefore, the absolute x-coordinate of c_i in the final placement solution must be within $[x_i - x_\Delta, x_i + x_\Delta]$. We use the variable $l \in [-x_\Delta, x_\Delta]$ to describe the **displacement** of a cell instance from its initial placement.

We use a node array $d[i][l]$ as the underlying structure for a dynamic programming recurrence. Each node represents a unique placement location of cell instance c_i with displacement l. The information contained in each node is summarized in Table 2.10. $nodeCost$ indicates the cost of placing the cell instance at the location implied by the node displacement index l. $pathCost$ is the lowest accumulated path cost from the first cell instance of the row to the current node. $pattern$ contains the information of the pre-selected best pin access pattern for the current node. Note that although pin access pattern, in theory, can be one dimension of the dynamic programming, we pre-select the pin access pattern for a cell instance at each possible location defined by displacement range in order to reduce runtime. We detail the pin access selection procedure in Section 2.2.2. $prevNode$ points to the previous cell instance node with the lowest path cost.

Table 2.10: Dynamic programming node notations.

Notation	Meaning
$nodeCost$	cost of placing the cell instance at the location indicated by the node
$pathCost$	lowest accumulated path cost to the current node
$pattern$	pre-selected pin access pattern for the current node
$prevNode$	pointer to the previous node with lowest path cost

Dynamic Programming (DP) Formulation

In our DP formulation, cell instances of a given cell row are placed sequentially from left to right in the same order as in the initial placement. Algorithm 6 describes our DP-based, pin access-driven detailed placement optimization procedure. Lines 2–3 populate DP nodes with pre-selected pin access patterns. Procedure $getPattern$, as described in Algorithm 7, pre-selects the pin access pattern for cell instance c_i with displacement l. Lines 4–5 initialize the pathCost of each node. Lines 6-16 describe the core part of the DP algorithm. Starting with the first cell instance, the algorithm sequentially places cell instances based on the recursive relation as described in Lines 8–14. Lines 9–11 prune placement solutions with overlapped cells. Procedure $cost$ computes the placement cost between partial placement solutions described by $d[i][l]$ and $d[i+1][l']$. Lines 17–21 find and return the solution with the lowest overall cost. The overall runtime of Algorithm 6 scales as $\mathcal{O}(|C|x_{\Delta}^2)$ but is negligible in practice (see Table 2.13 below in Section 2.2.3).

The procedure $cost(^{i+1,l'}_{i,l})$ calculates the cost of placing c_{i+1} with displacement l' as a weighted sum of (i) wirelength cost $cost_{WL}$ and (ii) pin access cost $cost_{PA}$ as shown in Equation 2.5. We use a wirelength weighting factor $\alpha \in [0,1]$ to balance the tradeoff between wirelength and pin accessibility.

$$cost(^{i+1,l'}_{i,l}) = \alpha \cdot cost_{WL}(i+1,l') + (1-\alpha) \cdot cost_{PA}(^{i+1,l'}_{i,l}) \tag{2.5}$$

Algorithm 6 Dynamic programming

```
1: **Inputs**: dynamic programming array $d$, track patterns *tps*, access patterns array *APs*
2: **Output**: minimum cost of optimized placement solution
3: **Initialize pin access pattern for all nodes** ($l \in [-x_\Delta, x_\Delta]$)
4:   $d[i][l].pattern \leftarrow getPattern(APs[\mathrm{i}], tps, d[i][l])(0 < i \leq |C-1|)$
5: **Initialize cost for all nodes**
6:   $d[0][l].pathCost \leftarrow 0$, $d[i][l].pathCost \leftarrow +\infty, (0 < i < |C|)$
7: for all $i = 0$ to $|C| - 1$ do
8:   for all $d[i][l].pathCost \neq +\infty$ do
9:     for all $l' \in [-x_\Delta, x_\Delta]$ do
10:      if isOverlap($d[i][l]$, $d[i+1][l']$) then
11:        continue
12:      end if
13:      $t \leftarrow d[i][l].pathCost + cost(^{i+1,l'}_{i,l})$
14:      $d[i+1][l'].pathCost \leftarrow \min(d[i+1][l'].pathCost, t)$
15:    end for
16:  end for
17: end for
18: $finalCost \leftarrow \infty$
19: for all $d[|C|-1|][l]$ do
20:   $finalCost \leftarrow \min(d[|C-1|][l].pathCost, finalCost)$
21: end for
22: Return $finalCost$
```

2.2.2 Flow

We now describe the overall flow of our pin access-driven placement optimization, along with key elements including pin access pattern selection and cost function calculations.

Pin Access Pattern Selection

In our work, DRC-clean pin access patterns comprehending all pin shapes are an input from the detailed router's integrated pin access analysis engine. In order to enable efficient and accurate wirelength calculation, we perform pin access selection prior to our *in-route* detailed placement optimization. In advanced, especially sub-$7nm$, technology nodes, electrically-equivalent (EEQ) cell swapping is necessary for cell instance movement in order to ensure alignment between colored routing tracks and pin shapes. Therefore, for each dynamic programming node with displacement $l \neq 0$, we pre-calculate the best EEQ cell to use for the given placement site prior to pin access pattern selection.

For all dynamic programming nodes representing cell instances at original location (i.e., $l = 0$), we preserve the pre-defined access patterns which are already optimized for wirelength and pin accessibility. For all dynamic programming nodes with displacement $l \neq 0$, we perform the pin access pattern selection as described in Algorithm 7.

The main goal of Algorithm 7 is to preemptively avoid potential design rule violations between access patterns from neighboring cell instances.[11] Line 3 sorts the access patterns based on their on-gridness with respect to the routing tracks. The one with the most on-gridness is less likely to conflict with its neighboring cells. Line 4 initializes the best access pattern to the access pattern with the most on-grid access points. Lines 5–15 iterate through all access patterns. Lines 6–7 check whether the current dynamic programming node (i.e., $d[i][l]$), which represents the placement of the current instance, overlaps with the original placement solution of the next instance (i.e., $d[i+1][0]$). If the two nodes overlap, we return the access pattern with the most on-grid access points. Lines 9–10 check whether the current access pattern conflicts with the access pattern of the next instance at its original location. Lines 11–13 return the best access pattern that does not conflict with the access pattern of the next instance.

Algorithm 7 Pin access pattern selection *getPattern*

```
1: **Inputs**: access patterns aps, track patterns tps, dynamic programming node $d[i][l]$
2: **Output**: best access pattern $ap_{best}$ for $l \neq 0$
3: $aps$ = sort($aps$, $tps$)
4: $ap_{best}$ = $aps[0]$
5: for all access pattern $ap \in aps$ do
6:   if isOverlap($d[i][l]$, $d[i+1][0]$) then
7:     break
8:   else
9:     if isConflict($ap$, $d[i+1][0].pattern$) then
10:      continue
11:    else
12:      $ap_{best} = ap$
13:      break
14:    end if
15:  end if
16: end for
17: return $ap_{best}$
```

[11]We only check the right neighbor of an instance as DP propagates from left to right, since conflicts between neighboring access patterns are symmetric.

Cost Functions

As mentioned in Section 2.2.1, we consider two cost components in our formulation (i.e., $cost_{WL}$ and $cost_{PA}$).

Wirelength cost. The wirelength cost $cost_{WL}$ for a node $d[i][l]$ depends only on the placement of the cell instance c_i itself. The wirelength cost is calculated based on Equation 2.6.

$$cost_{WL}(i,l) = \sum_{pin \in c_i} (m(pin,l) - m(pin,0)) \tag{2.6}$$

In order to prevent timing degradation, we use a hybrid wirelength metric, as shown in Equation 2.7, with timing awareness where (i) $dist_{CG}$ denotes distance to the center of gravity of pins of the net which the given pin belongs to, and (ii) $dist_{driver}$ denotes distance to the driver pin of the net which the given pin belongs to.

$$m(pin,l) = min(dist_{CG}(pin,l), dist_{driver}(pin,l)) \tag{2.7}$$

Pin access cost. The pin access cost $cost_{PA}(^{i+1,l'}_{i,l})$ is calculated based on the boundary pin cost $cost_{BP}$ between cells c_i and c_{i+1}, having corresponding displacements l and l', as shown in Equation 2.8. For c_i and c_{i+1}, pin access conflicts could occur between the pins near the cell boundaries.

$$cost_{PA}(^{i+1,l'}_{i,l}) = cost_{BP}(^{i+1,l'}_{i,l}) - cost_{BP}(^{i+1,0}_{i,0}) \tag{2.8}$$

We calculate the boundary pin cost using Equation 2.9 where $BP_R(i)$ (resp. $BP_L(i+1)$) indicates the right boundary pins of c_i (resp. left boundary pins of c_{i+1}).

$$cost_{BP}(^{i+1,l'}_{i,l}) = \sum_{\substack{pin_1 \in BP_R(i) \\ pin_2 \in BP_L(i+1)}} cost_{pin(pin_1,pin_2)} \tag{2.9}$$

We use a threshold distance $dist_{thres}$ to query the boundary pins and check potential conflicts between boundary pins. Figure 2.22 illustrates the boundary pins defined by $dist_{thres}$.

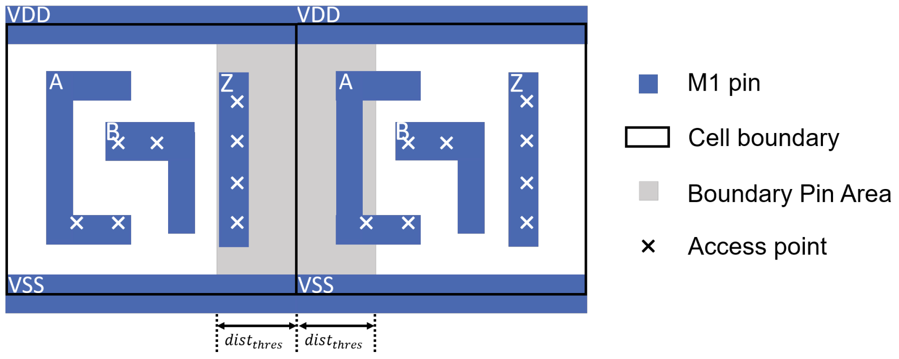


Figure 2.22: Illustration of boundary pins with threshold distance $dist_{thres}$.

We calculate the pairwise pin access cost for the boundary pin pairs from c_i and c_{i+1} using Equation 2.10, where *PAP(pin)* represents the set of pin access points of a boundary pin and $dist$ calculates the distance between two boundary pins.

$$cost_{pin}(pin_1, pin_2) = \frac{|PAP(pin1)| \cdot |PAP(pin2)|}{dist(pin_1, pin_2)} \tag{2.10}$$

Overall Flow

Figure 2.23 illustrates the difference between the conventional routing flow and the routing flow with our proposed placement optimization. The red box implements what we refer to as *in-route* detailed placement refinement. We take an initial placement solution and pin access patterns as inputs. We then select the access pattern for each DP node. Next, based on the (already well-optimized) placement solution, we use DP to further optimize pin accessibility and wirelength.

In advanced technology nodes, placement legality constraints are much more extensive and complex than cell non-overlapping (e.g., cell edge spacing constraint). Such constraints can cause placement violations between non-neighboring cell instances (e.g., c_{i-1} and c_{i+1}). Therefore, although we utilize the placement API from the commercial tool to check placement legality between neighboring cell instances, for each cell row, we perform row-based placement legality check after optimization and revert our changes to the row placement if any placement violation is observed.

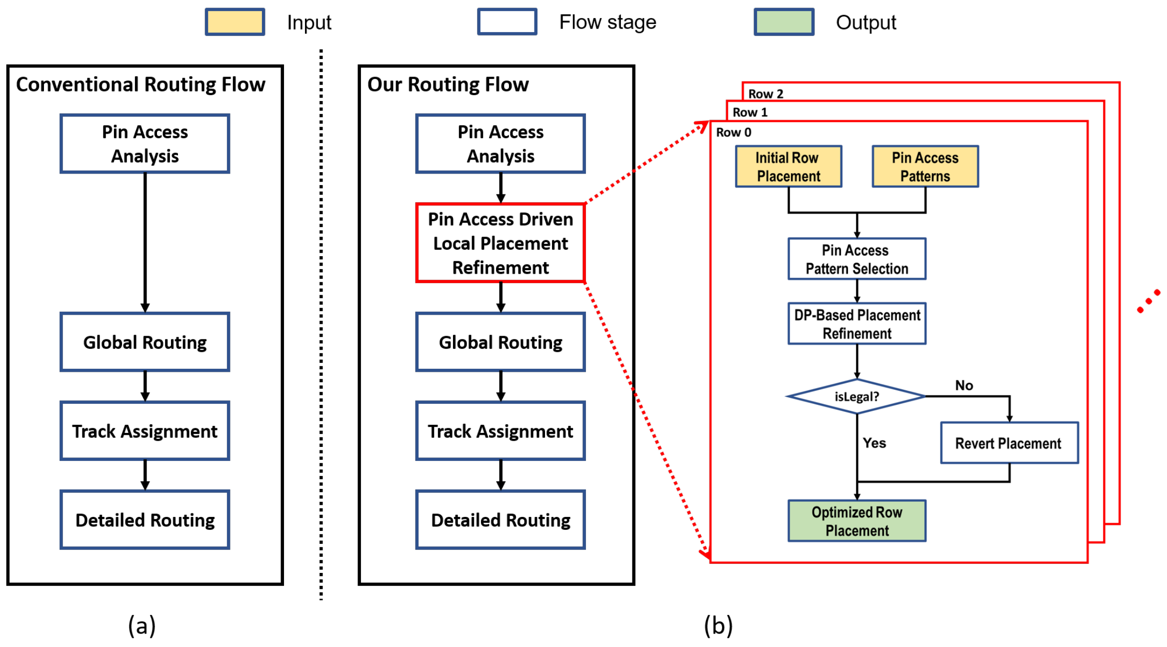


Figure 2.23: Illustrations of (a) conventional routing flow; and (b) our proposed routing flow with *in-route* pin access-driven detailed placement refinement.

2.2.3 Experiments

We integrate our *in-route* detailed placement optimization with a commercial tool and perform experiments in foundry technology nodes from 32nm to sub-5nm. We apply our detailed placement optimization to 19 industry designs. Design information is summarized in Table 2.11. We are not able to provide details of the design12 - design19 benchmarks due to product confidentiality constraints for these sub-5nm designs. All experiments are performed on Intel Xeon servers. Note that the placement solutions of these designs are already well-optimized by a leading commercial tool that considers pin accessibility in a node-specific manner during placement. All results are reported by the commercial tool.

Study of Displacement Constraint

To assess the impact of displacement (Disp.) constraint x_Δ, we sweep x_Δ from 0 to 4 (step size 1) for a sub-5nm design. Table 2.12 shows the difference in WNS, initial detailed routing violation count (#init. DRC), final routed wirelength (RWL) and percentage over-congested GCells (Overcon). With $x_\Delta \geq 2$, the timing, RWL and Overcon start to degrade versus $x_\Delta = 1$. This reflects imperfect correlation

Table 2.11: Benchmark information (omitting sub-$5nm$ Design12-19 cases).

Benchmark	**#stdcell**	**#macro**	**#net**	**Util. (%)**	**Node**
design1	119998	0	131514	78.385	$7nm$
design2	547180	20	566347	39.035	$7nm$
design3	521409	60	533777	36.371	$16nm$
design4	100005	0	111392	54.814	$16nm$
design5	213065	8	233781	38.244	$20nm$
design6	1790828	192	1811059	35.593	$20nm$
design7	815974	90	889974	64.043	$28nm$
design8	159429	0	197441	57.424	$28nm$
design9	337524	45	394336	65.887	$28nm$
design10	91667	0	101903	55.064	$28nm$
design11	43798	0	48171	54.855	$32nm$
design12-19	–				sub-$5nm$

of DP cost with routing and timing outcomes: max displacement range $x_\Delta = 1$ provides enough solution space for our optimization while honoring the already well-optimized original placement. We thus apply $x_\Delta = 1$ in all experiments below.

Table 2.12: Experimental results for displacement range constraint.

Disp.	**WNS** (ns)	**#init. DRC**	**RWL** (μm)	**Overcon** (%)
0	**-0.10**	241373	2952508	3.68
1	**-0.10**	239828	**2950170**	**3.63**
2	-0.12	238523	2954262	3.71
3	-0.11	**237391**	2953569	3.71
4	-0.11	241209	2957398	3.79

Main Results

We apply our pin access-driven dynamic programming-based optimization to all design blocks in Table 2.11. We use a WL weighting factor $\alpha = 0.01$ based on our empirical studies. Table 2.13 and Table 2.14 present experimental results, comparing the routing solutions based on (i) placement solution optimized by the commercial tool, and (ii) placement solution after further optimization by our methodology. Table 2.13 gives the comparison of timing, including worst negative slack (WNS) and total

Table 2.13: Comparison of worst negative slack (WNS), total negative slack (TNS), total power, optimization CPU time (Runtime) and detailed routing CPU time (DR runtime) between commercial tool (Comm.) and our work (Ours). Positive reduction values = improvements.

Benchmark	**Metrics**									
	WNS (ns)		**TNS** (ns)		**Total power** (mW)		**Runtime** (s)		**DR Runtime** (s)	
	Comm.	Ours	Comm.	Ours	Comm.	Ours	Comm.	Ours	Comm.	Ours
design1	**-0.161**	-0.174	-28.62	**-28.48**	20.56	**20.53**	–	13	10456	**7717**
design2	-0.201	**-0.197**	-14.45	**-13.34**	85.00	**84.97**	–	40	27609	**21833**
design3	-0.563	**-0.511**	-904.25	**-856.72**	188.50	**188.40**	–	91	34934	**23817**
design4	-0.122	**-0.087**	**-3.68**	-5.39	18.27	**18.27**	–	13	7590	**5696**
design5	-0.092	**-0.088**	-35.63	**-29.75**	288.40	**288.30**	–	16	8561	**6611**
design6	**-0.478**	-0.487	-1665.20	**-1367.50**	**957.50**	957.60	–	226	88134	**81295**
design7	-0.221	**-0.172**	-1405.20	**-646.07**	439.10	**438.90**	–	103	33822	**25147**
design8	-0.077	**-0.063**	-48.92	**-41.53**	45.39	**45.37**	–	14	8191	**6893**
design9	**-0.083**	-0.090	-65.97	**-65.31**	95.76	**95.74**	–	34	16969	**14436**
design10	-0.133	**-0.119**	-44.76	**-40.34**	28.17	**28.17**	–	5	4120	**3474**
design11	-0.085	**-0.080**	-17.54	**-15.71**	15.09	**15.08**	–	2	2014	**1499**
sub-5nm										
design12	-0.007	**-0.007**	-0.043	**-0.035**	84.2	**84.2**	–	35	1127	**1003**
design13	-0.063	**-0.054**	-18.1	**-16.934**	180.4	**180.4**	–	348	12556	**11417**
design14	-0.1	**-0.1**	-69.151	**-57.525**	198.1	**198.0**	–	245	9898	**9596**
design15	-0.048	**-0.042**	**-50.496**	-50.828	265.0	**265.0**	–	346	8768	**8395**
design16	-0.103	**-0.077**	-55.018	**-51.064**	294.0	**294.0**	–	372	11055	**9825**
design17	**-0.033**	-0.036	-9.567	**-8.192**	87.1	**87.1**	–	200	47758	**47153**
design18	-0.025	**-0.025**	-1.973	**-1.747**	89.5	**89.5**	–	178	17627	**16922**
design19	-0.038	**-0.033**	**-13.684**	-15.549	121.5	**121.5**	–	196	15960	**14332**
Avg. reduction (units)	0.01		60.01		–		–		–	
Avg. reduction (%)	–		–		0.03%		–		15.06%	

negative slack (TNS), total power, optimization CPU time, and subsequent detailed routing CPU time. Table 2.14 gives the comparison of initial detailed routing violation count (#init. DRC), final detailed routing violation count (#final DRC), routed wirelength, via count (#via) and percentage of over-congested GCell (Overcon) across all metal layers.

Table 2.13 confirms that the runtime of our optimization is negligible as compared to the runtime of detailed routing. Notably, the subsequent runtime of detailed routing is reduced by up to 31.82% (avg. 15.06%) with our *in-route* detailed placement optimization. With the negligible optimization runtime, we also reduce WNS by up to $52ps$ (avg. $10ps$), and reduce TNS by up to $759.13ns$ (avg. $60.01ns$). Moreover, we reduce total power by up to 0.15% (avg. 0.03%).

Based on Table 2.14, we observe that, for #init. DRC, we achieve improvement in 14 out of 19 testcases. We reduce #init. DRC up to 10.13% (avg. 1.28%), with similar #final DRC. For routed wirelength and via count, we achieve improvement in most testcases. We reduce routed wirelength by up

Table 2.14: Comparison of initial detailed routing violation count (#init. DRC), final detailed routing violation count (#final DRC), wirelength, via count (#via) and over-congested GCell (Overcon) between commercial tool (Comm.) and our work (Ours). Positive reduction values = improvements.

Benchmark	**Metrics**									
	#init. DRC		**#final DRC**		**Wirelength (μm)**		**#via**		**Overcon (%)**	
	Comm.	Ours	Comm.	Ours	Comm.	Ours	Comm.	Ours	Comm.	Ours
design1	2098	**2087**	7	**4**	1245216	**1242721**	1073763	**1069850**	0.07	**0.06**
design2	**4212**	4398	**1**	4	7059722	**7054360**	4765333	**4762265**	0.39	**0.39**
design3	9265	**9243**	**27**	31	9611743	**9605399**	5377754	**5377121**	0.19	**0.19**
design4	**1019**	1094	4	**1**	1231962	**1231019**	995369	**994380**	0.96	**0.96**
design5	123	**116**	18	**17**	5451472	**5446375**	1736060	**1733824**	0.06	**0.06**
design6	7122	**6838**	**60**	65	**93548447**	93584424	**20093121**	20099553	0.43	**0.41**
design7	10592	**10270**	**63**	71	16514279	**16503297**	6774719	**6747758**	0.01	**0.01**
design8	3714	**3660**	7	**3**	3416199	**3413156**	1546361	**1544693**	0.09	**0.09**
design9	1692	**1658**	32	**27**	9703209	**9697284**	**3063067**	3063454	0.01	**0.01**
design10	185	**171**	48	**42**	**1775344**	1775584	**702774**	703469	0.69	**0.66**
design11	79	**71**	**2**	3	656508	**655495**	341956	**340532**	0.42	**0.39**
sub-5nm										
design12	8721	**8829**	58	**55**	612379	**612304**	**672426**	673215	0.15	**0.15**
design13	204963	**203199**	21	**21**	2799052	**2798876**	**3979621**	3982364	8.78	**8.67**
design14	241373	**239828**	**15**	27	2952508	**2950170**	3934269	**3929686**	3.68	**3.63**
design15	176544	**176019**	12	**12**	**2972888**	2973352	**4099268**	4099386	7.82	**7.61**
design16	**235245**	235487	22	**22**	3181943	**3180857**	4176552	**4175833**	4.45	**4.42**
design17	194647	**192867**	1001	**908**	**894068**	895508	1999585	**1992910**	12.23	**11.88**
design18	**158588**	158639	158	**127**	896906	**896627**	1917938	**1917663**	3.79	**3.66**
design19	150608	**150389**	101	**90**	**943526**	944317	**2021737**	2022820	8.97	**8.76**
Avg. reduction (units)	–		6.68		–		–		0.06	
Avg. reduction (%)	1.28%		–		0.04%		0.09%		–	

to 0.20% (avg. 0.04%), and reduce via count by up to 0.42% (avg. 0.09%). For congestion, we reduce the percentage of over-congested GCell by up to 0.35% (avg. 0.06%). Note that these improvements are achieved over final, well-optimized placement solutions from a leading commercial tool.

2.2.4 Conclusion

In this work, we present an *in-route* dynamic programming-based pin access-driven detailed placement optimization methodology to significantly reduce the detailed routing runtime, with noticeable benefits in initial detailed routing DRC count, timing, and routed wirelength. We show that with integration of our *in-route* placement optimization, the detailed routing runtime can be reduced by up to 31.82% (avg. 15.06%) with up to 10.1% (avg. 1.28%) reduction in initial detailed routing DRCs across a wide spectrum of industry designs and technology nodes. Our ongoing research directions include (i) DRC-driven detailed placement refinement; and (ii) a more comprehensive timing-aware optimization flow considering different cell timing criticality characteristics.

2.3 Acknowledgments

Chapter 2 contains the reprints of Changho Han, Kwangsoo Han, Andrew B. Kahng, Hyein Lee, Lutong Wang and Bangqi Xu, “Optimal Multi-Row Detailed Placement for Yield and Model-Hardware Correlation Improvements in Sub-10nm VLSI”, *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2017; Changho Han, Andrew B. Kahng, Lutong Wang and Bangqi Xu, “Enhanced Optimal Multi-Row Detailed Placement for Neighbor Diffusion Effect Mitigation in Sub-10nm VLSI”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38(9), 2019; and Andrew B. Kahng, Jian Kuang, Wen-Hao Liu and Bangqi Xu, “In-Route Pin Access-Driven Placement Refinement for Improved Detailed Routing Convergence”, to appear in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021. The dissertation author is a main contributor to, and a primary author of, each of these papers.

I would like to thank my coauthors Changho Han, Kwangsoo Han, Andrew B. Kahng, Jian Kuang, Hyein Lee, Wen-Hao Liu and Lutong Wang, as well as the research support from Samsung Electronics.

Chapter 3

Essential Building Blocks towards DRC-Clean Routing

This chapter presents two works that provide essential elements toward DRC-clean routing solutions in advanced technology nodes. First, we present a geometry-based design rule check engine to meet routing tool needs in advanced technology nodes. The design rule checking engine is capable of capturing design rule violations in both academic and foundry technology nodes. The proposed engine includes a violation filtering mechanism that distinguishes between *detailed-routing-fixable* and *non-detailed-routing-fixable* violations, providing accurate design rule violation feedback for ripup-and-reroute in detailed routing. Our engine matches results from a commercial design rule checking tool in sub-14nm foundry enablement. Moreover, the incremental capability in our DRC engine enables further optimization in detailed routing for improved detailed routing convergence. Second, we present a multi-level pin access analysis framework with design rule satisfaction for advanced technology nodes. Our pin access analysis framework consists of cell pin-based access point generation, cell boundary conflict-aware access pattern generation, and dynamic programming-based access pattern selection for instance clusters. Our pin access analysis framework is capable of providing DRC-clean pin access for academic contest benchmark testcases as well as testcases in sub-14nm foundry technology nodes.

3.1 Geometry-Based Design Rule Checking for Detailed Routing

Design rule check is critical for electronic design automation enablement. Every new technology node comes with increasingly complicated design rules to satisfy patterning challenges (multiple litho and etch steps in patterning, CMP, OPC and other reticle enhancements, local layout effect and other variability sources) for smaller sizes [53].

Although design rule checking has been studied for more than thirty years, to the best of our knowledge, a comprehensive documentation of implementation in the context of advanced-node detailed routing is still missing. Most of VLSI handbooks [4][80][86] cover from floorplanning to detailed routing and manufacturability, without discussions of design rule check. The handbook of [84] covers low-level geometric algorithms and data structures for design rule checking, with a handful of references that provide foundations for design rule check algorithms. However, the book and its references do not describe any high-level, end-to-end frameworks that can be used in detailed routing. Several patents [18][19][30][65][69][95] describe point implementation of specific data structures and methodologies to handle various design rules in advanced nodes. However, the description of these techniques may not be complete, and are strongly tied to internal implementation of commercial tools. As a result, most recent academic detailed routers [12][13][49] attempt to apply correct-by-construction approach with design rule models to reduce design rule check (DRC) violations. In the context of advanced node detailed routing, to the best of our knowledge, no previous publications has ever discussed (1) the set of data structures needed for a specific set of design rules; (2) the methodology to check LEF-based design rules instead of pure polygon-based, post-route, foundry representations; and (3) the capability to identify and properly distinguish *detailed-routing-fixable* DRCs and *non-detailed-routing-fixable* DRCs.

The open-source TritonRoute [124] with its integrated design rule check is the only academic detailed router that makes a claim for capability to detect and fix design rule violations. However, the scalability and solution quality of the TritonRoute DRC is limited.

Given the above, a scalable design rule checking methodology would provide strong support to existing academic detailed routers, and help enable academic detailed routers on commercial technology nodes. The ISPD-2018 and ISPD-2019 benchmark suites provides testcases derived from $65nm$, $45nm$

and $32nm$ technology nodes with simplified design rules. The ISPD-2019 benchmark suite includes all design rules presented in the ISPD-2018 benchmark suit, and provide more realistic and more complicated design rules for evaluation of design rule checking capability.

Based on the ISPD contest benchmark suites, we present a geometry-based design rule check for detailed routing. Our main contribution is a geometry-based design rule checking framework which is capable of identifying *detailed-routing-fixable* DRCs. Our design rule check is capable of matching DRCs from commercial tools in TSMC65LP technology node. Highlights of our work are summarized as follows.

- We propose a geometry-based design rule checking methodology for advanced design rules. The proposed method is capable of capturing design rule violations in commercial technology nodes.
- We propose a design rule violation filtering method. To the best of our knowledge, our proposed method is the first in open literature that can distinguish between *detailed-routing-fixable* violations and *non-detailed-routing-fixable* violations. Hence, our proposed method can be used to provide accurate design rule violation feedback to a ripup-and-reroute engine and speed up detailed routing convergence.
- We integrate our design rule check engine into the open-source TritonRoute. We show that the overall DRC count is reduced by up to 100% (avg. avg. 93.54%) and runtime is reduced by up to 50.28% (avg. 17.84%) when the feedback from our design rule check engine is considered.

3.1.1 Preliminaries

In this section, we describe geometry objects, design database objects, and the design rules that are supported.

Geometry objects

A *geometry object* refers to a specific type of 2D Manhattan shape(s). The basic Manhattan shapes include **Segment**, **Rectangle** and **Polygon (with holes)**. In this work, we use these three basic shapes as follows:

Polygon edge: the edge of a polygon. A polygon edge consists of two consecutive points in the exterior (or interior) ring of a polygon, represented by using the segment geometry type. Each polygon edge is tied to the two polygon corners which are the endpoints of the polygon edge.

Polygon corner: the corner of a polygon. A polygon corner is composed of two consecutive polygon edges. Each corner is tied to the two polygon edges which it is connected to.

Max rectangle: a maximal rectangle inside a polygon. For a given rectangle, the unique max rectangle is itself. In a polygon, there can be more than one max rectangles.

Polygon set: the union of polygons. The resulting polygon set holds zero or more disjoint polygons, with or without holes. The polygon set has an efficient data structure for polygon boolean operations (intersection and merging).

Database objects

Table 3.1: Database objects.

Object	Meaning	Status	Geometries
instTerm	cell pin	fixed	polygon(s)
term	block IO pin	fixed	polygon(s)
instBlockage	cell blockage	fixed	polygon(s)
blockage	block blockage	fixed	polygon(s)
pathSeg	regular net wire	routing	rectangle
pathSeg	special net wire	fixed	rectangle
via	regular net via	routing	rectangle(s)
via	special net via	fixed	rectangle(s)
patchMetal	regular net patch metal	routing	rectangle
patchMetal	special net patch metal	fixed	rectangle

The type of database objects in a typical physical design database is summarized in Table 3.1. Each type of database object holds one or more geometry objects. Each type of database object has a default *status* property – *fixed* or *routing*. The status property is crucial for design rule checking used in detailed routing because not all design rule violations are equal. To obtain high quality of results during detailed routing, only *detailed-routing-fixable* markers should be reported. For example, a violation between *fixed* objects may not be fixable, and thus should be ignored for fast convergence. We describe the methodology to filter violation markers in Section 3.1.3.

Design rules

Table 3.2: Design rules.

```
// metal layer
WIDTH defaultWidth ;
[MINWIDTH minWidth ;]
SPACINGTABLE
 PARALLELRUNLENGTH {length} ...
  {WIDTH width  {spacing} ...} ...  ;
[SPACING minSpacing SAMENET [PGONLY] ;]
[MINSTEP minStepLength [MAXEDGES maxEdges] ;]
[SPACING eolSpacing ENDOFLINE eolWidth WITHIN eolWithin
 [PARALLELEDGE parSpace WITHIN parWithin [TWOEDGES] ;] ...
[CORNERSPACING
  {CONVEXCORNER | CONCAVECORNER} [EXCEPTEOL eolWidth]
  {WIDTH width SPACING spacing ;} ...  ]  ...  ;
// cut layer
{SPACING cutSpacing [CENTERTOCENTER]
 [  ADJACENTCUTS numCuts WITHIN cutWithin [EXCEPTSAMEPGNET]
  | PARALLELOVERLAP
  | AREA cutArea] ;}...
[SPACING cutSpacingSN [CENTERTOCENTER] SAMENET ;]
```

The (industry-standard) LEF syntax [117] seen in the ISPD-2018 [66] and the ISPD-2019 [61] benchmark suites is summarized in Table 3.2, where each *italic* word indicates a numerical value. Note the separate metal and cut layer rules.

Minimum width specifies the minimum width for a polygon. By slicing a polygon into rectangles (in both directions), the length along the slicing direction of any sliced rectangle must be greater than or equal to *minWidth*. If MINWIDTH is not specified, *defaultWidth* is used. Figure 3.1 shows polygon slicing and the critical dimension to check against *minWidth*.

Metal short specifies the short violation between two max rectangles of different nets if the two max rectangles overlap.

Non-sufficient-metal overlap specifies the minimum diagonal length in case of metal overlaps. If two max rectangles of the same net overlap, then the overlapping rectangle must have diagonal length greater than or equal to *minWidth*, shown in Figure 3.2.

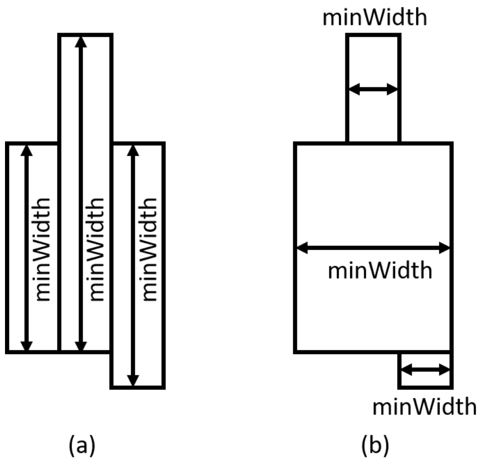


Figure 3.1: Minimum width: (a) polygon sliced vertically; and (b) polygon sliced horizontally.

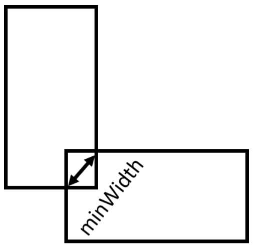


Figure 3.2: Non-sufficient-metal overlap.

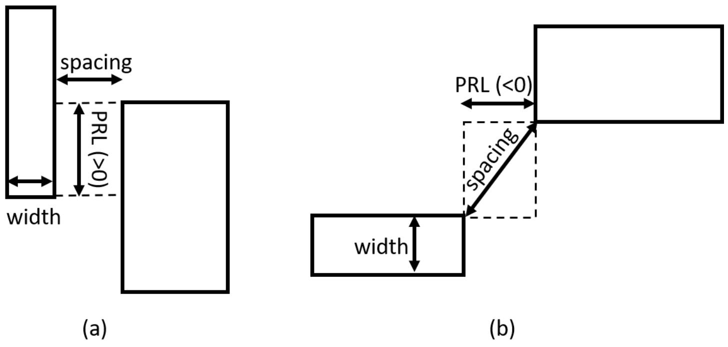


Figure 3.3: Parallel run length spacing: (a) positive PRL; and (b) negative PRL.

Parallel run length (PRL) spacing specifies the width- and parallel run length-dependent spacing between two max rectangles. If the maximum width of the two max rectangles is greater than *width*, and the parallel run length is greater than *length*, then the spacing between the two max rectangles must be greater than or equal to *spacing*. The first spacing value is the minimum spacing for a given width even if the PRL is not met. If SAMENET spacing is specified, then the spacing between the two max rectangles must be greater than or equal to the minimum of *spacing* and *minSpacing*. If PGONLY is specified, then *minSpacing* is only used if the two max rectangles belong to the same power or ground net. Figure 3.3 illustrates the spacing for positive and negative PRLs.

Minimum step specifies the shortest polygon edge length. The polygon edge length must be greater than or equal to *minStepLength*. If MAXEDGES is specified, then up to *maxEdges* consecutive edges that are less than *minStepLength* is allowed. A *maxEdges* value of 0 is equivalent to not specifying MAXEDGES statement.

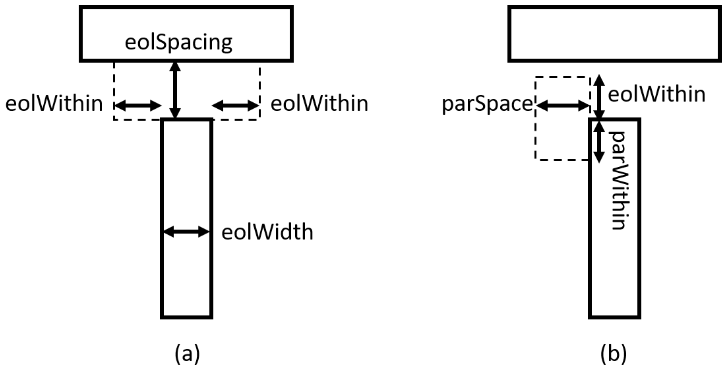


Figure 3.4: End-of-line spacing: (a) illustration of *eolWidth*, *eolWithin* and *eolSpacing*; and (b) illustration of *parWithin* and *parSpace*.

End-of-line (EOL) spacing specifies the spacing from an EOL edge to the exterior of the polygon. An EOL edge is a polygon edge that is shorter than *eolWidth*. The spacing to the exterior of polygon must be greater than or equal to *eolSpacing* anywhere within (less than) *eolWithin*, as shown in Figure 3.4(a). If PARALLELEDGE is specified, then the rule is applied only if there is a parallel edge (or two parallel edges if TWOEDGES is specified) that is (are) less than *parSpace* away, and is (are) also less than *parWithin* from the EOL edge, and *eolWithin* beyond the EOL edge, as shown in Figure 3.4(b).

Corner spacing rule specifies the spacing from a corner to the exterior of a polygon. CONVEX-CORNER (resp. CONCAVECORNER) specifies that the rule only applies to convex (resp. concave) corners. EXCEPTEOL specifies that if the corner is connected to an EOL edge that is shorter than *eolWidth*, then the rule does not apply. For the spacing table lookup, corner spacing rule works in a similar way as for PRL spacing rule except that (i) the rule only applies for non-positive PRL values and (ii) the width only account for the exterior of a polygon.

Cut short specifies a short if the two cuts overlap.

Cut spacing specifies the minimum spacing between two cuts. If CENTERTOCENTER is specified, then cutSpacing and cutWithin are calculated from cut center to cut center, otherwise, it is calculated from cut edge to cut edge. If ADJACENTCUTS is specified, then the rule is applied only if there are *numCut* cuts that are less than *cutWithin* distance. If EXCEPTSAMEPGNET is specified, then the rule is applied only if the two cuts are not on the same power or ground net. If PARALLELOVERLAP is specified, then the rule is applied only if the two cuts have a parallel run length greater than 0. If AREA is specified, then the rule is applied only if any of the two cuts is greater than or equal to *cutArea*. If SAMENET is specified, then spacing between the two cuts must be greater than or equal to the minimum of *cutSpacing* and *cutSpacingSN*.

3.1.2 Overall Flow

Given design database, bounding box and layer range, we first get all database objects, then initialize the necessary data structures to hold physical layout information. Next, we perform design rule checking and output *detailed-routing-fixable* design rule violation markers. A *marker* consists of a bounding box, layer number, violating net(s) and violation type.

Input: design database, design rule checking bounding box and layer range

Constraints: design rules

Output: *detailed-routing-fixable* design rule violation markers

Data structure

Figure 3.5 shows the data structure of layout information for design rule checking. On the top level, the layout information is organized per net, then organized per layer. Metal layer and cut layer are organized differently.

For the layout information of a net on a metal layer, we initialize two polygon sets. *Polygon set (fixed)* is generated by applying boolean OR operation to geometry objects from fixed database objects. *Polygon set (routing)* is generated similarly from routing database objects. We then merge the two polygon sets into one and decompose it into disjoint polygons. Each disjoint polygon holds all the max rectangles,

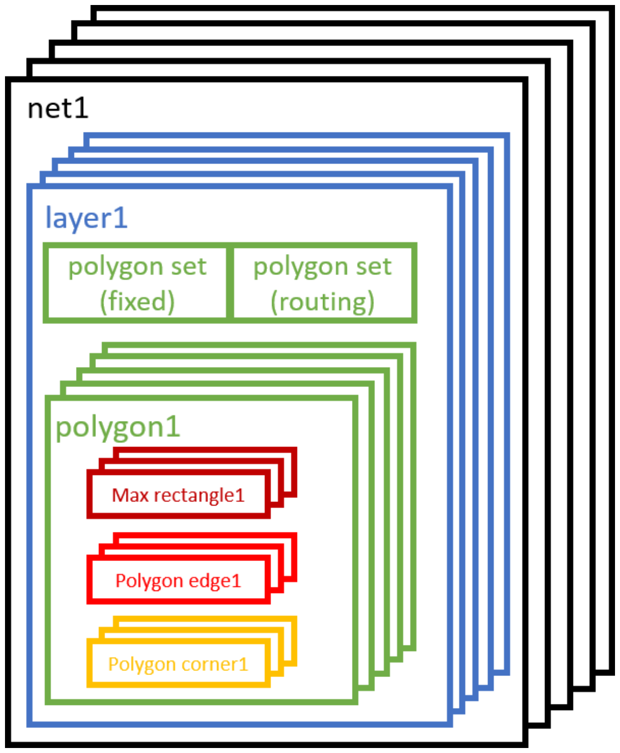


Figure 3.5: Data structure of layout information.

polygon edges and corners. Each max rectangle and polygon edge or polygon corner is marked with either *fixed* status or as *routing* status. As long as the max rectangle, polygon edge or polygon corner can be derived from polygon set (fixed), the shape is marked as *fixed*, otherwise it is marked as *routing*.[12]

For the layout information of a net on a cut layer, since each cut is supposed to be disjoint and rectangular, we skip the merging and decomposition steps. Each cut directly forms a polygon, holding one max rectangle, four polygon edges and four polygon corners. Overall, since polygon sets, polygons, max rectangles, polygon edges and polygon corners are all represented using vertex coordinates, the memory footprint is linear to the number of vertices of all geometries.

Region query

After initialization of data structures, we build region queries for max rectangles and polygon edges.

[12]A *routing* rectangle, polygon edge or polygon corner might still cause a *non-fixable* violation due to the complicated filtering process. Here the marking only serves to accelerate the filtering process.

Given a layer number and a bounding box, the region query engine returns all touching max rectangles (or polygon edges). For fast operation, the region query engine only handles rectangular geometry objects, instead of polygons. In this work, we use R-trees from Boost for region queries.

3.1.3 Design Rule Checking

We now describe design rule checking and filters.

Metal spacing

Metal spacing rules consist of short, non-sufficient-metal overlap and parallel run length spacing. The rule checking starts with a max rectangle m. In Algorithm 8, given m, we first query all neighboring max rectangles within *maxDist* that could possibly cause design rule violations. For each max rectangle pair (m, n), if they overlap and are of same net, we check non-sufficient metal overlap; if they overlap but are of different net, we check metal short; otherwise, we check parallel run length spacing.

Algorithm 8 Check metal spacing

```
1:  Input: max rectangle $m$
2:  $N \leftarrow$ queryMaxRectangles($m$, $maxDist$)
3:  for all $n \neq m$ in $N$ do
4:      if isOverlap($m$, $n$) then
5:          if getNet($m$) = getNet($n$) then
6:              checkNSMetal($m$, $n$)
7:          else
8:              checkMetalShort($m$, $n$)
9:          end if
10:     else
11:         checkPRL($m$, $n$)
12:     end if
13: end for
```

Algorithm 9 describes the methodology to check short. Line 2 gets the short violation bounding box. In Lines 3 – 5, we skip the violation if both max rectangles are *fixed*. Lines 6 – 8 deal with a special handling in LEF where metal short with blockage is allowed if it occurs fully within a cell pin, as shown in Figure 3.6. In Lines 9 – 11, we skip the violation if polygon set (routing) does not intersect with the short region.

Algorithm 9 Check metal short

```
1: Input: max rectangles $m$, $n$
2: $shortRect \leftarrow$ getIntersection($m$, $n$)
3: if isFixed($m$) AND isFixed($n$) then
4:     return
5: end if
6: if isCoveredByPin($shortRect$) AND isBlockage($m$, $n$) then
7:     return
8: end if
9: if not hasRouting($shortRect$) then
10:     return
11: end if
12: addMarker(MetalShort)
```

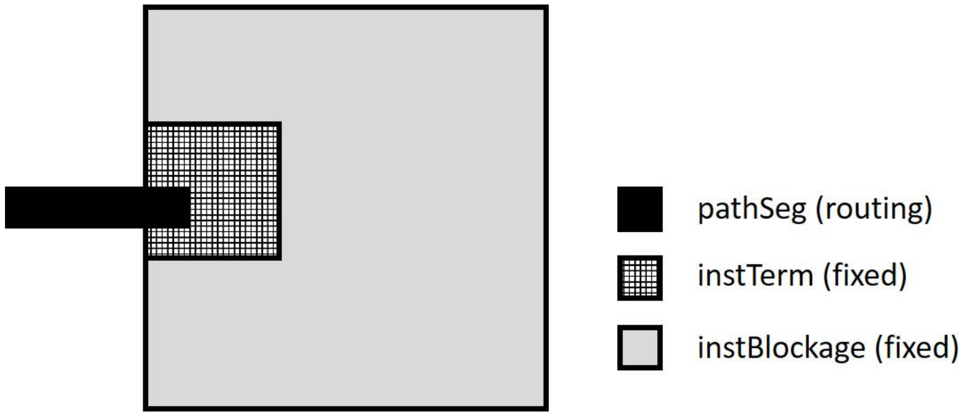


Figure 3.6: Metal short filter: short area is within pin.

Algorithm 10 describes the methodology to check non-sufficient-metal overlap. Line 2 gets the overlapping metal bounding box. Lines 3 – 5 check if there is sufficient metal overlap. In Lines 6 – 8, we skip the violation if any max rectangle has a width less than *minWidth* because such max rectangle is the pure result of polygon decomposition, and does not fully cover any database objects. In this work, *minWidth*-related violations are captured by *minWidth* rule checking in Algorithm 12. Note that *minWidth* rule checking is based on sliced rectangles instead of max rectangles. In Lines 9 – 11, we skip the violation if the two max rectangles are covered by a 3rd max rectangle. The 3rd max rectangle must be of the same net and wider than *minWidth* to serve as a bridge. Figures 3.7(a) and (b) illustrate the above two cases.

Algorithm 11 describes the methodology to check parallel run length spacing. Lines 2 and 3 get the actual and required spacing. Depending on whether the two max rectangles are of the same net, or

Algorithm 10 Check non-sufficient metal overlap

```
1: Input: max rectangles $m$, $n$
2: $nsRect \leftarrow$ getIntersection($m$, $n$)
3: if diagLen($nsRect$) $\geq minWidth$ then
4:     return
5: end if
6: if width($m$) $< minWidth$ OR width($n$) $< minWidth$ then
7:     return
8: end if
9: if hasValid3rdObj($nsRect$) then
10:     return
11: end if
12: addMarker(NonSufficientMetalOverlap)
```

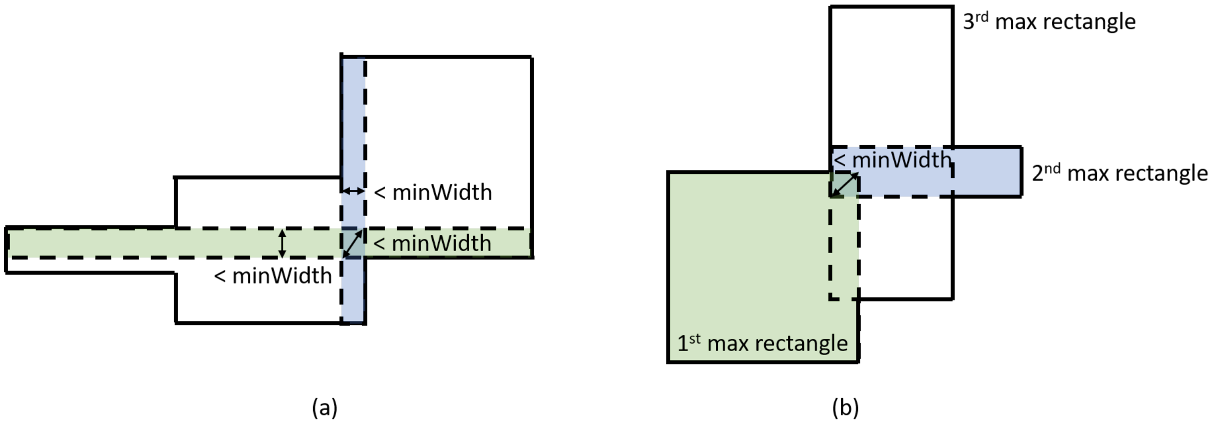


Figure 3.7: Non-sufficient-metal overlap filters: (a) max rectangles narrower than *minWidth*; and (b) two max rectangles bridged by a 3rd max rectangle.

at least one of them are blockages, the required spacing value can be overridden by same-net spacing or minimum spacing. Lines 4 – 6 check if parallel run length spacing is satisfied. In Lines 7 – 9, we skip the violation if both max rectangles are *fixed*. Line 10 calculates the generalized intersection of the two disjoint max rectangles. In Lines 11 – 13, we skip the violation if the generalized intersection does not overlap with specific number(s) of valid polygon edges. If the spacing direction is diagonal, then any polygon edge is valid; otherwise only polygon edges orthogonal to the spacing direction is valid. Figure 3.8 shows two same-net max rectangles (light green and light blue) decomposed from a single polygon. We skip the violation because on the orthogonal direction of spacing, there is no polygon edge overlapping with the generalized intersecting region. In Lines 14 – 17, we skip the violation if the polygon sets (routing) minus the polygon sets (fixed) of the two max rectangles do not intersect with the generalized intersecting region. In such case, *fixed* geometries result in a *non-detailed-routing-fixable* violation. For example, in Figure 3.8, we skip the violation if no area in the darker green or blue region is exclusively from polygon set (routing).

Algorithm 11 Check parallel run length spacing

```
1:  Input: max rectangles $m$, $n$
2:  $actVal \leftarrow$ getActualSpacing($m$, $n$)
3:  $reqVal \leftarrow$ getRequiredSpacing($m$, $n$)
4:  if $actVal \geq reqVal$ then
5:      return
6:  end if
7:  if isFixed($m$) AND isFixed($n$) then
8:      return
9:  end if
10: $prlRect \leftarrow$ getIntersection($m$, $n$)
11: if not hasPolyEdge($prlRect$) then
12:     return
13: end if
14: $maxWidth \leftarrow$ getMaxWidth($m$, $n$)
15: if not hasExclusiveRoutingWithin($prlRect$, $maxWidth$) then
16:     return
17: end if
18: addMarker(ParallelRunLengthSpacing)
```

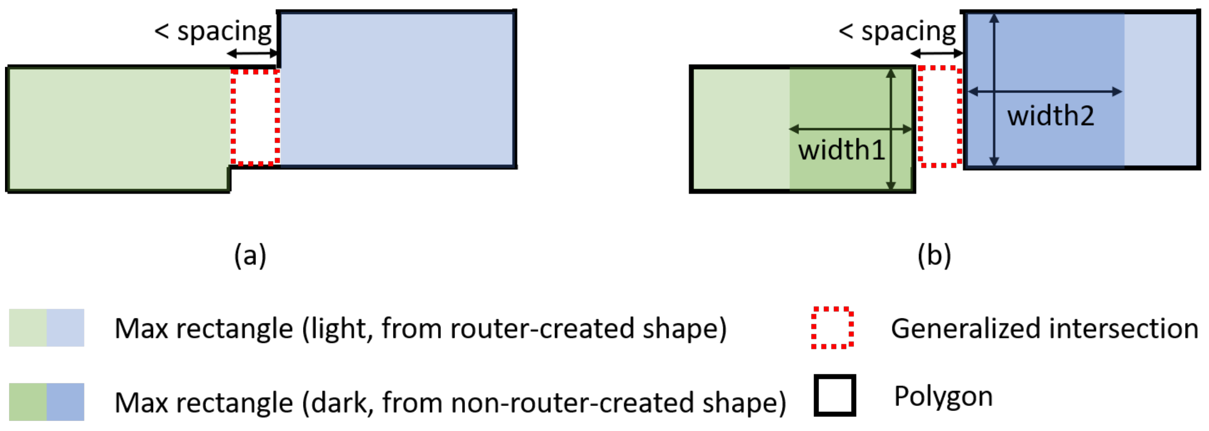

< spacing
< spacing
width1
width2
(a)
(b)
Max rectangle (light, from router-created shape)
Max rectangle (dark, from non-router-created shape)
Generalized intersection
Polygon

Figure 3.8: Parallel run length spacing filter.

Metal Shape

Metal shape rules consist of minimum width and step.

Algorithm 12 describes the design rule checking for minimum width. In Lines 2 – 11, given a polygon, we first slice the polygon vertically and check each sliced polygon separately. Lines 4 – 6 check whether the shape satisfies the minimum width. In Lines 7 – 9, we skip the violation if the sliced rectangle does not overlap with the polygon set (routing). Lines 12 – 20 repeat the above procedure but by slicing in the horizontal direction.

Algorithm 12 Check minimum width

```
1: **Input**: polygon $m$
2: $N \leftarrow$ slicePolygon($m$, $vertical$)
3: **for all** $n$ in $N$ **do**
4:     **if** $ySpan(n) \geq minWidth$ **then**
5:         return
6:     **end if**
7:     **if** not hasRouting($n$) **then**
8:         return
9:     **end if**
10:    addMarker(MinimumWidth)
11: **end for**
12: $N \leftarrow$ slicePolygon($m$, $horizontal$)
13: **for all** $n$ in $N$ **do**
14:     **if** $xSpan(n) \geq minWidth$ **then**
15:         return
16:     **end if**
17:     **if** not hasRouting($n$) **then**
18:         return
19:     **end if**
20:     addMarker(MinimumWidth)
21: **end for**
```

Algorithm 13 describes the design rule checking for minimum step. A minimum step consists of consecutive shorter-than-*minStepLength* edge(s) between two different not-shorter-than-*minStepLength* edges. In Lines 2 – 16, we get the first and last polygon edges that are larger than *minStepLength*, with all intermediate edges shorter than *minStepLength*. In Lines 17 to 19, we skip the violation if the first and last edges are the same. In Lines 20 – 22, we check whether the number of short edges is allowed. In Lines 23 – 25, we skip the violation if the bounding box of short edges does not intersect with any routing object.

Algorithm 13 Check minimum step

```
1: Input: polygon edge $e$
2: if length($e$) $<$ $minStepLength$ then
3:    return
4: end if
5: initializeBBox($bbox$, endPoint($e$))
6: $beginEdge \leftarrow e$
7: $numEdges \leftarrow 0$
8: while $beginEdge \neq$ nextEdge($e$) do
9:    $e \leftarrow$ nextEdge($e$)
10:   updateBBox($bbox$, endPoint($e$))
11:   if length($e$) $<$ $minStepLength$ then
12:      $numEdges \leftarrow numEdges + 1$
13:   else
14:      break
15:   end if
16: end while
17: if $e = beginEdge$ then
18:    return
19: end if
20: if $numEdges <= maxEdges$ then
21:    return
22: end if
23: if not hasRoute($bbox$) then
24:    return
25: end if
26: addMarker(MinimumStep)
```

End-of-line spacing

Algorithm 14 describes the design rule checking for end-of-line spacing. Lines 2 – 4 check whether the input edge is an EOL edge. Lines 5 – 7 check if there exists parallel edge(s) in case the rule contains the PARALLELEDGE statement. Lines 8 – 18 check all potential EOL spacing violations between the EOL edge and an opposite edge on the exterior side of the polygon. In Lines 11 and 13, we skip the violation if the generalized intersection of the EOL edge and the opposite edge contains any object since anything in between, results in a finer EOL violation. In Lines 14 – 16, we skip the violation if none of the EOL edge, opposite edge, or parallel edge(s) if any are the results of routing objects. Figure 3.9 shows an EOL violation between two *fixed* geometries, only given the existence of a parallel edge from a *routing* object.

Algorithm 14 Check end-of-line spacing

```
1: Input: polygon edge $e$
2: if len($e$) $\geq$ $eolWidth$ then
3:     return
4: end if
5: if not hasParallelEdge($e$) then
6:     return
7: end if
8: $E \leftarrow$ queryPolygonEdge($e$, $eolWithin$, $eolSpacing$)
9: for all $e'$ in $E$ do
10:     $eolRect \leftarrow$ getIntersection($e$, $e'$)
11:     if not isEmpty($eolRect$) then
12:         return
13:     end if
14:     if not hasRoute($e$) then
15:         return
16:     end if
17:     addMarker(EndOfLineSpacing)
18: end for
```

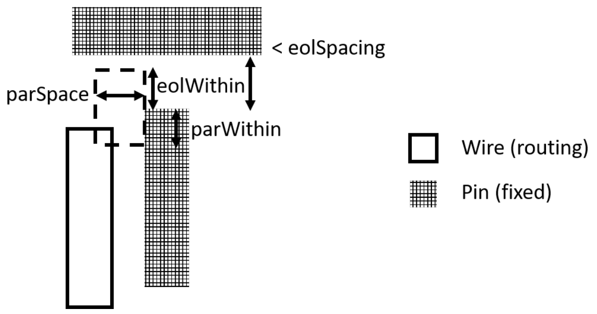


Figure 3.9: End-of-line spacing violation between *fixed* EOL edge, and *fixed* opposite edge, given the existence of a parallel edge from a *routing* object.

Corner Spacing

Algorithm 15 describes the design rule checking procedure for corner spacing. Lines 2–4 check whether the input corner c has the same corner type specified in the rule. Lines 5–7 check whether the input corner connects to an edge that meets the EOL exception condition. Line 8 queries all max rectangles which potentially have violation with c. For all queried max rectangles, Lines 10–12 check whether each max rectangle is overlapped with c or the max rectangle has non-positive PRL with c. In lines 13–15, we skip the max rectangle if both the max rectangle and c are fixed. Lines 16–22 compare required spacing value and actual spacing value, and add a DRC marker for corner spacing accordingly.

Algorithm 15 Check corner spacing

```
1: Input: polygon corner $c$
2: if $c$.type $\neq$ $cornerType$ then
3:     return
4: end if
5: if len($c$.prevEdge) $<$ $eolWidth$ OR len($c$.nextEdge) $<$ $eolWidth$ then
6:     return
7: end if
8: $N \leftarrow$ queryMaxRectangles($c$, $maxDist$)
9: for all $n$ in $N$ do
10:     if isOverlap($c$, $n$) OR hasPositivePRL($c$, $n$) then
11:         continue
12:     end if
13:     if isFixed($c$) AND isFixed($n$) then
14:         continue
15:     end if
16:     $prlRect \leftarrow$ getIntersection($c$, $n$)
17:     $reqVal \leftarrow$ getRequiredSpacing($n$.width)
18:     $actVal \leftarrow$ maxXY($prlRect$)
19:     if $actVal \geq reqVal$ then
20:         continue
21:     end if
22:     addMarker(CornerSpacing)
23: end for
```

Cut spacing

Check cut spacing follows a similar procedure shown in Algorithm 8 to first identify neighboring cuts, and to identify whether the two cuts potentially short or violate cut spacing. Algorithm 16 describes the design rule checking if the two cuts do not short. In Lines 2 – 6, we check whether the cuts satisfy the spacing. Lines 7 – 9 skip the violation if both cuts are *fixed*. In Lines 10 – 18, we skip the violation if the cut spacing rule has ADJACENTCUTS/PARALLELOVERLAP/AREA but the layout does not satisfy these constraints.

Algorithm 16 Check cut spacing

```
1: Input: cuts $m$, $n$
2: $actVal \leftarrow$ getActualSpacing($m$, $n$)
3: $reqVal \leftarrow$ getRequiredSpacing($m$, $n$)
4: if $actVal \geq reqVal$ then
5:     return
6: end if
7: if isFixed($m$) AND isFixed($n$) then
8:     return
9: end if
10: if not hasAdjCuts($m$) then
11:     return
12: end if
13: if not hasParallelOverlap($m$, $n$) then
14:     return
15: end if
16: if not hasArea($m$, $n$) then
17:     return
18: end if
19: addMarker(CutSpacing)
```

Incremental DRC Checking Capability

Due to the net-by-net nature of ripup-and-reroute, it is desired for the DRC engine have incremental capability to update and check after the routing of a given net is modified. Incremental capability of a DRC engine can further enable optimizations in routing (see Chapter 4). In our work, incremental DRC checking for a modified net can be achieved by (i) updating the layout data structure of the modified net in the DRC engine; and (ii) perform DRC checking and filtering for the modified net only, which can be achieved by skipping DRC checking if the input object(s) of Algorithms 8–16 does not belong to the modified net.

3.1.4 Experiments

We implement our design rule checking in C++ with Boost C++ libraries. We use the Boost Polygon library for polygon manipulation and Boost Geometry Index Rtree for region query. We perform the experiments by integrating our code into the open-source TritonRoute-WXL [123]. The experiments are performed with eight threads, using the testcases from ISPD-2018 and ISPD-2019 initial detailed routing contest [61][66]. The testcase information is summarized in Table 3.3.

Table 3.3: Testcases [61][66].

Benchmark	#std	#blk	#net	#pin	#layer	Die size	Tech. node
ISPD-2018							
ispd18_test1	8879	0	3153	0	9	0.20×0.19mm^2	45nm
ispd18_test2	35913	0	36834	1211	9	0.65×0.57mm^2	45nm
ispd18_test3	35973	4	36700	1211	9	0.99×0.70mm^2	45nm
ispd18_test4	72094	0	72401	1211	9	0.89×0.61mm^2	32nm
ispd18_test5	71954	0	72394	1211	9	0.93×0.92mm^2	32nm
ispd18_test6	107919	0	107701	1211	9	0.86×0.53mm^2	32nm
ispd18_test7	179865	16	179863	1211	9	1.36×1.33mm^2	32nm
ispd18_test8	191987	16	179863	1211	9	1.36×1.33mm^2	32nm
ispd18_test9	192911	0	178857	1211	9	0.91×0.78mm^2	32nm
ispd18_test10	290386	0	182000	1211	9	0.91×0.87mm^2	32nm
ISPD-2019							
ispd19_test1	8879	0	3153	0	9	0.15×0.15mm^2	32nm
ispd19_test2	72094	4	72410	1211	9	0.87×0.59mm^2	32nm
ispd19_test3	8283	4	8953	57	9	0.20×0.20mm^2	32nm
ispd19_test4	146442	7	151612	4802	5	1.60×1.55mm^2	65nm
ispd19_test5	28920	6	29416	360	5	0.91×0.91mm^2	65nm
ispd19_test6	179881	16	179863	1211	9	1.36×1.33mm^2	32nm
ispd19_test7	359746	16	358720	2216	9	1.58×1.52mm^2	32nm
ispd19_test8	539611	16	537577	3221	9	1.80×1.71mm^2	32nm
ispd19_test9	899341	16	895253	3221	9	2.01×2.15mm^2	32nm
ispd19_test10	899404	16	895253	3221	9	2.01×2.15mm^2	32nm

To assess the benefit of the design rule checking, we perform detailed routing on the 20 ISPD-2018 and ISPD-2019 contest benchmark testcases. We run detailed routing for ten iterations with and without the design rule violation (DRC) marker cost, and compare the corresponding violation counts. The result is shown in Table 3.4 where **w/o DRC** refers to the detailed routing results without DRC marker cost and **w/ DRC** refers to the detailed routing results with DRC marker cost. We can observe that when detailed routing fully utilizes the DRC marker information from the DRC engine, the detailed routing not only converges better in terms of DRC count, but also finishes in shorter runtime. With the DRC marker information, the detailed routing finishes with up to 100% (avg. 93.54%) DRC reduction and the runtime for ten iterations of detailed routing is reduced by up to 50.28% (avg. 17.84%).

Table 3.4: Detailed routing comparison of wirelength, via count, DRC count and runtime between ten iterations of detailed routing without the design rule violation marker cost (w/o DRC) and with the design rule violation marker cost (w/ DRC).

Benchmark	**Wirelength** (μm)		**Via count**		**DRC count**		**Runtime** (s)	
	w/o DRC	w/ DRC	w/o DRC	w/ DRC	w/o DRC	w/ DRC	w/o DRC	w/ DRC
ispd18_test1	86412	86440	35378	35406	2	**0**	34	**23**
ispd18_test2	1572614	1572819	359598	359982	9	**0**	272	**188**
ispd18_test3	1750609	1751250	355230	355692	179	**158**	320	**298**
ispd18_test4	2620519	2621231	721747	723862	251	**84**	532	**887**
ispd18_test5	2763934	2763875	888616	889397	134	**0**	656	**493**
ispd18_test6	3552063	3551801	1368425	1369517	12	**0**	909	**758**
ispd18_test7	6475774	6475058	2228360	2228527	204	**2**	1662	**1391**
ispd18_test8	6504479	6503655	2244665	2245489	219	**0**	1561	**1328**
ispd18_test9	5434518	5433658	2238245	2238810	74	**0**	1423	**1255**
ispd18_test10	6759337	6760074	2416542	2420093	498	**17**	2536	**1636**
ispd19_test1	62930	63151	36786	37194	108	**0**	122	**174**
ispd19_test2	2467044	2470886	781667	787290	974	**1**	2583	**1682**
ispd19_test3	82065	82411	63334	63852	341	**2**	421	**209**
ispd19_test4	2998959	3001424	1038410	1046033	253	**0**	943	**768**
ispd19_test5	474344	474240	165302	165477	22	**0**	101	**68**
ispd19_test6	6533158	6537203	1922912	1928029	808	**5**	4003	**3376**
ispd19_test7	12148272	12157094	4468531	4511492	1440	**0**	7754	**6589**
ispd19_test8	18685390	18694595	6915128	6980779	4061	**4**	10751	**5457**
ispd19_test9	28264059	28280151	11467368	11581563	6412	**4**	15028	**12613**
ispd19_test10	27936501	27957722	11563181	11712563	6518	**100**	17362	**9508**

3.1.5 Conclusion

In this work, we present a new geometry-based design rule checking for detailed routing. Our methodology supports design rule checking in the technology nodes from ISPD initial detailed rouitng contests, and is capable of identifying detailed-routing-fixable violations. We integrate our design rule checking into the open-source TritonRoute. We show up to 100% (avg. 93.54%) DRC reduction and up to 50.28% (avg. 17.84%) detailed routing runtime reduction when the feedback from the DRC engine is considered. Our future research directions include: (i) support of color-aware design rule checking and (ii) DRC engine-based design rule violation fixing for shape-related violations (e.g., minimum step violation).

3.2 Pin Access Analysis Framework for Detailed Routing

Pin accessibility has been one of the major crucial issues [3][82] in advanced node enablement. Various related topics have been widely studied in recent works, ranging from detailed placement optimization, standard cell layout optimization and new design rule-aware access model. (See Section 3.2.1 below for our definition.)

The works of [59][110] perform detailed placement optimization using a global routing solution as guidance, with pin accessibility modeled only in the form of pin density. Ding [22] develops a dynamic programming and linear programming-based detailed placement optimization considering pin access per instance pin. Ye [107] proposes an integer linear programming formulation to solve the unidirectional cell layout optimization under middle-of-line structure. However, the above models are over-simplified with assumptions of 1D gridded design and distance-based cost function, with no precise awareness of design rules. Recently, Xu [102][104] develops a series of pin access planning and regular routing techniques for self-aligned double patterning. These works, still under the assumption of 1D gridded design, are the first open literature trying to address both cell-level and instance-level pin accessibility. However, the methodology has a few drawbacks: (i) there is no robust flow to generate "hit points" given any 1D/2D, gridded/non-gridded design, with or without specific (e.g., self-aligned double patterning) design rules; (ii) the flow is unrealistic in that the number of "hit point combinations" is far too large, resulting in a complex lookup table that is impractical to use; and (iii) the benchmark suite is not public and includes testcases only up to 12K cells. These small testcases nevertheless consume as much as 800 seconds of wall time in multithreaded mode, which is a prohibitive runtime cost for real industry testcases and use contexts.

To our knowledge, no works present a complete, fully defined pin access analysis flow, or demonstrate robustness with a real detailed routing contest benchmark suite. In this work, we present a real, robust, scalable and design rule-aware dynamic programming-based pin access analysis framework that performs both standard cell-based and instance-based pin access analysis. With the integration to the open source TritonRoute [49][124], we demonstrate superior solution quality over the best known results [54] using the official ISPD-2018 benchmark suite [66]. Our main contributions are summarized as follows.

- We propose a multi-level, standard cell-based and instance-based pin access analysis framework with intra-cell and inter-cell pin accessibility awareness.
- We propose a robust and design rule-aware pin access point generation methodology for unique instances, supporting both planar and via access, and both on-track and off-track access.
- To achieve **intra-cell** pin compatibility, we propose a dynamic programming-based, design rule and boundary conflict-aware access pattern generation methodology for unique instances.
- We propose a dynamic programming-based access pattern selection methodology for standard cell instance clusters, which minimizes **inter-cell** pin access conflicts. To the best of our knowledge, this proposed framework is the only scalable solution in the open literature.
- We improve the pin access over the open-source TritonRoute v0.0.6.0 [125] (the latest release as of this writing), achieving design rule check (DRC)-clean via access for all of ISPD-2018 benchmark suite testcases. With the integration to TritonRoute, we demonstrate superior solution quality over the best known results using the official ISPD-2018 benchmark suite.

The remainder of this work is organized as follows. Section 3.2.1 provides background information for pin access. Section 3.2.2 describes our pin access methodology. Section 3.2.3 presents our experimental setup and results. Section 3.2.4 gives conclusions and directions for future work.

3.2.1 Preliminaries

In this section, we describe fundamental concepts that underlie pin access analysis: *unique instance*, *access point*, *access pattern*, and *coordinate types*.

Unique Instance

A **unique instance** is defined by a *signature*, which consists of (i) the cell master of the instance (e.g., NANDX1, NORX4, etc.); (ii) the orientation of the instance (e.g., R0, R180, MX, MY); and (iii) offsets to all track patterns that exist in the design DEF. Two instances having different signatures require separate intra-cell pin access analysis flows. Figures 3.10(a) and (b) illustrate two different unique instances.

Although the two instances share the same cell master and orientation, they are considered as different unique instances because they have different offsets to routing track patterns, resulting in different on-track, off-track conditions for the same pin access location (relative to the origin of the cell master). Thus, these instances require separate intra-cell pin access analyses. By contrast, two instances having the same signature would have exactly the same intra-cell pin access analysis result. Thus, we only need to perform intra-cell pin access analysis once for each unique instance.

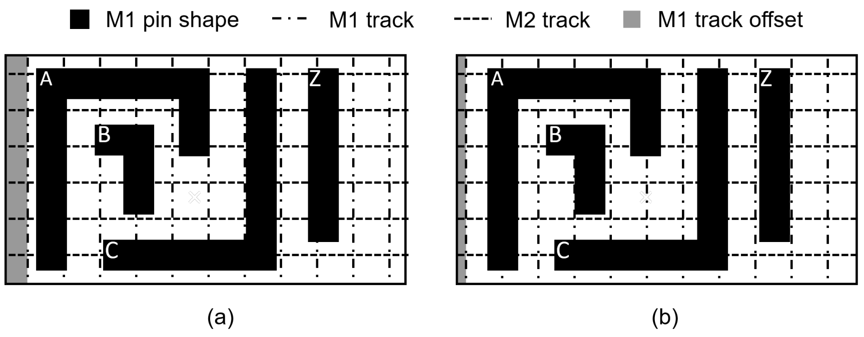


Figure 3.10: Illustration of two different unique instances that have the same cell master and orientation, but different offsets to track patterns.

Access Point

For each pin, an access point is an (x, y) coordinate on a metal layer where the detailed router ends routing. Each access point stores from which direction the router can access the pin. For example, in Figure 3.11, pin A has an access point indicating the up direction. We use a via12 enclosure to show that an up-via (i.e., a via connecting the pin to the upper metal layer) is valid to escape from this access point. Similarly, pin B (resp. C) has an access point indicating that routing to the east (resp. south) is valid. In our implementation, Each access point may indicate multiple valid access directions. For the up direction, we also store which vias are valid to use, among which one via is primary (preferred to use). The access point must be on the pin shape.

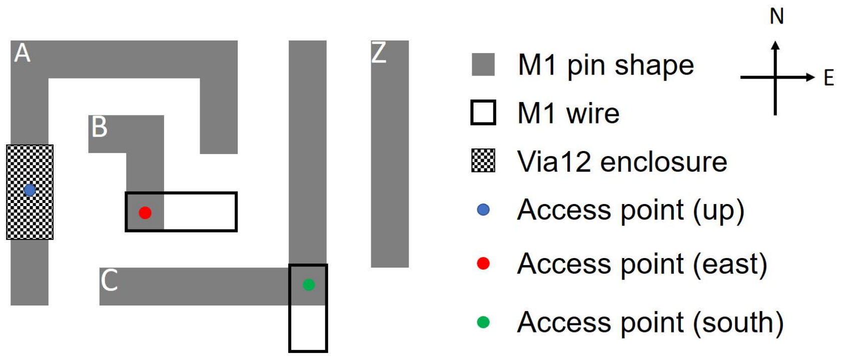


Figure 3.11: Illustration of access points.

Access Pattern

For each unique instance, an access pattern consists of one access point per pin, so that the primary vias from these access points are compatible (i.e., DRC-clean) with each other.

Coordinate Type

To accommodate a broad range of technology nodes, we define four coordinate types (and respective *cost* values, given in parentheses) as follows.

- An **on-track (0)** coordinate is on a preferred or non-preferred routing track. We always use the upper-layer preferred direction routing tracks as the non-preferred direction routing tracks for the current metal layer so that the on-track up-via access aligns to both the current and its immediately above metal layers.
- A **half-track (1)** coordinate is at the midpoint between two neighboring routing tracks.
- A **shape-center (2)** coordinate is at the midpoint between the left and right (or top and bottom) coordinates of a rectangular pin shape. If the pin consists of polygon(s), we generate the maximum rectangles of the polygon(s) (all overlapping rectangles that are maximal in area) to obtain shape-center coordinate(s). We skip the shape-center x (resp. y) coordinate if the x-span (resp. y-span) of the rectangle touches at least two tracks; we do this to reduce the occurrence of unique, off-track coordinates.

- An **enclosure boundary (3)** coordinate satisfies the via-in-pin requirement for an up-via access and the via enclosure alignment with the pin shape boundary.

Figure 3.12 illustrates examples of the coordinate types for a horizontal preferred direction. In Figures 3.12(a) and (b), we see that up-vias at the on-track and half-track coordinates cause minimum step DRCs. In such cases, we need shape center or enclosure boundary access points although they are off-track as illustrated in Figures 3.12(c) and (d). The above four types of coordinates are concise, while satisfying a broad range of technology nodes – from mature nodes where 2D, off-track pin access is required, to advanced nodes where 1D, on-track pin access is required. The cost serves as the priority (the lower, the better) when we loop through different types of coordinates to generate access points (cf. Lines 3 and 4 in Algorithm 17, in Section 3.2.2 below).

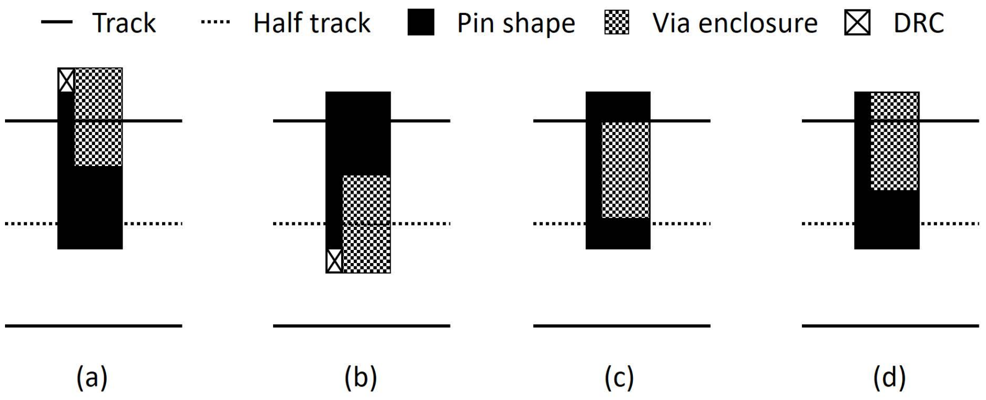


Figure 3.12: Illustration of four y-coordinate types, overlaid with same-layer up-via enclosure at the access point: (a) on-track; (b) half-track; (c) shape-center; and (d) enclosure boundary. Only (c) and (d) are DRC-clean.

3.2.2 Methodology

In this section, we describe our methodology to analyze pin accessibility for detailed routing. We perform three analyses in a multi-level sequence of three steps: (i) **pin-based access point generation**; (ii) **unique instance-based access pattern generation**; and (iii) **cluster-based access pattern selection**. The first step enumerates valid *access points* per unique instance, **without** consideration of intra-cell or inter-cell pin access compatibility. The second step picks good access points per pin within a given unique instance, forming an *access pattern*, within which **intra-cell** pin accesses are mutually compatible. The third step selects the best access pattern for all instances in the design, with awareness of **inter-cell** pin compatibility.

Step 1: Pin-Based Access Point Generation

Although we could enumerate all coordinate types to generate every access point per pin, in a reasonable detailed routing-driven pin access analysis framework the number of generated access points per pin should be neither too small nor too large. Too small a number of access points will overly restrict the solution space in detailed routing, resulting in degraded solution quality. On the other hand, given the heuristic, cost-based nature of modern detailed routing [54][124], too large a number of access points will provide excessive options (e.g., many off-track access points) for the detailed router, again resulting in degraded solution quality. Thus, the access point generation flow must be robustly designed to generate a proper amount of access points. In our flow, for example, to generate an access point at (x, y) on Metal1, where the preferred routing direction is horizontal, we consider all four coordinate types for the y coordinate (corresponding to the preferred direction), but only consider the first three coordinate types for the x coordinate (corresponding to the non-preferred direction) to reduce unique, off-track coordinates. We explain below the determination of "proper amount" after the description of Algorithm 17.

Algorithm 17 Pin-based access point generation

```
1: Inputs: pin, track patterns tps, viadefs vias
2: Output: valid access points aps
3: for all nonPreferredDirCoordType t1 $\in \{0, 1, 2\}$ do
4:    for all preferredDirCoordType t0 $\in \{0, 1, 2, 3\}$ do
5:       tmpAps $\leftarrow$ genAccessPoint(pin, tps, vias, t0, t1)
6:       for all ap $\in$ tmpAps do
7:          if isValid(ap) then
8:             aps += ap
9:          end if
10:      end for
11:      if $|aps| \geq k$ then
12:         return
13:      end if
14:   end for
15: end for
```

Algorithm 17 describes the pin-based access point generation. In Lines 3 – 4, we loop through different combinations of x and y coordinates sequentially according to their cost. For example, we first generate all (on-track, on-track) points, then (off-track, on-track) points, etc. In Line 5, for each type of coordinates, we first generate all access points. Then in Lines 6 – 10, we add all valid access points to the output. An access point is valid if a via can be dropped DRC-free to access the pin. We use an accurate DRC engine similar to the one used in [124] to perform the design rule check, considering all design rules existing in the specific design. Next, in Lines 11 – 13, we check whether we have generated enough access points for a pin, and early-terminate the procedure once the number of generated access points is equal to or greater than our required number k. Given the above, all access points of given coordinate types are generated, DRC-checked and added before we try to early-terminate the procedure. Therefore, the number of access points generated may be slightly larger than k. This behavior allows more access points to be generated when we are given a large pin shape, while also reducing the occurrence of unique, off-track coordinates. In our implementation, $k = 3$ for both standard-cell and macro-cell pins.

Step 2: Unique Instance-Based Access Pattern Generation

For each unique instance, we now describe how to pick a good access point per pin to form an *access pattern* in which the chosen access points are compatible with each other. Figure 3.13 illustrates our unique instance-based access pattern generation flow. The access pattern generation mainly consists of (i) pin ordering, (ii) graph construction, and (iii) dynamic programming-based pattern generation.

Pin ordering. Pin ordering is a preparation step for graph construction and dynamic programming-based pattern generation. Given a unique instance and an ordering of the pins in the unique instance, we assume only the neighboring ordered two-pin pairs might have conflicting access points (i.e., the two access points cause DRCs). For example, if we have a pin order of <A, B, C, Z>, then our assumption is that only <A, B>, <B, C> and <C, Z> could have conflicting access points, while <A, C>, <A, Z> and <B, Z> should not have conflicting access points. In this way, the access patterns can be generated within reasonable amount of time, without the need to perform design rule check among all two-pin pairs. For corner cases where non-neighboring two-pin pairs have conflicting access points, we can still avoid such cases by a post-processing method, described at the end of the discussion below of DP-based access

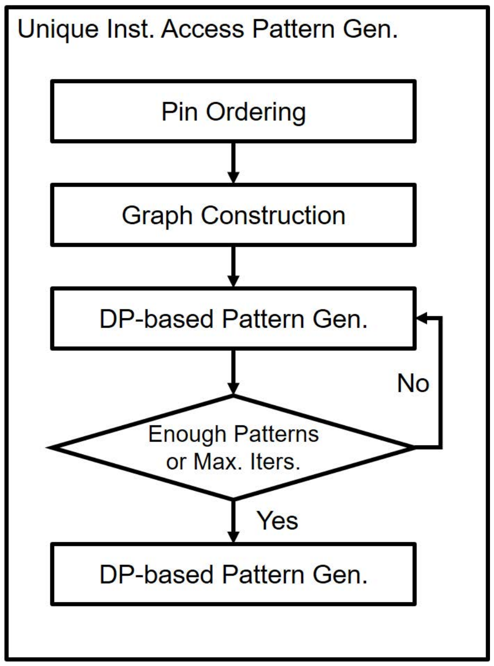


Figure 3.13: Iterative access pattern generation flow.

pattern generation. As shown in Section 3.2.3, this method works well in all ISPD-2018 benchmark suite testcases.

For a pin, if the averaged coordinates of all its access points are (x_{avg}, y_{avg}), then given a unique instance, we sort the pins according to $(x_{avg} + \alpha \cdot y_{avg})$. Figure 3.14 illustrates an example of unique instances with four pins. If $\alpha = 0$, then the pin ordering is equivalent to the ordering of x_{avg}. Thus, we obtain a pin order of <A, B, C, Z>. The first and last pin according to the pin order are **boundary pins**, which receive special treatment in access pattern generation as described below. Generally, given a reasonably small α ($\alpha < 1$), the first and last pins are the leftmost and the rightmost pins in the unique instance, respectively. In our implementation, we use $\alpha = 0.3$.

Graph construction. We build a graph for dynamic programming. Figure 3.15 shows the directed graph corresponding to the unique instance shown in Figure 3.14, assuming $\alpha = 0$. All edges are directed from left to right in the figure. The leftmost (resp. rightmost) vertex in the graph is the (virtual) starting (resp. ending) vertex, which serves as the starting (resp. ending) point in the dynamic programming that we describe below. Vertices between the starting and ending vertices represent access points; these are grouped by the owner pin of the access point, and ordered sequentially following the aforementioned pin

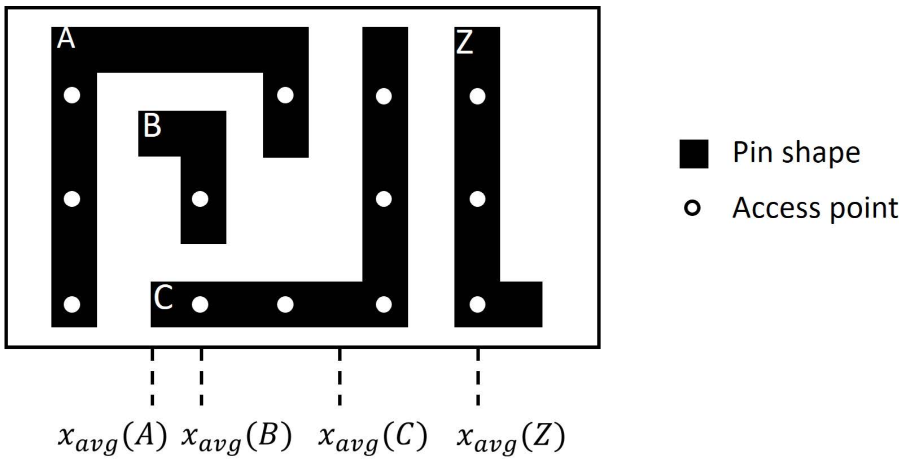


Figure 3.14: Pin ordering.

order. We build complete bipartite graphs over neighboring groups' respective vertex sets. A path from the starting vertex to the ending vertex visits one access point vertex per pin. The visited access points represent an access pattern.

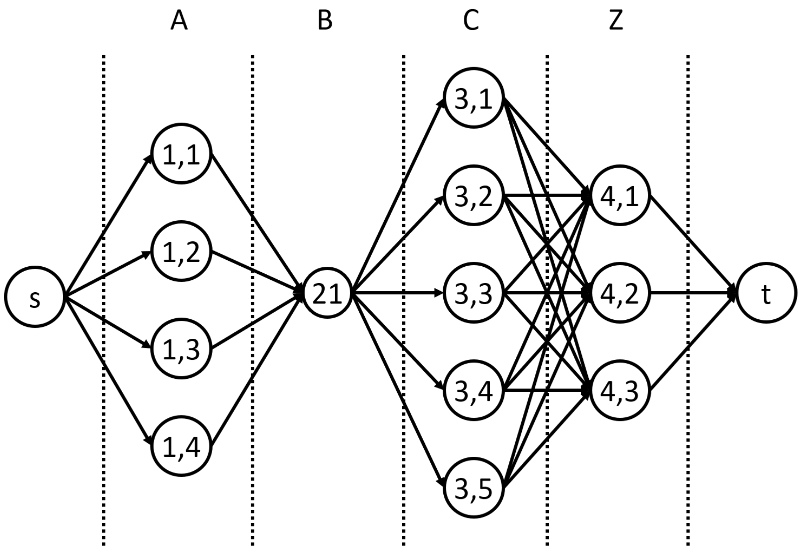


Figure 3.15: Graph for dynamic programming-based access pattern generation.

DP-based access pattern generation. Algorithm 18 describes our dynamic programming-based access pattern generation. The input is the graph. We describe all access points according to the pin index (m) and access point index (n). For example, access point {3,2} in Figure 3.15 is the second access point (n = 2) of the third pin (m = 3). Line 3 initializes the dynamic programming array dp. The array stores the minimum cost up to the current vertex, and its previous vertex. The minimum cost is initialized to infinity for every vertex except for the source. In Lines 4 – 17, we loop through all vertices (access points) of the current pin. For each vertex of the current pin, we find one vertex from the previous pin, from which the total path cost is minimized. Line 9 gets the edge cost from one previous access point vertex to the current access point vertex. Line 10 gets the total cost. The total path cost equals the previous path cost plus the edge cost. In Lines 11 – 14, we update the path cost up to the current vertex if the path cost is smaller than the existing path cost stored in the vertex. We also update the previous vertex, from which the path comes from, so that we can trace back the path to obtain the access pattern solution. Line 18 traces back the *dp* array and returns the access pattern with the lowest cost. We perform Algorithm 18 several times to generate up to three access patterns. Each time, the edge costs are slightly different so as to obtain different access patterns.

Algorithm 18 Access pattern generation

```
1: Inputs: graph $G(V, E)$
2: Output: access patterns $APs$
3: Initialize array $dp[m][n]$ $G(V, E)$
4: for all currPinIdx $m$ do
5:     for all currApIdx $n$ do
6:         for all prevApIdx $n'$ do
7:             $prev \leftarrow aps[m-1][n']$
8:             $curr \leftarrow aps[m][n]$
9:             $edgeCost \leftarrow$ getEdgeCost($prev$, $curr$)
10:            $pathCost \leftarrow prev.cost + edgeCost$
11:            if $pathCost < curr.cost$ then
12:                $curr.cost \leftarrow pathCost$
13:                $curr.prev \leftarrow prev$
14:            end if
15:        end for
16:    end for
17: end for
18: $APs \leftarrow$ traceBack()
19: return APs
```

Algorithm 19 details the edge cost calculation. The edge cost calculation is **boundary conflict-aware (BCA)**. In Lines 3 – 6, we assign a penalty cost to the boundary pin (the first and last pins according to the pin order) access points that have been selected in existing access patterns. This helps to generate access patterns with different boundary pin access points. Thus, two neighboring instances have more flexibility choosing compatible access patterns, as described in Section 3.2.2. Lines 7 – 8 check whether the two access points have design rule violations, and apply design rule violation cost if two access points are not compatible. Lines 9 – 10 further look back one more pin, and check whether the two access points (indexed $prev - 1$ and $curr$) have design rule violations. This step generates a history-based cost to avoid DRCs between non-neighboring access points. We call this step **history-aware optimization**. We note that since there can only be one intermediate solution when we reach node $curr$, the nodes $prev$ and $prev - 1$ are always deterministic, and thus the cost of each edge is still fixed. Line 12 calculates the edge cost according to the quality metric of the two access patterns if neither the penalty nor the violation cost applies.

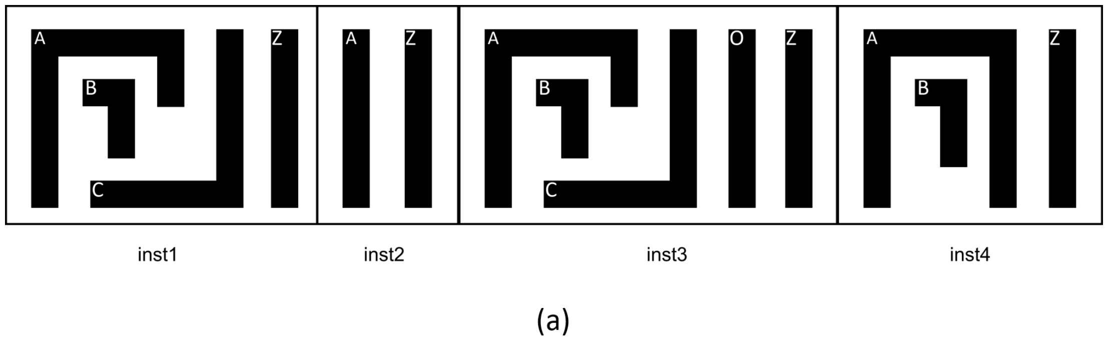
Algorithm 19 Edge cost calculation

```
1:  Inputs: previous dp array vertex prev current dp array vertex curr
2:  Output: edge cost cost
3:  if isUsed(prev) and prev $\in$ boundaryAp then
4:      edgeCost = penaltyCost
5:  else if isUsed(curr) and curr $\in$ boundaryAp then
6:      edgeCost = penaltyCost
7:  else if isDRCClean(prev, curr) then
8:      edgeCost = drcCost
9:  else if isDRCClean(prev-1, curr) then
10:     edgeCost = drcCost
11: else
12:     edgeCost = apCost(prev) + apCost(curr)
13: end if
14: return edgeCost
```

Finally, for all the access patterns that we generate, we use a DRC engine similar to the one used in [124] to validate whether there exist unseen DRCs, i.e., between non-neighboring groups of access points, or between multiple objects. To accelerate the access pattern generation, only up-vias are included for DRC.

Step 3: Cluster-Based Access Pattern Selection

Given access patterns per unique instance, we select the best access patterns per instance so that the access patterns of neighboring instances are compatible. Our cluster-based access pattern selection is performed on a continuous chunk of instances. We first group all instances according to their rows, and each continuous chunk of instances (no empty site in between) forms a cluster. We only consider the access pattern compatibility within a cluster while assuming that the neighboring clusters within or across rows always allow compatible access patterns. The cluster-based access pattern selection works similarly to the access pattern generation. The pin ordering step, in Algorithm 18, is now replaced with the instance ordering step, which naturally follows the left-to-right instance ordering. The graph construction works the same way except that now each vertex represents an access pattern of an instance. Finally, the dynamic programming-based optimization selects the best access pattern per instance to minimize the total cost. To accelerate the procedure, only up-vias of boundary access points (pin A and pin Z of each instance in Figure 3.16(a)) are included for DRC.



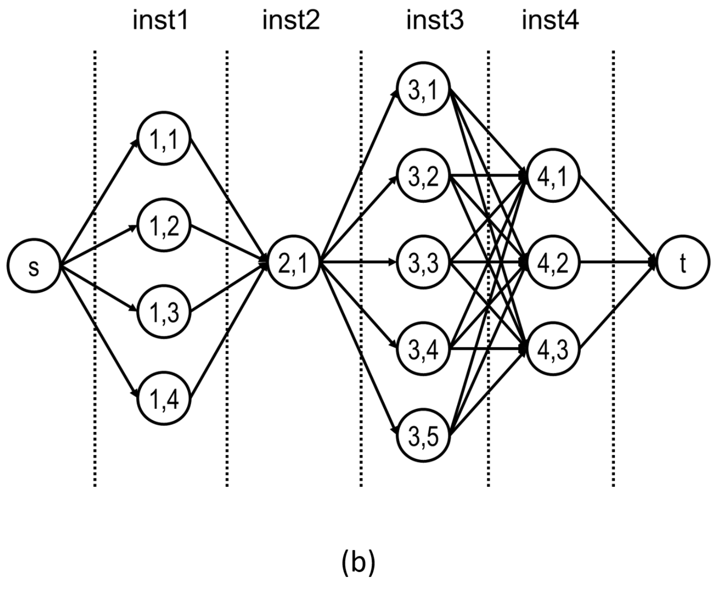


Figure 3.16: Illustration of (a) ordered cell instances and (b) corresponding graph.

3.2.3 Experiments

In this section, we present our experimental setup and results.

Experimental Setup

We implement our pin access analysis in C++ and integrate our framework with the open-source TritonRoute [124]. We perform all our experiments using the official ISPD-2018 initial detailed routing contest benchmark suite [66]. Table 3.5 summarizes the testcase information. These testcases are real industry designs with up to 290K standard cells in two technology nodes. We note that these testcases use real industry LEF-based design rule syntax, which is much more realistic than the testcases used in previous works [102][104]. Currently, no pin access framework targets the ISPD-2018 benchmark suite. To our best knowledge, no pin access framework has ever demonstrated enough robustness and scalability in publicly accessible, large benchmark testcases. Thus, we compare our work with the pin access framework from the latest release of the open-source TritonRoute v0.0.6.0 [125]. Furthermore, to enable a broader horizontal comparison to other frameworks, we also make necessary improvements to TritonRoute in addition to the integration of pin access analysis. We compare final routed designs to the best known academic detailed router – Dr. CU 2.0 [54]. All our experiments are performed using a Xeon 2.6GHz server in single-threaded mode. We perform three experiments.

- **Experiment 1**: We compare the quality of access points for all unique instance pins (without consideration of intra-cell or inter-cell pin access compatibility) from this work with that from TritonRoute v0.0.6.0.
- **Experiment 2**: We compare the quality of access points for all instance pins (with consideration of intra-cell and inter-cell pin compatibility) from this work with that from TritonRoute v0.0.6.0.
- **Experiment 3**: By integrating our framework with the open-source TritonRoute and making additional improvements, we enable a preliminary comparison of pin accesses from the final routed design, and also of the final routed #DRCs, between the original TritonRoute, the best known published result from Dr. CU 2.0 [54][115], and our pin access analysis framework. We further demonstrate the capability to extend our PAAF into 14nm and below nodes.

Table 3.5: Testcase information [66].

Benchmark	#Standard cell	#Macro cell	#Net	#IO pin	#Layer	Die size	Tech. node
ispd18_test1	8879	0	3153	0	9	0.20×0.19mm^2	45nm
ispd18_test2	35913	0	36834	1211	9	0.65×0.57mm^2	45nm
ispd18_test3	35973	4	36700	1211	9	0.99×0.70mm^2	45nm
ispd18_test4	72094	0	72401	1211	9	0.89×0.61mm^2	32nm
ispd18_test5	71954	0	72394	1211	9	0.93×0.92mm^2	32nm
ispd18_test6	107919	0	107701	1211	9	0.86×0.53mm^2	32nm
ispd18_test7	179865	16	179863	1211	9	1.36×1.33mm^2	32nm
ispd18_test8	191987	16	179863	1211	9	1.36×1.33mm^2	32nm
ispd18_test9	192911	0	178857	1211	9	0.91×0.78mm^2	32nm
ispd18_test10	290386	0	182000	1211	9	0.91×0.87mm^2	32nm

Experimental Results

Experiment 1. Table 3.6 shows the experimental results of the quality of access points for all unique instance pins, between the original TritonRoute (TrRte) and our pin access analysis framework (PAAF). This experiment only evaluates the quality of each access point, but does not consider intra-cell or inter-cell pin access compatibility. Total #APs means the total number of access points generated. #Dirty APs means #access points with DRCs. Ideally, a robust pin access point generation methodology should not generate any access points with DRCs. In *ispd18_test6*, with nearly 3K unique instances, our method generates 90K access points, all DRC-clean, within 80 seconds in single-threaded mode. Overall, our method generates only DRC-clean access points, while the original TritonRoute produces several hundreds of dirty access points. Also, our method generates more access points, while consuming less runtime.

Table 3.6: Comparison between the original TritonRoute (TrRte) and our pin access analysis framework (PAAF) for all unique instance pins (without considering intra-cell or inter-cell pin access compatibility) in terms of total #access points generated (Total #APs), #access points with DRCs (#Dirty APs), and runtime.

Benchmark	#Unique Inst	Total #APs		#Dirty APs		Runtime (s)	
		TrRte	PAAF	TrRte	PAAF	TrRte	PAAF
ispd18_test1	182	2320	**3102**	0	**0**	4	2
ispd18_test2	222	3638	**4867**	1	**0**	8	4
ispd18_test3	227	3672	**4970**	1	**0**	8	4
ispd18_test4	2725	98220	**99356**	416	**0**	120	63
ispd18_test5	2733	76290	**80027**	385	**0**	142	71
ispd18_test6	2886	84012	**87876**	469	**0**	163	78
ispd18_test7	148	3982	**4152**	4	**0**	7	3
ispd18_test8	414	11814	**12316**	10	**0**	20	12
ispd18_test9	404	11832	**12342**	12	**0**	21	11
ispd18_test10	426	11749	**12254**	12	**0**	20	13

Experiment 2. Table 3.7 shows the experimental results of the quality of access points for all instance pins, between the original TritonRoute (TrRte) and our pin access analysis framework (PAAF). We have two setups for PAAF. The first setup is "without BCA" (w/o BCA): we generate only one access pattern per unique instance, hence the access pattern is not boundary conflict-aware and there could be inter-cell pin accessibility issues. The second setup is "with BCA" (w/ BCA): we generate up to three access patterns per unique instance. Total #pins means the total number of all instance pins (with net attached). Since all of these pins must be connected in detailed routing, we need a good (i.e., DRC-clean) access point per pin. #Failed pins means the number of pins without a DRC-clean access point. We can see that the original TritonRoute fails to provide legal pin access for thousands of instance pins, while our PAAF can generate intra-cell and inter-cell DRC-clean pin access. For up to 790K instance pins, PAAF takes less than a minute of runtime in single-threaded mode. Note that runtime is one of the most important aspects of a pin access analysis framework in physical design, especially for support of placement optimizations (i.e., detailed placement, sizing, buffering), where frequent changes in placement require a tremendous amount of inter-cell pin access analysis.

Table 3.7: Comparison between the original TritonRoute (TrRte) and our pin access analysis framework (PAAF) for all instance pins (considering intra-cell and inter-cell pin access compatibility) in terms of #pins without a DRC-clean access point (#Failed Pins), and runtime. Total #pins means the total number of all instance pins (with net attached).

Benchmark	**Total #Pins**	**#Failed Pins**			**Runtime** (s)		
		TrRte	PAAF		TrRte	PAAF	
			w/o BCA	w/ BCA		w/o BCA	w/ BCA
ispd18_test1	17203	31	0	**0**	4	3	5
ispd18_test2	157990	665	0	**0**	7	5	8
ispd18_test3	158110	663	0	**0**	7	5	7
ispd18_test4	316652	1305	0	**0**	95	84	94
ispd18_test5	316220	2529	80	**0**	107	85	98
ispd18_test6	474300	4048	0	**0**	113	96	121
ispd18_test7	790550	7816	0	**0**	8	7	23
ispd18_test8	790550	7816	0	**0**	20	17	39
ispd18_test9	790550	7816	0	**0**	20	17	38
ispd18_test10	790550	7816	0	**0**	21	18	49

Experiment 3. By integrating our framework with the open-source TritonRoute v0.0.6.0 [125] (the latest release as of this writing) and making additional improvements, we show a preliminary result of

pin accesses from the final routed design, and also of the #DRCs for the final routed design, for testcase *ispd18_test5*. Figure 3.17 compares two pin accesses from the final routed design, between Dr. CU 2.0 and our PAAF. As noted above, PAAF is capable of generating DRC-clean pin access for all instance pins. By using our robust PAAF, we surpass the best known academic detailed routing result in terms of #DRCs. The current best known result comes from Dr. CU 2.0 [54][115], with 755 DRCs. By contrast, we complete detailed routing with only two DRCs, and with no pin access issues remaining.

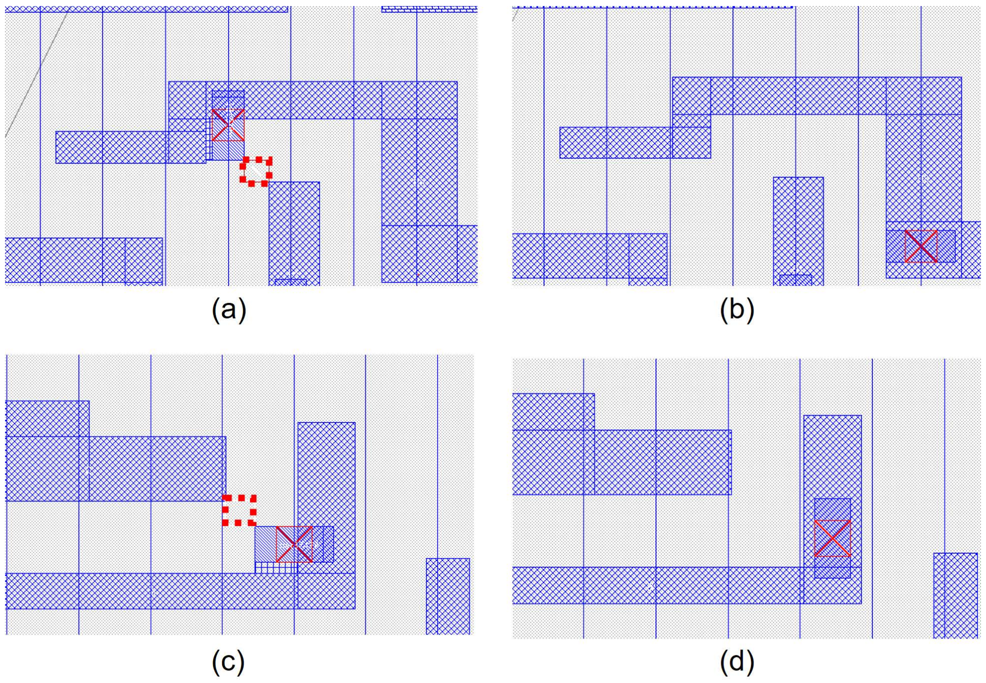


Figure 3.17: Comparison of pin access between Dr. CU 2.0 and PAAF: (a) Dr. CU 2.0 (Case 1), (b) PAAF (Case 1), (c) Dr. CU 2.0 (Case 2), and (d) PAAF (Case 2). Dashed red boxes are DRCs. Testcase: *ispd18_test5*.

We also perform a preliminary study on pin accessibility using a commercial $14nm$ library. We perform our experiments using the AES testcase from OpenCores [121] (20K instances, 779 unique instances). Our preliminary study shows that our PAAF successfully generates and selects DRC-clean access points for all 57K instance pins in a runtime of 9 seconds. An example of standard cell pin access is shown in Figure 3.18.

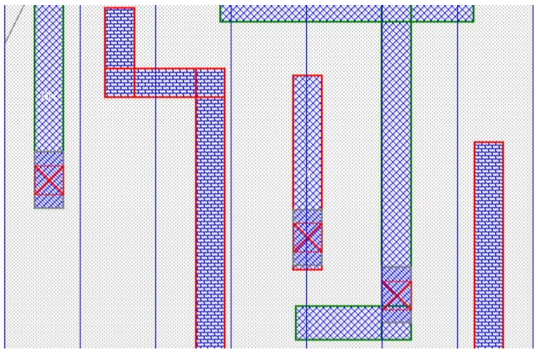


Figure 3.18: Illustration of pin accesses in $14nm$. Note that off-track pin access is enabled automatically in PAAF.

3.2.4 Conclusion

In this work, we present a multi-level, standard cell- and instance-based, complete, robust, scalable and design rule-aware pin access analysis framework. We describe our robust pin-based access point generation, boundary conflict-aware access pattern generation and cluster-based access pattern selection based on dynamic programming. We achieve 100% DRC-clean pin access and demonstrate a superior final detailed routing solution as compared to the best known results using the ISPD-2018 initial detailed routing contest benchmark suite. Our future research directions include: (i) inter-cluster and inter-row pin access co-optimization for better access point alignment; and (ii) incremental pin access analysis for application in engineering change order (ECO) contexts.

3.3 Acknowledgments

Chapter 3 contains reprints of Andrew B. Kahng, Lutong Wang and Bangqi Xu, “The Tao of PAO: Anatomy of a Pin Access Oracle for Detailed Routing”, *Proc. ACM/IEEE Design Automation Conference*, 2020. Chapter 3 also contains part of the draft of Andrew B. Kahng, Lutong Wang and Bangqi Xu, “TritonRoute-WXL: The Open Source Router with Integrated DRC Engine”, in submission to *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021. The dissertation author is a main contributor to, and a primary author of, each of these papers.

I would like to thank my coauthors Andrew B. Kahng and Lutong Wang for their support and work.

Chapter 4

Open Source Routing Framework for Advanced Technology Nodes

This chapter presents TritonRoute-WXL – a complete, end-to-end routing flow for advanced technology nodes. TritonRoute-WXL is the academia-leading router that consists of an in-memory router database, a global routing engine, a track assignment engine, and a detailed routing engine with integrated pin access analysis engine and design rule check engine. The global and detailed routing engines adopt a region-based ripup-and-reroute methodology. Our router comprehends design rule constraints in industry-standard formats for advanced technology nodes. TritonRoute-WXL delivers unparalleled solution quality for academic contest benchmarks in various technology nodes. For foundry technology nodes, TritonRoute-WXL is capable of delivering DRC-clean routing solutions for sub-14nm technology nodes.

4.1 TritonRoute-WXL: The Open-Source Router with Integrated DRC Engine

Routing is a crucial stage in a modern design automation tool flow for advanced technology nodes. A new advanced technology node enablement comes with increasingly more complex design rules. Such complex design rules introduce ever-more challenges to routing, especially detailed routing. The key element for detailed routing to comprehend such complex design rules is a design rule check engine (DRC engine). Although design rule checking has been studied for more than thirty years, to the best of our knowledge, a comprehensive documentation of implementation in the context of advanced-node detailed routing is still missing. Moreover, for detailed routing in advanced technology nodes, incremental capability of a DRC engine is highly desired due to the nature of per-net ripup-and-reroute in detailed routing.

More complex design rules along with the decreasing feature sizes in new technology nodes make standard cell design more challenging as well. For old technology nodes, intra-cell connections are routed mostly at or below the first metal layer (M1). For most of the standard cell pins, they are preferred to be accessed by vias. However, for complex logical cells (e.g., Flip-Flops) in advanced technology nodes, standard cell designers increase the usage of M2. Some of such M2 usage form standard cell pins which intend to be accessed by planar wires. As a result, such a mix of via accesses and planar accesses for standard cell pins introduce extra challenges in global routing and detailed routing correlation in terms of routing resource modeling.

The VLSI routing problem is commonly divided into two separate stages–global routing stage and detailed routing stage. Although both global routing and detailed routing problems have been extensively studied for decades, the connection and / or correlation between global routing and detailed routing is still an open question in the open literature. The two-stage (i.e., global routing and detailed routing) approach greatly simplifies the routing problem based on the assumption that the global routing has a near perfect routing resource model that correlates with detailed routing. Therefore, it is essential to have an accurate routing resource model that well reflects various aspects of routing resource in detailed routing including routing tracks, pin access, design rules, etc.

Another benefit of dividing the routing problem into two separate subproblems is that it enables academic researchers to focus on a specific subproblem. Various academic contests have strongly spurred academic research activities. The ISPD-2007 [71] and ISPD-2008 [70] global routing contests, along with the recent ICCAD-2019 global routing contest [24], have stimulated research efforts on global routing. The ISPD-2018 [66] and ISPD-2019 [61] initial detailed routing contests have stimulated academic efforts on detailed routing.

A drawback of separating global routing and detailed routing research is that almost no academic works attempt to present an *end-to-end* routing flow. Hence, application of academic routing works to real-world IC physical design (P&R) is extremely difficult. Moreover, direct application of academic routing works to industrial benchmarks in sub-65nm nodes can commonly leave unacceptable amounts of design rule violations (DRCs). Even for academic contest benchmarks, existing known best routing solutions from academic routers can still have hundreds, if not thousands, of DRCs, which is far from "DRC converged" from an industry perspective. We further note that most contest-based academic global routing works model routing resources based on adjacent global routing cell (**GCell**) edges rather than the GCells themselves. Such routing resource modeling approach is straightforward for a contest. However, a GCell edge-based resource model makes considering the impact of local nets and pin accessibility difficult, since it only captures inter-GCell routing resource.

Given the above, a capable, end-to-end routing flow is very meaningful for the field to (i) bridge the gap between academic research efforts and industrial technology needs, and (ii) enable further academic research works (e.g., placement) that can be directly evaluated with a usable routing flow instead of academic contest-centric evaluation metrics. Towards this end, we present TritonRoute-WXL, an open source router for advanced VLSI technologies with integrated DRC engine, in this work. Our main contribution is an end-to-end routing framework that aims to narrow the gap between academia and industry. Highlights of our work are summarized as follows.

- We propose an end-to-end routing framework. Our proposed framework is capable of well correlating global routing and detailed routing to achieve faster routing convergence.
- We build an integrated design rule check engine that provides design rule check capability and enables further routing optimization with its incremental capability.
- We present a global routing resource model that comprehends various detailed routing aspects to achieve better DR convergence.
- We present an improved detailed routing methodology that is capable of achieving faster DR convergence as compared to existing detailed routing methodologies.

- Our router is capable of delivering DRC clean routing solutions for 15 out of the 20 ISPD-2018 and ISPD-2019 benchmark suite testcases. For the remaining, testcases, we still reach an unparalleled level of DRCs ($<$20).
- To the best of our knowledge, we provide the first and the only free and open source software (FOSS) router which is capable of delivering DRC-clean routing solution in sub-16nm technology nodes.

The remainder of this work is organized as follows. Section 4.1.1 provides a brief overview of previous works in the open literature. Section 4.1.2 presents our overall routing flow. Section 4.1.3 details our global routing methodology. Section 4.1.4 presents our detailed routing database. Section 4.1.5 details our overall detailed routing flow. Section 4.1.6 presents our detailed routing methodology. Section 4.1.7 presents our experimental results using the official ISPD-2018 benchmark suite. Section 4.1.8 gives conclusions.

4.1.1 Related Work

As surveyed in [8], previous works on detailed routing can be categorized into fundamental and conventional algorithms, and recent developments. Further, we summarize the recent works targeting the ISPD-2018 initial detailed routing contest.

Fundamental and conventional algorithms. Lee [52] proposed the first maze routing algorithm, i.e., a breadth-first search that guarantees to find a minimum-cost path between two terminals if a path exists. Use of “best-first search”, also known as A* search [73], sometimes in its bidirectional [79] form, enables maze-based search to focus itself toward desired targets, and reduces effort needed to find a minimum-cost feasible path. Hadlock [35] and Soukup [88] applied speedups to Lee’s algorithm and others applied the line-search paradigm [41] to improve time and space efficiency as compared to Lee’s and A* algorithms. Hetzel [40] developed a sequential routing approach using a shortest path algorithm with respect to euclidean distance. Specialized contexts such as channel routing [28] and switchbox routing [63], along with general frameworks such as multicommodity flow [87] and ripup-and-reroute [93], have respective sub-literatures and remain as fundamental building blocks of the detailed router today (cf. [31]).

Global routing. Many works have been developed based on the ISPD-2007 [71] and ISPD-2008 [70] global routing contests. NCTU-GR 2.0 [60], NTHU-Route 2.0 [6], NTUgr [9] and FastRoute 4.0 [105] adopt similar flow of (i) projecting 3D routing problems into 2D routing problems, (ii) routing decomposed multi-pin nets and (iii) performing layer assignment to obtain 3D GR solutions. FGR [83] performs global routing on a 3D graph based on Discrete Lagrange Multiplier. GRIP [98] applies integer programming to solve the global routing problem. MGR [101] adopts a multi-level approach to more efficiently explore the large routing solution space. The recent ICCAD-2019 global routing contest [24] evaluates a global routing solution by assessing the corresponding detailed routing solution, to accurately capture routability from a detailed routing perspective. CUGR [62], the contest's winning global router, performs a detailed routability-driven 3D global routing based on a probabilistic resource model.

Many works explore machine learning techniques, based on global routing information, to predict the outcomes of subsequent detailed routing stage. Qi et al. [81] and Zhou et al. [113] build multivariate regression models. Chan et al. [5] adopts support vector machine for DRC distribution prediction. Xie et al. [99] train a fully convolutional network for such prediction. Recently, Chen et al. [10] propose a fully convolutional network-based plug-in to optimize global routing solutions, thus reducing post-route DRCs.

Detailed routing. More recent academic works on detailed routing focus on certain aspects of the modern routing challenge, mainly to address issues arising with advanced nodes. [53] gives an excellent summary of the academia-industry gap for detailed routing as of 2003; much of this gap remains today. Examples of focused recent works include Nieberg [72], which proposes techniques for gridless pin access in detailed routing. Xu [104] proposes pin-access planning and regular routing for self-aligned double patterning (SADP). The works of [21][25][29][58] address the detailed routing problem in an SADP process context. MANA [7] introduces an end-end separation and minimum wire length-aware shortest path algorithm. Han [37] develops a framework to reduce various DRCs in advanced nodes using multicommodity flow-based integer-linear programming. BonnRoute [1][31] and RegularRoute [111] are two works prominent in the recent literature that present more complete portraits of overall detailed routing solutions.

Recently, a few works in the open literature attempt to address the gap between modern industrial designs and academic detailed routing flows, based on the ISPD-2018 and ISPD-2019 initial detailed routing contest [61][66]. Sun [89] presents a multi-stage ripup-and-reroute flow for detailed routing. Kahng [49] proposes an integer linear programming (ILP)-based parallel intra-layer and sequential inter-layer routing flow. Chen [12][13] and Li [54] propose a detailed routing flow using min-area-captured path search on a sparse grid graph. Gonçalves et al. [32][33][34] propose a tunnel-aware A* lower bound, and a design-rule-aware path search algorithm for detailed routing. Although most recent works use correct-by-construction or safe-by-construction approaches to prevent DRCs, none of them is capable of delivering decent solution quality (that is, in a practical sense) due to the complexity of developing the necessary router infrastructure.

4.1.2 Overall Routing Flow

In this section, we describe our global-detailed routing flow. As shown in Figure 4.1, our router takes industry-standard LEF and DEF files as inputs. Based on the input LEF and DEF files, we first set up the design database. Next, we perform preparation steps to generate essential data for routing. Then, we perform pin access analysis. Both global routing and detailed routing are based on the same pin access information. We next perform global routing followed by track assignment. Finally, we perform detailed routing to obtain a routed DEF. Routing is performed in non-overlapping regions in parallel.

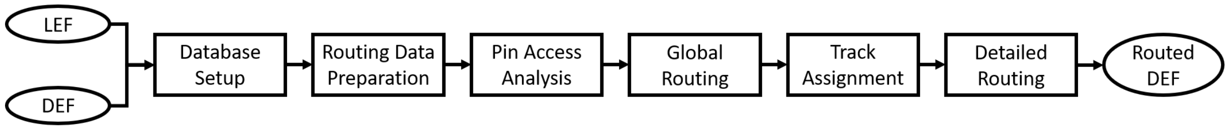


Figure 4.1: Overall global-detailed routing flow.

4.1.3 Global Routing

In this section, we describe our global routing flow that operates on the global routing cells (GCells) level. As shown in Figure 4.2, we first set up the congestion map using our GCell-based routing resource modeling. Next, we perform initial global routing which consists of (i) Steiner tree construction and (ii) iterative pattern routing. Then, we perform 2D ripup-and-reroute to resolve 2D congestions. After that, we perform layer assignment to obtain an initial 3D global routing solution. Finally, we perform 3D ripup-and-reroute to refine the 3D global routing solution to resolve 3D congestions.

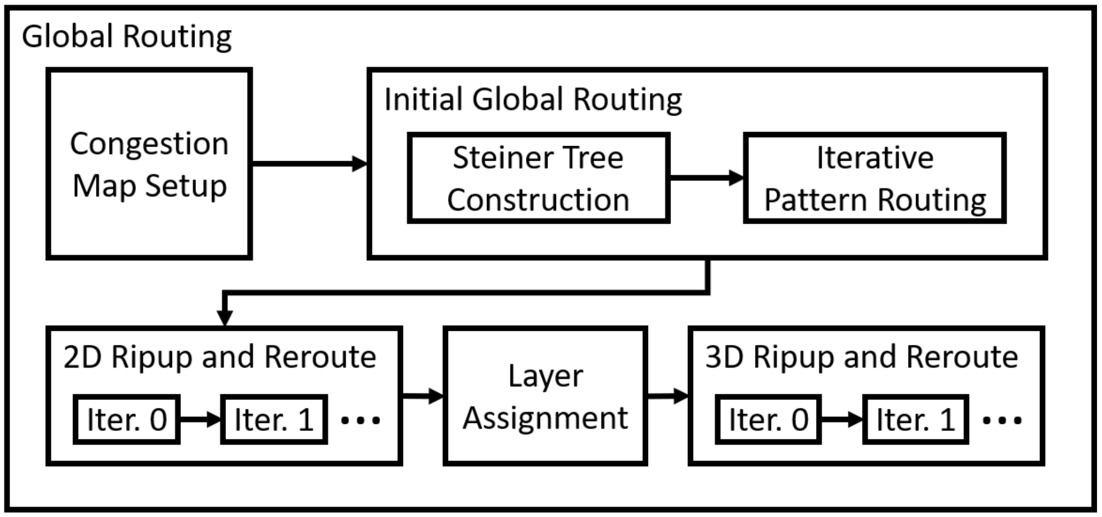


Figure 4.2: Global routing flow.

Routing resource modeling

We now describe our routing resource modeling that we use to analyze the routing resource of the design and set up the initial congestion map considering (i) routing tracks, (ii) design rules and (iii) pin accesses. In the global routing context, routing resource is usually abstracted with two concepts–**supply** and **demand**. As pointed out in [3], the edge capacity model that is widely used in the ISPD global routing contest-based works ignores the impact of local nets. In this work, we associate both supply and demand to the GCell itself, so as to enable a unified resource model that considers both global nets and local nets in terms of routing wire and pin access. We use a GCell size of 15×15 M1 track pitch in this work.

Supply. Conventional method of obtaining the supply of a GCell is to simply count the number of routing tracks within the GCell. In most cases, the supply can be calculated by dividing the size of the GCell by *track-to-track* pitch. However, such calculation can be optimistic due to its unawareness of design rules. For a given routing layer, the via to the upper routing layer can have a wide enclosure. Dropping such a via can block its neighboring routing tracks as shown in Figure 4.3. Therefore, for routing layers (e.g., Metal6 in Figure 4.3) that use such via with wide enclosure, the supply needs to be adjusted based on the *track-to-via* pitch which is the minimum spacing required between the enclosure and a neighboring routing wire.

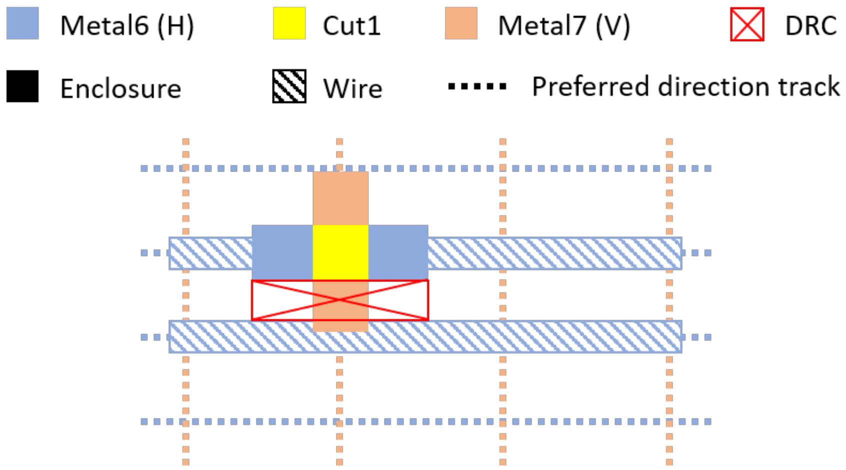


Figure 4.3: Illustration of optimism in supply calculation for routing layer with wide via enclosure.

Demand. Demand of a GCell consists of a *static* part and a *dynamic* part. The static part has two sources–fixed objects (i.e., pin shapes and obstacles) and pin accesses. For each fixed object, it creates one demand for each routing track which the fixed object overlaps or is too close to, because essentially the track is blocked by the fixed object. For each pin, its pin access creates an additional half unit of demand because if (a) the pin is connected to another pin within the same GCell, we consider that the two pins together consume one track for their local connection; or if (b) the pin is connected to the outside of the GCell, we consider that there is a virtual boundary pin at the GCell boundary whereby the pin takes an outgoing route from the GCell. Hence the pin and the virtual pin together create one demand. Note that the pessimism in routing layers with pin accesses allows more flexibility for detailed routing to resolve pin access-related violations.

To better correlate global routing and detailed routing in terms of the routing resource consumed by pin access, we introduce a variable *viaAccessLayer*. If the access point is at the *viaAccessLayer*, the pin access creates demand on its upper layer as it indicates a via access; otherwise, the pin access creates demand on the current layer. We refer readers to [50] for more details of our pin access methodology.

The dynamic part of a GCell demand is purely contributed by routing wires that intersect with the given GCell. Following the idea of virtual boundary pin at GCell boundary, each time a routing wire intersects with a GCell, it creates a boundary pin. Therefore, a routing wire that routes through a GCell creates two boundary pins, and in total creates one demand (i.e., routing resource of one track).

Figure 4.4 illustrates our unified resource model. Figure 4.4(a) shows the layout within a GCell that consists of two routing wires and a cell pin. Figure 4.4(b) illustrates the corresponding resource model. For M1, a total of three units of demand consist of two static units from pin shapes and one dynamic unit from boundary pins. For M2, the via access to the M1 pin contributes half a unit of dynamic demand and the routing wire contributes one unit of dynamic demand.

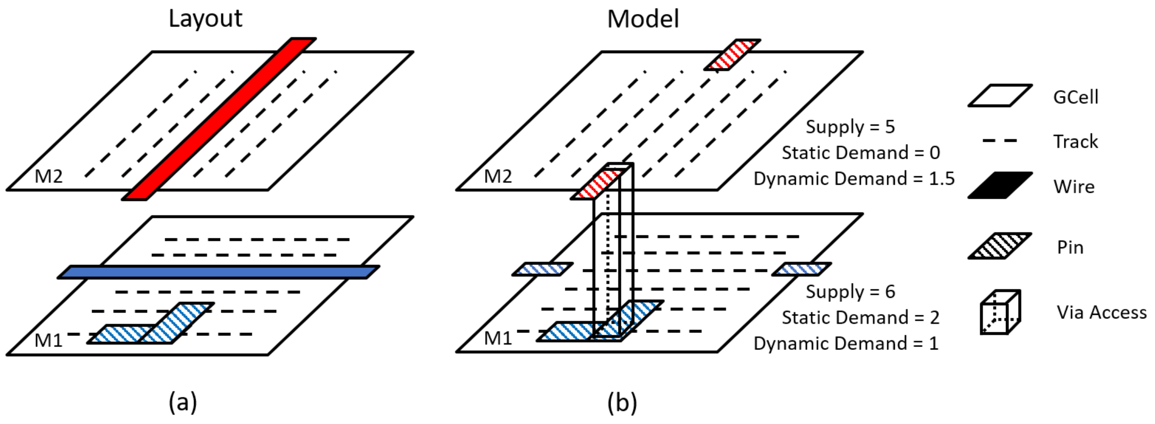


Figure 4.4: Illustration of unified routing resource model: (a) the layout of a pin and a routing wire inside a GCell; and (b) the corresponding modeled routing resource with boundary pins and via access.

Blocked GCell. For the GCells that have greater static demands as compared to their supplies, we consider that such GCells are blocked. Blocked GCells are associated with a very large cost so that they should be avoided if possible.

We set up the initial congestion map in 3D view and obtain a corresponding 2D congestion map based on the 3D congestion map. For each GCell, we project the supplies and demands from all routing layer to 2D plane. A GCell is considered as blocked if all of its corresponding GCells in 3D congestion map are blocked.

Initial global routing

The initial global routing consists of (i) Steiner tree construction and (ii) iterative pattern routing. We use FLUTE [15] to obtain a Steiner tree topology with low wirelength for each net. For the non-colinear edges from FLUTE, we perform iterations of L-shape pattern routing to minimize congestion.

2D ripup-and-reroute

Considering the limited solution space from L-shape pattern routing, it is likely to have overflow after initial global routing. To deliver a solvable problem for layer assignment, we perform three *outer* iterations of 2D ripup-and-reroute to resolve overflow in the 2D view.

Region-based ripup-and-reroute. In each *outer* iteration, we partition the design into non-overlapping, 200$\times$200-GCell-sized clips. For each clip, we create a GR worker that performs two *inner* iterations of ripup-and-reroute to resolve overflow. We shift the clips in different iterations with offsets of 0, -70 and -150 GCells to enable optimization at clip boundaries.

Algorithm 20 details our flow within a GR worker. Each GR worker takes a congestion threshold variable *congThres* as input. A congestion threshold variable *congThres* of 0.8 indicates that any GCell whose demand exceeds 80% of its supply is considered as having overflow. Line 2 initializes the worker database from the global database, including the netlist within the worker and a local congestion map. Lines 3–11 perform $maxIter$ iterations of ripup-and-reroute. Within each iteration, Lines 4–10 iterate over all nets within the worker. If a given net routes through any GCell that has overflow, Line 6 increments history cost counter for all of the overflowed GCells that the given net routes through. Line 7 rips up the given net and updates the local congestion map accordingly. Line 8 reroutes the given net. Note that during reroute, the path search algorithm has the freedom to alter the topology of the given net in order to mitigate congestion. Line 12 writes back to the global database. We gradually decrease *congThres* from 1.0 to 0.8.

Each global routing worker takes a congestion threshold variable *congThres* as input. A congestion threshold variable $congThres = 0.8$ indicates that any GCell whose demand exceeds 80% of its supply is considered as having overflow. Line 2 initializes the worker database from global database, including the netlist within the worker and a local congestion map.

Algorithm 20 Global routing flow

```
 1: Input: congestion threshold $congThres$
 2: WorkerDBInit()
 3: while $currIter < maxIter$ do
 4:    for all $net \in nets$ do
 5:       if hasCongestion($net$, $congThres$) then
 6:          addHistoryCost($net$)
 7:          ripupNet($net$)
 8:          routeNet($net$)
 9:       end if
10:    end for
11: end while
12: DBCommit()
```

Lines 3–11 perform $maxIter$ iterations of ripup-and-reroute. Within each iteration, Lines 4–10 iterate over all nets within the worker. If a given net routes through any GCell that has overflow, Line 6 increments history cost counter for all of the overflowed GCells that the given net routes through. Line 7 rips up the given net and updates the local congestion map accordingly. Line 8 reroutes the given net. Line 12 writes back to the global database.

Routing cost. We use five types of costs: **wirelength cost**, **congestion cost**, **history cost**, **blockage cost** and **overflow cost**. Different cost components have their own use cases. Wirelength cost helps A* to minimize the overall wirelength when there is no congestion. Congestion cost helps A* to avoid congestion. History cost helps A* to avoid regions that have or had overflow. Blockage cost prevents A* from reaching a blocked GCell. Overflow cost helps differentiate among GCells that have demands close to their supplies. The overall cost of routing from GCell *i* to GCell *i+1* is the wirelength between GCell *i* to GCell *i+1*, weighted by cost function in Equation 4.1. The overall relation among wirelength cost, congestion cost and history cost is inspired by [67].

$$cost_{tot}(i) = 1 + w_1 \cdot cost_{cong} + w_2 \cdot cost_{hist} + \\ w_3 \cdot cost_{block} + w_4 \cdot cost_{overflow} \tag{4.1}$$

$$cost_{cong}(i) = \frac{demand(i)/(supply(i)+1)}{(1+e^{supply(i)-demand(i)})} \tag{4.2}$$

$$cost_{hist}(i) = histCnt(i) \cdot cost_{cong}(i) \tag{4.3}$$

$$cost_{overflow}(i) = \begin{cases} 1, & \text{if } demand(i) \geq supply(i) \\ 0, & \text{otherwise} \end{cases} \tag{4.4}$$

We adopt similar congestion cost function from [62]. The idea behind the congestion cost function is to allow very small cost when the demand is low and to noticeably increase the congestion cost as the demand approaches the supply. Variations of the congestion cost function with similar idea have been discussed in [6], [60] and [83]. We adopt similar history cost from [67] and we use a history cost counter *histCnt* and increment the counter each time an overflow is encountered. The history cost counter *hisCnt* is decayed (i.e., multiplied by a fractional value) after each iteration. The weights of each cost components are chosen to achieve the following order for a blocked and overflowed GCell: $w_1 \cdot cost_{cong} < w_2 \cdot cost_{hist} < w_4 \cdot cost_{overflow} \ll w_3 \cdot cost_{block}$.

Layer assignment and 3D ripup-and-reroute

We adopt a simplified version of dynamic programming based layer assignment from [16]. We sort nets that need layer assignment using the score function from Equation 4.5 as a “flexibility” measurement which is similar to the one in [105].

$$Score(net) = \frac{HPWL(net)}{|pins(net)|} \tag{4.5}$$

Note that although an overflow-free 3D routing solution can be constructed based on an overflow-free 2D routing solution using layer assignment [83], layer assignment solution refinement is usually still desired to further improve the solution quality. Unlike the iterative reassignment approach in previous works (e.g., [16]), we perform 3D ripup-and-reroute in smaller regions. The benefit of region-based 3D ripup-and-reroute is that optimizations can be performed in parallel for improved scalability. For 3D ripup-and-reroute, we partition the design into 10×10-GCell-sized clips for local optimization.

4.1.4 Database

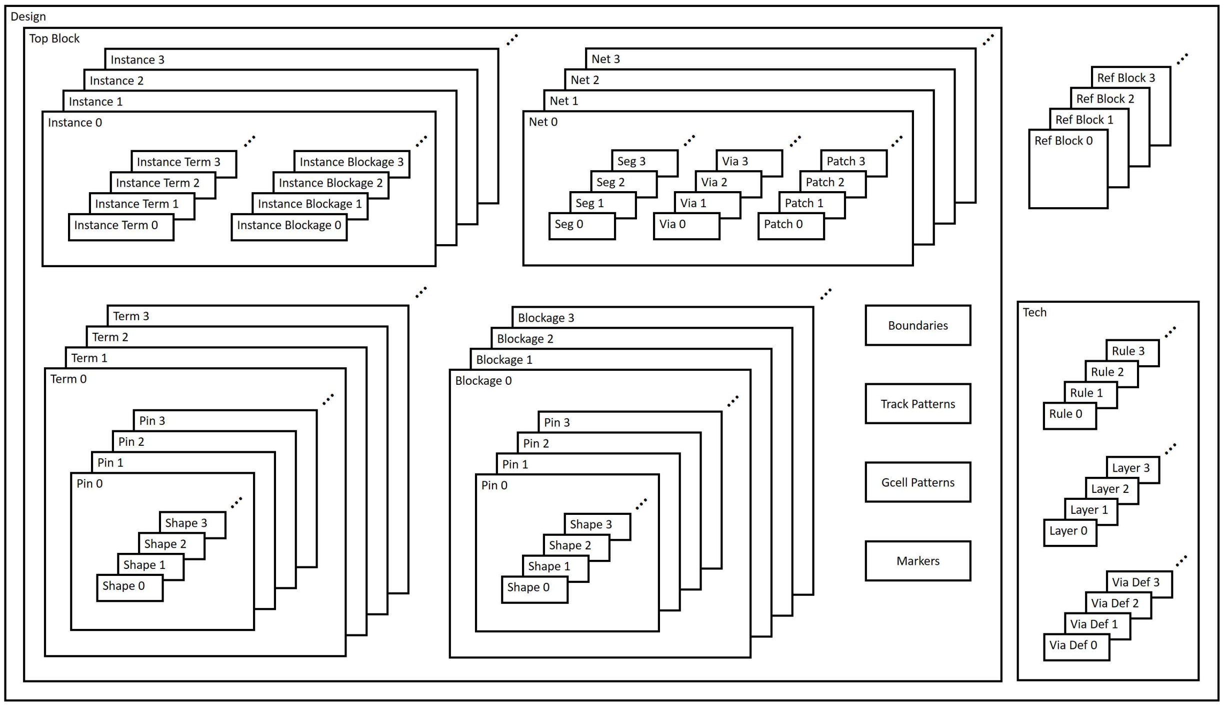


Figure 4.5: Major database structures.

In this section, we list all major objects and structures in the routing database. In building this database, we follow the LEF/DEF [117] data model, and reuse the naming convention from OpenAccess [126] as much as possible. The objects from LEF are summarized in Table 4.1, and the objects from

Table 4.1: Database objects from LEF.

Object	**LEF Keyword**	**Meaning**
tech		back-end-of-line metal stacks
layer	LAYER	metal or cut layers
viadef	VIA	via definitions
constraint	WIDTH	default routing width
	AREA	minimum area rule
	SPACING	spacing rule
	SPACINGTABLE	spacing table rule
	MINIMUMCUT	minimum cut rule
	MINWIDTH	minimum width rule
	MINSTEP	minimum step rule
block	MACRO	standard or macro cells
term	PIN	standard or macro cell pin
blockage	OBS	standard or macro cell blockage
pin	PORT	physical pin
rect	RECT	rectangle
polygon	POLYGON	polygon

DEF are summarized in Table 4.2. The structure of the database is described in Figure 4.5. The database is an in-memory, flattened physical design database. In the top level, the database consists of a **technology library**, a **top block** and several **reference blocks**.

Technology library

Technology library stores all metal and cut **layers**, **viadefs**, and design rule **constraints**. A back-of-end-stack layer consists of basic layer information, i.e., type, direction, pitch, offset, as well as all its applied design rule constraints. A viadef holds one or more **shapes** (rectangles or polygons) on two consecutive metal layers with shape(s) in the middle cut layer, realizing physical connection between neighboring metal layers at the same *x*-*y* coordinate. We summarize the design rules that we support in Table 4.3. For definitions, examples, and detailed handling methodology of each rule, please refer to [118].

Table 4.2: Database objects from DEF.

Object	**DEF Keyword**	**Meaning**
block	DESIGN	block-level design
inst	COMPONENTS	instance of standard or macro cell
term	PINS	block-level IO pin
blockage	BLOCKAGES	block-level blockage
net	SPECIALNETS	special net
	NETS	regular net
instTerm		points to a term
instBlockage		points to a blockage
pathSeg		routing segment
via		routing via
patchMetal		routing patch rectangle

Block

The top block describes the flattened logical and physical connections, following the DEF model. There are four major types of objects: **term**, **blockage**, **instance** and **net**. A reference block is a standard or macro cell from LEF, having the same data structure as the top block, except that only terms and blockages are populated.

Terms are IO pins for the top block, and standard or macro cell pins for the reference blocks. Each term consists of one or more physical **pins**. Each pin consists of one or more physical shapes across one or more metal and cut layers.[13]

Blockages are user-defined routing blockages from DEF BLOCKAGES for the top block, and are from LEF OBS statement for reference blocks. We reuse the pin object to hold physical shapes of the blockages.

Instances are from DEF COMPONENTS. Each instance is an instantiation of either a standard cell or a macro block, holding zero or more **instance terms** and **instance blockages**. An instance term points to the related term from its reference block. An instance blockage points to the related blockages from its reference block.

[13]A term including more than one pin with "MUSTJOIN" keyword indicates that the two pins should be physical connected in detailed routing. In this work, we assume that each term holds one physical pin to simplify the description.

Table 4.3: Design rules.

```
// metal layer
WIDTH defaultWidth ;
[MINWIDTH minWidth ;]
SPACINGTABLE
 PARALLELRUNLENGTH {length} ...
  {WIDTH width  {spacing} ...} ...  ;
[SPACING minSpacing SAMENET [PGONLY] ;]
[MINSTEP minStepLength [MAXEDGES maxEdges] ;]
[SPACING eolSpacing ENDOFLINE eolWidth WITHIN eolWithin
 [PARALLELEDGE parSpace WITHIN parWithin [TWOEDGES] ;] ...
[CORNERSPACING
  {CONVEXCORNER | CONCAVECORNER} [EXCEPTEOL eolWidth]
  {WIDTH width SPACING spacing ;} ...  ]  ...  ;
// cut layer
{SPACING cutSpacing [CENTERTOCENTER]
 [  ADJACENTCUTS numCuts WITHIN cutWithin [EXCEPTSAMEPGNET]
  | PARALLELOVERLAP
  | AREA cutArea] ;}...
[SPACING cutSpacingSN [CENTERTOCENTER] SAMENET ;]
```

Nets are from DEF NETS and SPECIALNETS. A net stores its logical connections, and its physical connections, i.e., **pathSegs**, **vias** and **patchMetals**. A pathSeg is a point to point routing wire on a specific layer, defined with the start and end points, width and extensions. A via is an instantiation of viadef at a specific coordinate. A patchMetal is a patching rectangular metal used to satisfy various design rules.

Other types of objects in a block include **boundary**, **trackPattern**, **gcellPattern**, **marker**, etc. The gcellPattern object defines the global routing cells (GCells) [24] in 2D grids;[14] and marker object represents a design rule check (DRC) violation, including the bounding box, layer, violation type and source objects. In our implementation, we also build several assisting objects and structures. Some of the procedures are described in Section 4.1.5. A complete picture and details of the database implementation are visible at [124].

[14]In our work, we derive the GCell size based on global routing solution, in the "route guide" format of ISPD18, ISPD19 and ICCAD19 contests. GR solutions in practice (to our knowledge) commonly use ∼15 M2 tracks as a typical GCell dimension.

4.1.5 Detailed Routing

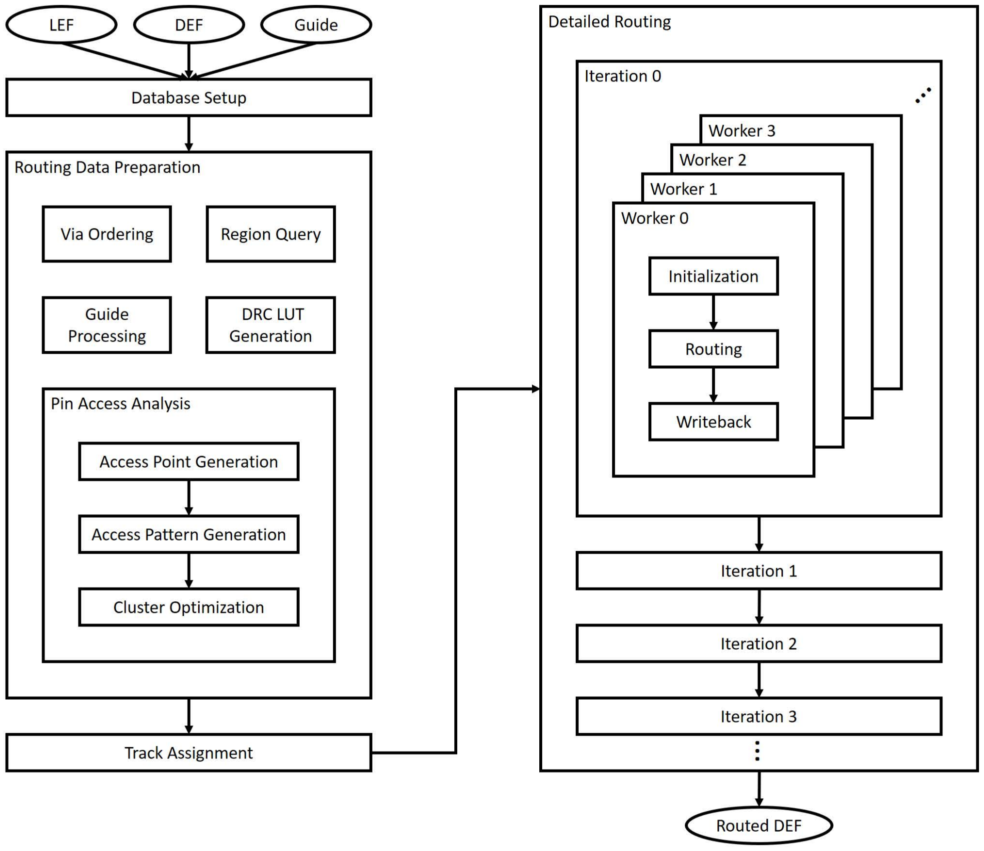


Figure 4.6: Overall detailed routing flow.

In this section, we describe the detailed routing flow. As shown in Figure 4.6, the inputs to the router are LEF, DEF and guide files. LEF and DEF files are industry-standard formats. The route guide file serves as the global routing solution. Given the inputs, we first set up the design database. Next, we take several data preparation steps. Then, we perform track assignment, multiple iterations of detailed routing and output a routed DEF.

Data preparation

The data preparation step processes the design database to generate assisting structures, including via ordering, guide processing, region query, DRC LUT generation and pin access analysis.

Via ordering is the step to select default viadef(s) used for pin access and detailed routing. We sort all viadefs according to (i) number of cuts; (ii) default via property; (iii) enclosure direction; (iv) enclosure area; and (v) enclosure width. In detailed routing, we only use the minimal-enclosed default single-cut viadef, with both lower and upper-layer enclosure along the preferred routing direction. In pin access analysis, in addition to the viadef we use in detail routing, we also use the minimal-enclosed default single-cut viadef, with the lower-layer enclosure orthogonal to the preferred routing direction, and the upper-layer enclosure along the preferred routing direction. Overall, we select one of two viadefs to access the pin, and only use one viadef for all other connections. Figure 4.7 illustrates the ordered viadefs for detailed routing, additional viadef for pin access analysis, and a non-preferred viadef.[15]

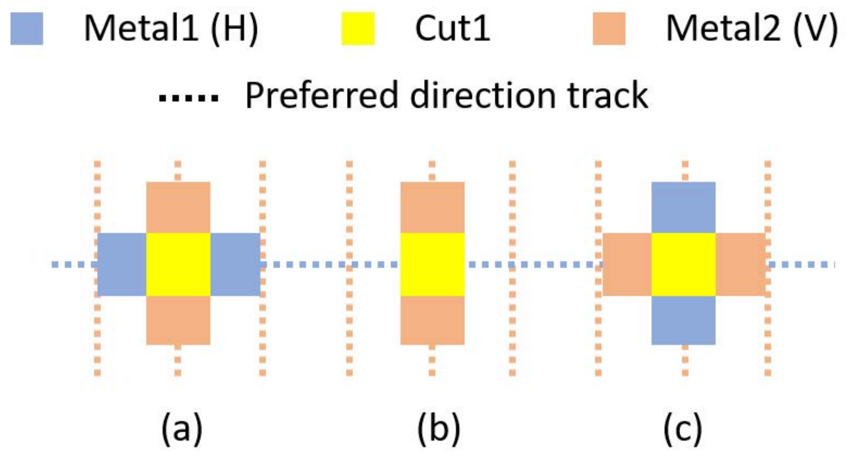


Figure 4.7: Illustrations of ordered viadefs: (a) preferred viadef for detail routing; (b) additional viadef for pin access analysis; and (c) non-preferred viadef.

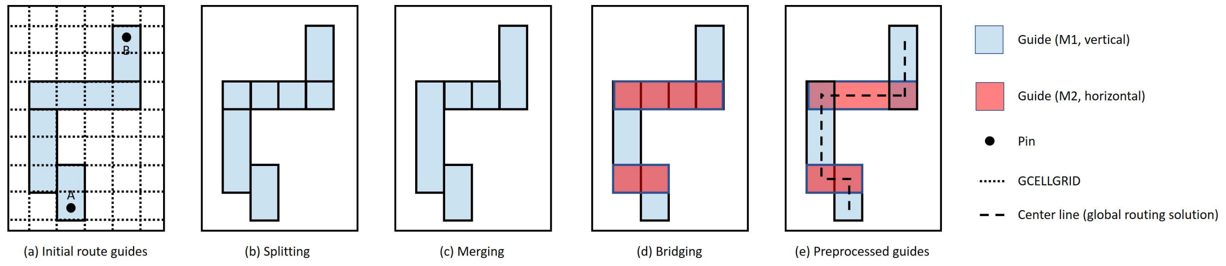


Figure 4.8: Preprocessing: (a) initial route guides; (b) splitting; (c) merging; (d) bridging; and (e) preprocessed guides. The preferred direction for M1 is vertical, and for M2 is horizontal.

[15]Ultimately, the via ordering step should be replaced with a more robust via generation and LEF matching strategy in a future work.

Guide processing [24] [49] is the step to transform a set of input route guides into a standardized tree-like global routing solution.[16] A route guide specifies a rectangular region on a specific metal layer. A global routing solution for a net may contain several route guides on some or all of the metal layers. If we abstract the guide by drawing a center line for each guide along the preferred routing direction, we take the center lines to form a connected graph, as shown in Figure 4.8(e).

To standardize on a guide dimension that is conducive to form a trimmed tree-like global routing solution, we first extract the most common offset and width of all guides to form GCELLGRIDS [24], then process all route guides with **splitting**, **merging** and **bridging** techniques. Given the input guides in Figure 4.8(a), we first split the guide according to the GCELLGRID along the preferred routing direction for each metal layer, as shown in Figure 4.8(b); then merge touching guides along the preferred routing direction, as shown in Figure 4.8(c). Last, for abutting guides along the non-preferred routing direction, we bridge them by creating upper-layer (or, otherwise, lower-layer) guides, as shown in Figure 4.8(d).

The above procedures guarantee a connected global routing solution as long as the input guides satisfy the assumption described in [24]. To remove redundant edges (i.e., loops) in a global routing solution, we further perform A* search from any pin to all other pins through the processed guides. All off-path guides are removed.

Region query is the data structure for fast shape queries. The inputs to the region query engine is a bounding box on a specific layer. The outputs are all intersecting shapes, in the form of $\{$bbox, owner$\}$ pairs. For polygon shapes, we decompose the polygon into rectangles to be used in the region query engine. The owner belongs to one of the following types: term, instTerm, blockage, instBlockage, pathSeg, via or patchMetal.

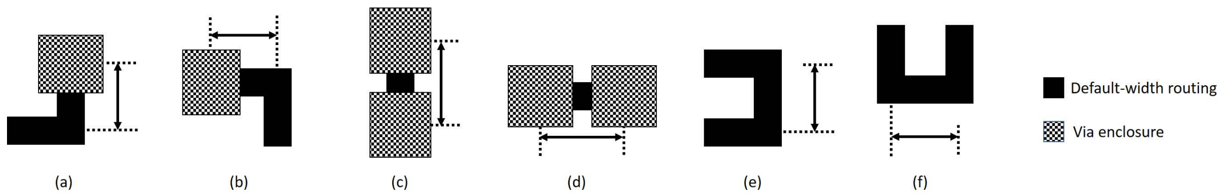


Figure 4.9: DRC LUT: (a) via to jog (vertical); (b) via to jog (horizontal); (c) via to via (vertical); (d) via to via (horizontal); (e) jog to jog (vertical); and (f) jog to jog (horizontal).

[16]Ultimately, the solution quality of detailed routing may be improved with an input of a better global routing solution that satisfies our guide processing behavior in a future work.

LUT generation is the step to construct assisting data structure to avoid same-net design rule check violations. In grid-based path search, we use object cost (described in Section 4.1.6) to avoid potential DRCs to existing objects. To prevent DRCs within the current path, i.e., same-net violation, we characterize the minimum default-width routing length between any two-object pair of an up via, a down via and a jog, on all metal layers, and in all directions. Figure 4.9 illustrates three types of minimum length requirement: via to jog, via to via, and jog to jog, in both x and y directions. In our implementation, we characterize separately for the up via and down via. In grid-based path search, we apply additional cost if the minimum length between vias and/or jogs is not satisfied.

The pin access analysis framework is described in Chapter 3.

Track assignment

We adopt a simplified version of greedy track assignment [96]. To reduce the problem size and lay a foundation for future parallel implementation, we perform the track assignment every 50 GCell panels. Each GCell panel has length along the preferred routing direction and spans 50 GCell heights. The initial track assignment is applied once on all horizontal layers, then on all vertical layers. According to [96], we then perform one iteration of track reassignment to optimize the solution quality.

Detailed Routing

Given the track assignment result, we perform multiple iterations of detailed routing. In each iteration, we partition the design into 7×7, non-overlapping GCell-aligned clips, and create one **detailed routing worker** for each clip. Each detailed routing worker first initializes its own data structures (worker database) from the global database, then performs routing and design rule checking, all without touching the global database. Last, each worker commits the changes by writing back to the global database. In alternate iterations, we shift the partitioning of 7×7 clips with an offset of 0 and -4 to enable optimization at clip boundaries. We describe the detailed routing flow inside the detailed routing worker in Section 4.1.6.

In the construction of a detailed routing worker, each clip comes with three bounding boxes: **standard**, **DRC** and **extended box**. The standard box is the above-mentioned 7×7, non-overlapping GCell-aligned clip. The detailed routing worker can only modify objects with their center lines on or within the standard box. The DRC box is slightly larger than the standard box, enclosing the bounding box of all modifiable objects. We only count and writeback those markers intersecting with the DRC box. The extended box is slightly larger than the DRC box, allowing design rule check across the DRC box. In the detailed routing worker database, all objects within the extended box are constructed locally. Only the objects that are on or within the standard box are modifiable, while other objects are fixed. The fixed objects are used for cost calculation and design rule checking.

4.1.6 Detailed Routing Worker

In this section, we describe the methodology to perform gridded, A*-based detailed routing inside the detailed routing worker. We first describe the grid graph structure and various types of costs. Then, we describe the overall ripup-and-reroute flow of a detailed routing worker. Last, we detail the methodology to route one net.

Grid graph

The grid graph is an essential part of detailed routing because the path search algorithm works directly on the grid graph, and various costs and properties are associated with the grid vertices and edges in the grid graph. In TritonRoute, we build a non-regular-spaced 3D grid graph supporting **irregular tracks** and **off-track routing**.

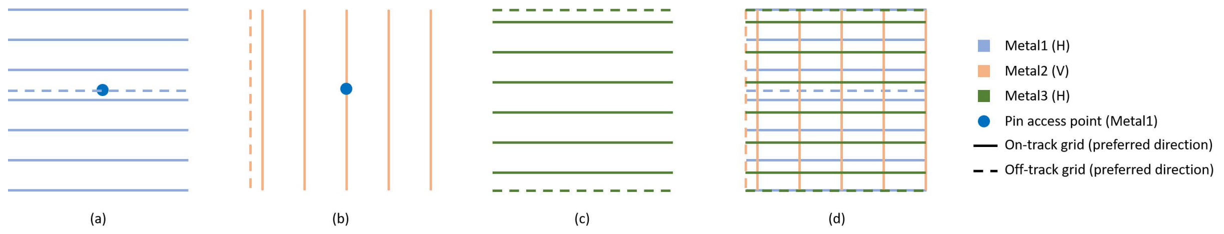

(a)
(b)
(c)
(d)
Metal1 (H)
Metal2 (V)
Metal3 (H)
Pin access point (Metal1)
On-track grid (preferred direction)
Off-track grid (preferred direction)

Figure 4.10: Grid graph: (a) preferred-direction grid lines on Metal1; (b) preferred-direction grid lines on Metal2; (c) preferred-direction grid lines on Metal3; and (d) overlay of grid lines (3D grid graph projected onto the x-y plane).

Construction. We now describe how to generate the preferred-direction grid lines on each metal layer. We first form all grid lines that are on-track – i.e., align with the DEF TRACKS definitions. Then we form all grid lines that are off-track – i.e., the center lines along the preferred direction for any existing pathSegs, vias and pin access points. We also form the grid lines on the boundary. We do not generate the grid lines in the non-preferred direction. However, bi-directional routing is still available as described later.

Figure 4.10 shows how we form the grid lines. Figure 4.10(a) shows horizontal Metal1, with 7 regular-spaced tracks from DEF. The Metal1 pin has an access point with an off-track y-coordinate. Thus, we create an off-track grid line according to the pin access point location. Figure 4.10(b) shows vertical Metal2, with 5 regular-spaced tracks from DEF. We additionally create an off-track grid on the left boundary. By always creating grid lines along the boundaries of the routing region, we make sure that at least one path exists in the grid graph in any direction, in the case that no on- and off-track grid lines exist (e.g., given a small routing region). Since the center line of the Metal1 pin access point aligns with a Metal2 track, we do not build additional off-track grid lines on Metal2. Similarly, we build grid lines on Metal3. Note that Metal1 and Metal3 grid lines do not necessarily align.

In Figure 4.10(d), we show the overlay of x- and y-direction grid lines. The grid vertices are formed by intersecting all x- and y-direction grid lines, and repeating $|Z|$ times along the z-direction. Each vertex has six neighbors (except the boundary vertices) – west, east, south, north, down and up; this is the 3D grid (projected onto the x-y plane) that we use in TritonRoute.

Table 4.4: Edge properties.

Type	Name	Meaning
boolean	isEnable	whether the edge exists in path search
boolean	isOnTrack	whether the edge is on track
boolean	isOnPrefDir	whether the edge is on the preferred direction
viadef	specialVia	special via
int	objCost	object cost
int	markerCost	marker cost

Edge. The edge properties are summarized in Table 4.4. As shown in Figure 4.10, not every grid line exists in every metal layer. We use *isEnable* to show whether the edge exists in the path search. A planar edge in the preferred direction is enabled if it is on a current layer grid line. A planar edge in the non-preferred direction is enabled if it is on an upper-layer grid line (if any, otherwise lower-layer). Via edges are enabled between any two preferred-direction grid lines on neighboring metal layers. For each edge, we use *isOnTrack* to show whether the edge is on track; we use *isOnPrefDir* to show whether the edge is on the preferred direction. For a via edge, *specialVia* indicates whether the router should choose a special via instead of the default via. Only pin access points may have this special via property. We preprocess and mark relevant via edges for all up-via pin accesses (using non-default via). There are two types of costs associated with each edge, object cost and marker cost. We describe these costs in Section 4.1.6.

Table 4.5: Vertex properties.

Type	**Name**	**Meaning**
enum	prevDir	incoming direction
boolean	isSrc	whether the vertex is the source
boolean	isDst	whether the vertex is the destination

Vertex. The vertex properties are summarized in Table 4.5. In A*-based path search, after a path is found, we only know the ending vertex. We use *prevDir* to indicate the incoming direction of the current vertex so that we are able to trace back the path. We use *isSrc* (resp. *isDst*) to indicate whether the vertex is a source (resp. destination).

Routing Cost

We use two types of costs: **object cost**, and **marker cost**. Overall, object cost is applied around an existing shape. This cost preemptively guides the path search to go around existing objects to avoid potential DRCs. The marker cost is applied around an existing DRC marker. In the ripup-and-reroute scheme, this cost helps the nets to be routed avoiding the DRC hotspots given the history of DRC data.

Object cost is the cost originated from an object, and stored in neighboring edges to the object. We modify this cost whenever the worker database adds or removes an object, e.g., at the time of database initialization, after net ripup, or after routing of one net. We use the object cost to prevent potential design rule check violations. The evaluation of object cost is non-precise but quick, and does not invoke the DRC engine.[17] We support three types of spacing rules for object cost: (i) SPACINGTABLE PARALLELRUNLENGTH; (ii) SPACING ENDOFLINE; and (iii) SPACING (cut).

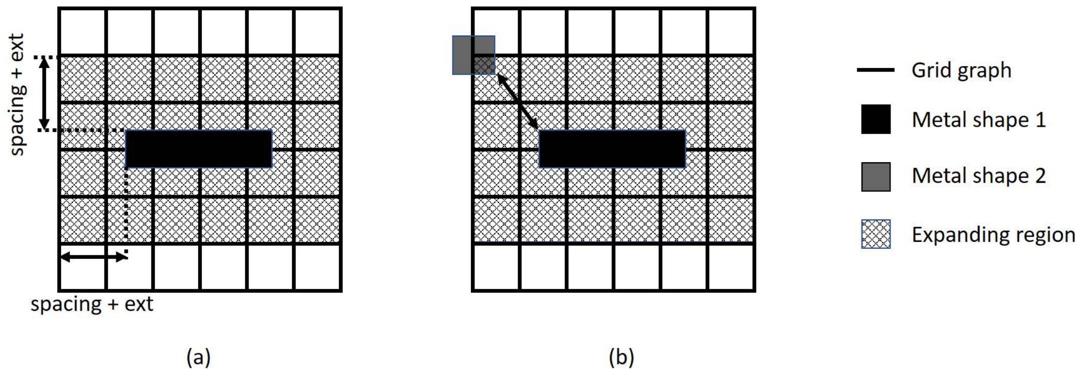


Figure 4.11: Object cost from parallel run length spacing: (a) expanding region; and (b) shadow object.

For parallel run length spacing, given a **target object**, we first draw an expanding region in which objects on the intersecting edges may cause DRCs, as shown in Figure 4.11(a). The expanding region extends beyond the target object up to the maximum required spacing plus half the default width for planar edges, and half the via enclosure for via edges. We then assume a **shadow object** (either a default-width pathSeg or a via) on each of the neighboring planar and via edges, and check against the target object, as shown in Figure 4.11(b). For a pathSeg on a planar edge, since the exact length of the shadow object can be arbitrarily longer than the edge length, we add pessimism by assuming maximum parallel run length between the two objects to accelerate convergence. The maximum parallel run length is the length of the target object regardless of the actual parallel run length. For each via edge, we assume a default via, or the special via stored with the edge, and check the via enclosure against the target object. The parallel run length between a shadow via enclosure and the target object is calculated by their actual parallel run length. We modify the cost of the edge if there is a violation. Here, the modification of the costs also helps to avoid short violations since the expansion region implicitly includes those edges that may have potential short violations with the target object.

[17]We do not have a metric for "precision" of object cost evaluation. The goals of the quick object cost evaluation, in decreasing priority order, are: (i) quickness, and (ii) help avoidance of repeated cycles of violations (e.g., arising due to DRC marker cost in A* search). In practice, we see that our use of quick object cost evaluation – which naturally must be pessimistic – helps avoid cycling.

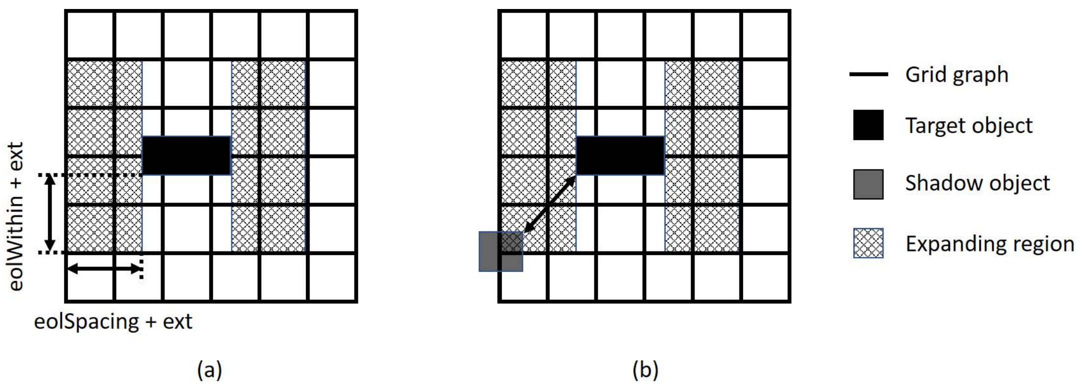


Figure 4.12: Object cost from end-of-line spacing: (a) expanding region; and (b) shadow object. The preferred routing direction is horizontal.

For end-of-line spacing, we only check the target object if it is a via, and the spacing is only checked along the preferred routing direction of the metal layer. Spacing orthogonal to the preferred routing direction is not checked to avoid pessimism since almost all jogs end with a preferred-direction routing or a default via, making the line end a non-end-of-line edge. Figure 4.12 illustrates the procedure.

For cut spacing, given a target via, we check all neighboring via edges which could potentially cause a cut spacing violation. For each via edge, we assume a default via (or the special via stored with the edge) and check against the target via. We modify the cost of the via edge if there is a violation.

The object cost has no history. For example, an object cost is added to the neighboring edges of the target object after the object is created, and subtracted from the neighboring edges of the target object after the object is removed. The object cost calculation supports same-net overriding, blockage spacing overriding and other exceptions. For more details pertaining to this and other parts of our discussion, please refer to [124].

Marker cost is the cost applied according to the DRC markers after each call to the DRC engine. For each marker, we get all objects touching the marker, and add costs to the nearest edge(s) that are used to form the objects. The marker cost has history within the detailed routing worker. For example, a marker cost is added to an edge and decayed over time ($currIter$ in Algorithm 21), but is never subtracted due to the removal of a specific marker. Here, marker cost history only persists within the detailed routing worker. There is no history between detailed routing iterations shown in Figure 4.6.

Detailed routing flow

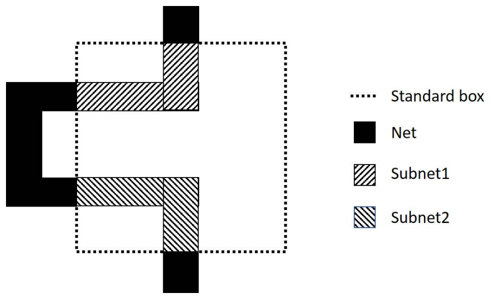


Figure 4.13: Local netlist construction: two disjoint subnets constructed in the detailed routing worker from one global net.

Now we describe the detailed routing flow inside a detailed routing worker. In Algorithm 21, Line 2 first initializes the worker database from the global database. In this step, we construct a local netlist from the connectivity of routing objects. Figure 4.13 shows an example, where a single net passes through the standard box twice, with two parts disjoint. In this case, we construct two subnets so that ripup-and-reroute does not change the connectivity of the net.

In Lines 3 – 20, we perform up to *maxIter* iterations of ripup-and-rerouting.[18] In each iteration, we ripup the problematic nets and reroute each one sequentially. Line 4 adds the marker cost according to all existing markers. Line 5 gets all nets that are associated with markers. We order the nets according to their distance to the nearest marker and route them sequentially. Line 6 rips up those nets and Line 7 subtracts the object cost from the ripped-up objects. Here, the boundary objects outside the standard box are not removed and their object costs remain. Since nets are routed sequentially, according to the net ordering, we would like to avoid the i^{th} net blocking the pin access of the $j^{th}(j > i)$ net. In Line 8, we reserve the pin access of all unrouted nets (ripped-up nets) by adding the object cost of their preferred pin access (an up via) as if those pin access points are used.

In Lines 9 – 15, we route each net once according to the net ordering. Before routing, Line 10 unreserves the pin access for the current net by subtracting the corresponding object cost of the preferred pin access (up via). Line 11 subtracts the object cost for the boundary objects outside the standard box to

[18]Note that this number of iterations is different from the number of "outer" iterations in Figure 4.6. For the results that we report in this work, we perform seven (outer) iterations. The *maxIter* number of iterations in Algorithm 21 defines the maximum number of ripup-and-reroute iterations a net inside a DRWorker can undergo. In the current implementation/results represented in this work, we use (1, 4, 4, 4, 4, 4, 4) as the maxIter (for ripup-and-reroute) for each net in the seven "outer" iterations, respectively.

avoid unnecessary costs when we connect the net to the boundary pin. Line 12 routes the current net. Line 13 adds the object cost for all the newly routed objects. Line 14 adds back the object cost for boundary objects to prevent design rule violations between these objects to the remaining unrouted nets. Lines 16 – 19 perform design rule checking, and terminates the ripup-and-reroute flow once the clip is clean.

Line 21 commits the worker database back to the global database.

Algorithm 21 Detailed routing flow

```
1: Input: worker database, worker markers $markers$
2: WorkerDBInit()
3: while $currIter < maxIter$ do
4:     addMarkerCost($markers$)
5:     $nets \leftarrow$ getMarkeredNets($markers$)
6:     ripupNets($nets$)
7:     subObjCost($nets$)
8:     reservePA($nets$)
9:     for all net $\in$ nets do
10:        unreservePinAccess($net$)
11:        subBoundCost($net$)
12:        routeOneNet($net$)
13:        addObjCost($net$)
14:        addBoundCost($net$)
15:    end for
16:    DRC($nets$)
17:    if numMarkers $= 0$ then
18:        break
19:    end if
20: end while
21: DBCommit()
```

Routing one net

Flow. We now describe the methodology to route one net in a detailed routing worker. In our current implementation, in the standard box, a net is either fully routed or unrouted, but not partial routed. Algorithm 22 describes the methodology to route one net. Line 2 gets all unconnected pins, including standard box boundary pins and pins from instTerm and term. Line 3 holds the set of visited grid vertices, and we initialize the set to be empty. Lines 4 and 5 select the source pin to perform path search and remove it from the unconnected pins. To select the source pin, we first calculate the center of gravity for all pins in the x-y plane, then select the pin furthest away from the center of gravity as the source. Line 6 performs the initialization described later. In Lines 7 – 11, we perform the path search as long as there are still unconnected pins. After path search, we update the grid graph in preparation for the next round of path search. The writeDB function backtraces the path to create the routing objects according to the path.

Algorithm 22 Route one net

```
 1: Input: net n, grid graph G
 2: unConnPins ← allPins(n)
 3: visitedGrids ← ∅
 4: srcPin ← selectSrcPin(unConnPins)
 5: unConnPins.removePin(srcPin)
 6: init(n, srcPin, unConnPins, visitedGrids, G)
 7: while not isEmpty(unConnPins) do
 8:     path ← search(visitedGrid, G)
 9:     update(n, path, unConnPins, visitedGrids, G)
10:     writeDB(n, path)
11: end while
```

During backtracing, we calculate the total metal area and add necessary patch metals to satisfy the minimum area rule. The patch metals are always created with default routing width along the preferred routing direction. In our implementation, we also build assisting structures to calculate necessary patch metal area for objects connected to the boundary pin. Figure 4.14 gives two examples of patch metal addition. We assume the preferred routing direction is horizontal. We do not allow the patch metal to exceed the standard box. If there are more than one patch metal choices, e.g., adding to the left or to the right of a routing object, we choose the one with smaller object cost. The path search is completed once all pins are connected. The path search algorithm is described in Algorithm 24. The update function is described in Algorithm 25.

Initialization. Algorithm 23 describes the initialization procedure. In Line 2, we first reset the previous direction flag for each grid vertex. In Lines 3 – 6, we set the source flag for all vertices on the access points of the source pin, and add the vertices to the visited grids. In Lines 7 – 11, we set the destination flag for all vertices on the access points of all destination pins. After initialization of the grid graph, the core path search algorithm does not need to look for objects and properties of the net, which is beneficial to the runtime.

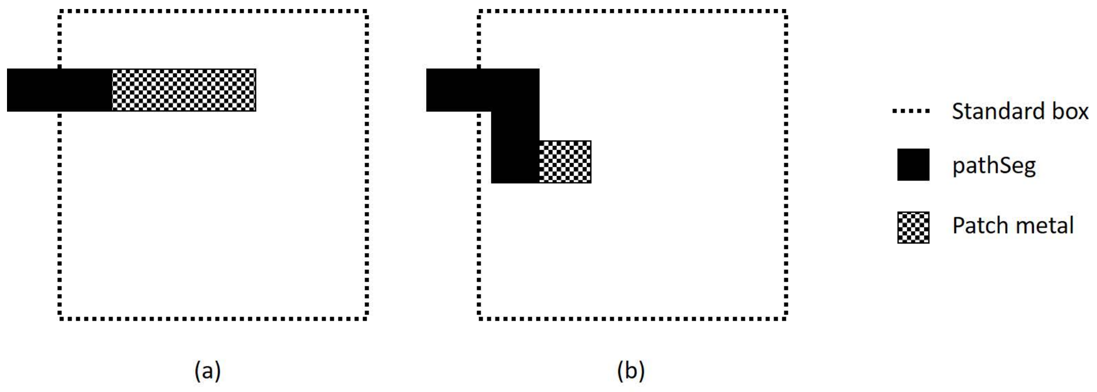


Figure 4.14: Minimum area patch metal: (a) patch metal considering area outside of standard box; and (b) patch metal always along the preferred routing direction even if the routing ends in the non-preferred direction.

Algorithm 23 Initialization

```
1: Input: n, srcPin, unConnPins, visitedGrids, G
2: G.resetPrevDir()
3: for all grid ∈ srcPin do
4:     G.setSrc(grid)
5:     visitedGrids.add(grid)
6: end for
7: for all dstPin ∈ unConnPins do
8:     for all grid ∈ dstPin do
9:         G.setDst(grid)
10:    end for
11: end for
```

Path search. Algorithm 24 details the path search. The A*-based path search is based on a priority queue. Each element in the priority queue is an element of the search's wavefront, representing that a path exists from the source up to the wavefront grid vertex. In Lines 3 – 5, we first push all visited grids (source) to the queue as the initial wavefront vertices. Then in Lines 6 – 16, we pop the wavefront vertex with the least cost. We use the previous direction to indicate whether the wavefront vertex has been visited before. Lines 9 – 11 skip the wavefront vertex if it has been visited before. In Lines 12 – 14, we check whether the wavefront vertex is the destination, and return the path when reaching the destination. Otherwise, we expand the wavefront vertex by pushing its neighbors into the priority queue (with proper cost) as new wavefront vertices.

Algorithm 24 Search

```
1: Input: visitedGrids, G
2: Initialize wf
3: for all grid∈visitedGrids do
4:     wf.push(grid)
5: end for
6: while not isEmpty(wf) do
7:     currGrid ← wf.top()
8:     wf.pop()
9:     if hasPrevDir(currGrid) then
10:        continue
11:    end if
12:    if isDst(currGrid) then
13:        return path
14:    else
15:        expand(currGrid)
16:    end if
17: end while
```

Here, the cost in the priority queue is the A* cost, consisting of an existing path cost and an estimated future cost, as shown in Equation (4.6). Whenever we expand from a wavefront vertex to its neighboring vertex, the existing cost is the cost from the wavefront vertex plus the cost to its neighbor, as shown in Equation (4.7). The cost is the sum of edge length, plus 8× edge length if the edge has a non-zero object cost, and 64× edge length if the edge has a non-zero marker cost. In addition, we apply a penalty p if any match to the DRC LUT is found. The estimated future cost is the Manhattan distance to a pre-determined destination, as shown in Equation (4.8). If there are more than one unconnected pins to be connected, the pre-determined destination is the bounding box of the unconnected pin that is the closest to the bounding box of all visited grids. The Manhattan distance in z-direction (between two neighboring metal layers) is calculated as 4× the lower metal layer pitch.

$$cost_{tot} = cost_{wf'} + cost_{est} \tag{4.6}$$

$$cost_{wf'} = cost_{wf} + len_e + objCost_e + markerCost_e + p \tag{4.7}$$

$$cost_{est} = dist_{wf',dst} \tag{4.8}$$

As described in Lines 9 – 11, we avoid expanding an already-visited vertex by checking its previous directional flag. In an ideal A*-based path search with a consistent path cost and a lower-bounded estimated future cost, each vertex only needs at most one visit to get the minimum cost path. However, considering the inconsistent nature of the penalty applied from the DRC LUT, the worst-case complexity of A*-based path search becomes $O(n^2)$. To balance the tradeoff between runtime and solution quality, we write the previous direction to a vertex only after two more wavefront expansions are performed from that vertex.

Update. Algorithm 25 describes the methodology to update the grid graph. In Line 2, we reset the previous direction flag for every grid vertex in preparation of the next path search. In Lines 3 – 6, we set the source flag for every grid vertex along the path. We then add these grid vertices to the visited grids. Here the source flag and the visited grids serve the same purpose as they both identify the new sources for the next round of path search. However, visited grids are stored in a vector-like container to allow us to initialize the wavefront for the next path search in batches. In Lines 7 – 15, we identify the destination pin that we route to in the current round of path search, remove it from the unconnected pins, and reset the destination flag on all access points of the destination pin.

We now describe two special cases for **pin feedthrough**. Pin feedthrough describes a scenario where two (or multiple) parts of the net are connected to different access points of the same pin. We can either enable, or disable pin feedthrough. Disabling pin feedthrough forces that only one access point per pin can be used.

In case of enabling feedthrough, all access points of the destination pin, even those we do not route to, now become new sources for the next round of path search, as shown in Lines 12 –14.

In case of disabling feedthrough, special handling methodology is needed for the first source pin of the net, described in Lines 17 – 24. Recall that in Line 4 of Algorithm 23, we set the source flag on all access points of the source pin. Given feedthrough disabled, we must reset the source flag on all unused access points of the source pin once the first path search completes.

Algorithm 25 Update

```
1: Input: $n$, $path$, $unConnPins$, $visitedGrids$, $G$
2: $G$.resetPrevDir()
3: for all $grid \in path$ do
4:     setSrc($grid$)
5:     $visitedGrids \leftarrow$ add($grid$)
6: end for
7: $endGrid \leftarrow path$.end()
8: $currDstPin \leftarrow$ findPin($endGrid$)
9: $unConnPins$.removePin($currDstPin$)
10: for all $grid \in currDstPin$ do
11:     $G$.resetDst($grid$)
12:     if isAllowPinAsFeedThrough() then
13:         $G$.setSrc($grid$)
14:     end if
15: end for
16: $beginGrid \leftarrow path$.begin()
17: if not isAllowPinAsFeedThrough() then
18:     if findPin($beginGrid$) then
19:         $currSrcPin \leftarrow$ findPin($beginGrid$)
20:         for all $grid \neq beginGrid \in currSrcPin$ do
21:             $G$.resetSrc($g$)
22:         end for
23:     end if
24: end if
```

Inefficiency in existing ripup-and-reroute flow

Use of ripup-and-reroute to resolve DRC can rely heavily on net ordering. Figure 4.15 illustrates potential inefficiency in resolving DRC in a 2D routing scheme. If *net0* is always routed before *net1*, it takes seven iterations to explore routing solutions with DRC if the DRC does not provide sufficient cost (part (a) of the figure), before a DRC-clean solution is found (part (b)). In a real-world scenario, the interactions among different nets are much more complicated than in Figure 4.15. Hence, simple net ordering heuristics (e.g., shuffling) may not efficiently converge to a feasible solution. Figure 4.15(a) illustrates how ripup-and-reroute flows in our [51] and other previous works suffer from being only aware of the short violation between *net0* and *net1*, while leaving unutilized the fact that *net0* is routed before *net1*. The latter is a key piece of information that can help improve DRC convergence.

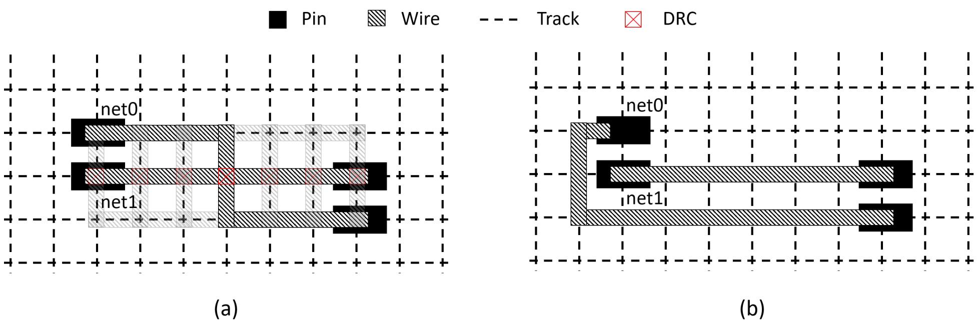


Figure 4.15: Illustration of DRC convergence depending on net ordering: (a) routing solutions with DRC and (b) a DRC-clean routing solution.

Queue-based ripup-and-reroute flow

In this work, we propose a queue-based ripup-and-reroute flow to improve efficiency of ripup-and-reroute, thus improving DRC convergence and runtime. Rather than relying on ordering a certain number of nets and rerouting them in a batch followed by a full design rule check on all objects, we introduce an incremental route-and-check flow, based on the use of a FIFO queue and the capability shown in Chapter 3. Each net is rerouted and design rule-checked incrementally. When we pop a net from the queue, we perform either (1) rerouting and incremental design rule checking or (2) design rule checking only. If new violations are found related to the popped net, we push relevant nets back to the queue. Each net in the queue is designated for a task that is either (1) or (2). Each element in the queue is a 3-tuple that contains a net, associated with (i) task type (type (1) is *true* since it does perform reroute), and (ii) number of times that the net has been rerouted when it is pushed to the queue. Note that if this number does not match the actual number of times that a net has been routed, we will skip routing the net. We discuss the details of our ripup-and-reroute queue in the following paragraphs.

To illustrate how we push nets to the queue, we introduce the concept of **aggressor** and **victim**. Recall that in the Figure 4.15, *net0* is routed first. When *net1* is being routed, *net1* attempts to avoid DRC, but the detour cost is so large that *net1* routes across *net0*. In this case, we consider *net0* as the aggressor and *net1* as the victim because *net0* invades the solution space where *net1* can achieve DRC-clean routing solution. Therefore, the aggressor should be ripped up and rerouted next and the victim should be DRC checked after the aggressor is rerouted. In general, after a certain net is routed, if the net has any violation with other nets, the net that is lastly routed is considered as the victim and the other nets are considered as the aggressors. We first push all aggressors for task (1), then push the victim for task (2).

We describe the queue-based ripup-and-reroute flow in Algorithm 26. Line 2 first initializes the worker database. Line 3 initializes the ripup-and-reroute queue with existing DRC markers. Line 4 adds the marker cost for all input markers. In Line 5–20, we perform iterative ripup-and-reroute until the queue is empty. Lines 6–8 obtain the information of the front element of the queue. Line 9 pops the front element. Lines 10–16 rip up and reroute the *net* and decays marker costs only if the *net* is set for reroute and it has not been rerouted more than *maxIter* times. Line 17 performs incremental DRC check for the *net*. For the DRC markers associated with the *net*, Line 18 adds the marker cost and Line 19 updates the *queue* accordingly. Line 21 performs DRC check for all nets in the worker and Line 22 commits the routing from the worker.

Algorithm 26 Queue-based routing flow

```
 1: Input: database, DRC markers markers
 2: WorkerDBInit()
 3: queue.init(markers)
 4: addMarkerCost(markers)
 5: while queue.size() do
 6:    net = queue.front.net
 7:    isRoute = queue.front.isRoute
 8:    numReroute = queue.front.numReroute
 9:    queue.pop_front()
10:    if isRoute and numReroute < maxIter then
11:       ripupNet(net)
12:       subObjCost(net)
13:       routeOneNet(net)
14:       addObjCost(net)
15:       decayMarkerCost()
16:    end if
17:    netMarkers = GC(net)
18:    addMarkerCost(netMarkers)
19:    queue.update(netMarkers)
20: end while
21: GC()
22: DBCommit()
```

We illustrate the operation of a ripup-and-reroute *queue* in Figure 4.16 with three two-pin nets to be routed in 2D. Each figure shows the layout and the corresponding elements in the *queue* before a net is to be routed. Each net has an associated counter to keep track of the number of times that the net has been routed. Such a counter prevents a net from being (i) routed more than the number of allowed ripup-and-reroute iterations (i.e., *maxIter*); and (ii) routed unnecessarily for a DRC that has already been addressed (see Figures 4.16(e) and (f), for example). Figure 4.16(a) shows that the three nets are initially routed in the order of *net0*, *net1* and *net2*. Figure 4.16(b) illustrates the layout and *queue* after *net0* is routed. Figure 4.16(c) shows that after *net1* is routed, there is a short violation between *net0* and *net1*. Considering that *net0* is routed before *net1*, *net0*, as the aggressor, is pushed to the back of the *queue* for rerouting. *net1*, as the victim, is pushed to the back of the *queue* for DRC checking. Similarly, Figure 4.16(d) shows that after *net2* is routed, *net2* has a short violation with *net0*. Therefore, *net0* is pushed to the back of the *queue* for rerouting and *net2* is pushed to the back of the *queue* for DRC checking. Figure 4.16(e) shows that after *net0* is rerouted, both of the two short violations are resolved and DRC checking from *net1* does not detect new violations. Figure 4.16(f) shows that when *net0* is popped from the *queue*, the routing is skipped because *net0* has been routed twice while the routing counter indicates that *net0* is pushed to the *queue* for routing when it was routed only once. At this point, the last two elements in the *queue* are popped without pushing new elements to the *queue*. Therefore, the routing for the three two-pin nets are completed.

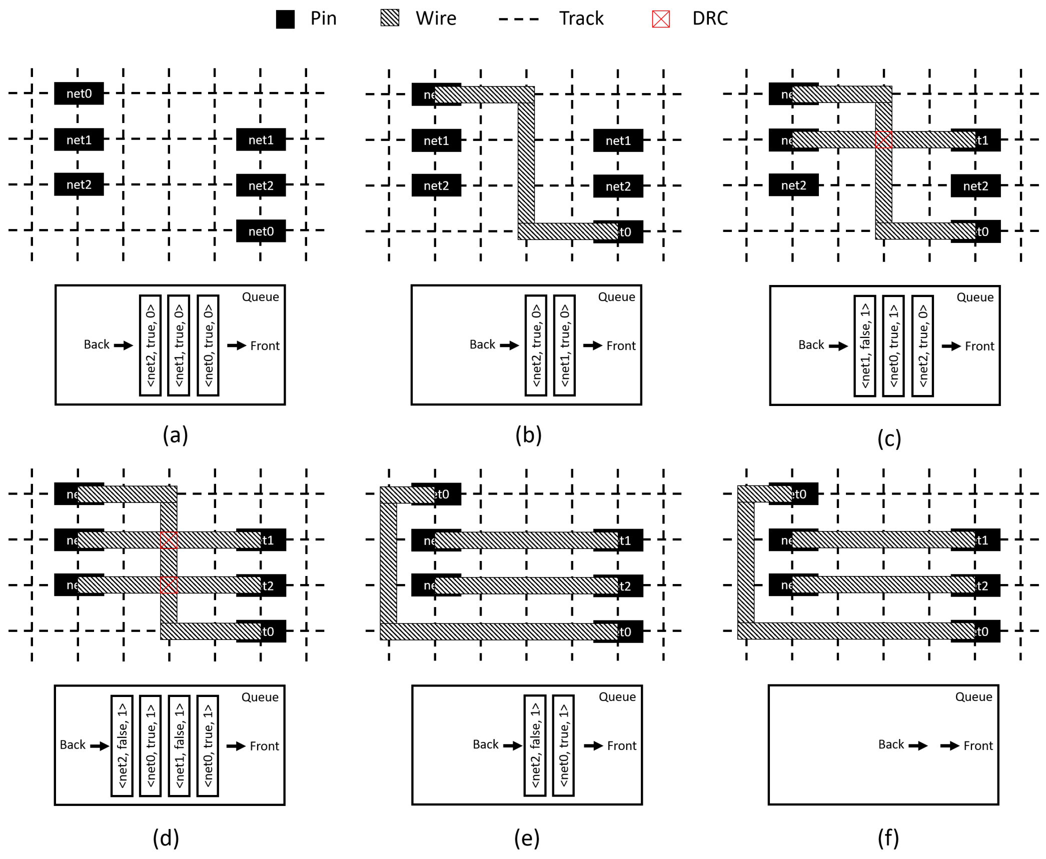


Figure 4.16: Illustration of operation of ripup-and-reroute queue on three two-pin nets.

Ripup-and-reroute queue update

We now describe the procedure to initialize the ripup-and-reroute *queue* based on a given list of DRC markers and update the ripup-and-reroute *queue* based on a given list of DRC markers. Algorithm 27 describes the ripup-and-reroute *queue* update procedure. Lines 2–3 initialize a *uniqueAggressors* set and a *uniqueVictims* set. Pushing redundant element into the *queue* can cause exponential increase in size of the *queue*. Lines 4–9 iterates all DRC marker, obtain the aggressors and the victim of each marker and update the two aforementioned sets accordingly. Lines 10–12 push all unique aggressors involved in DRC markers to the *queue* for ripup-and-reroute. Lines 13–15 push all victims of the markers to the *queue* for DRC checking.

Algorithm 27 Update ripup-and-reroute queue

```
1: Input: ripup-and-reroute queue queue, DRC markers markers
2: uniqueAggressors = $\emptyset$
3: uniqueVictims = $\emptyset$
4: for all marker $\in$ markers do
5:     for all aggressor $\in$ marker.getAggressors() do
6:         uniqueAggressors.insert(aggressor)
7:     end for
8:     uniqueVictims.insert(marker.getVictim())
9: end for
10: for all aggressor $\in$ uniqueAggressors do
11:     queue.push_back(<aggressor, true, 0>)
12: end for
13: for all victim $\in$ uniqueVictims do
14:     queue.push_back(<victim, false, 0>)
15: end for
```

4.1.7 Experiments

In this section, we present experimental setup and results. We implement our router in C++ with LEF/DEF parser [117] and Boost C++ libraries [119]. We enable multi-threading with OpenMP [122]. We perform experiments using the ISPD-2018 and ISPD-2019 benchmark suites [61][66] with overall 20 testcases in 65nm, 45nm and 32nm technology nodes, with up to 899K standard cells and 895K nets. Compared to the ISPD-2018 benchmark suite, the ISPD-2019 benchmark suite includes more advanced routing rules. We summarize the benchmark information in Table 4.6.

Table 4.6: Benchmark information [61][66].

Benchmark	**#std**	**#blk**	**#net**	**#pin**	**#layer**	**Die size**	**Tech. node**
ISPD-2018							
ispd18_test1	8879	0	3153	0	9	0.20×0.19mm^2	45nm
ispd18_test2	35913	0	36834	1211	9	0.65×0.57mm^2	45nm
ispd18_test3	35973	4	36700	1211	9	0.99×0.70mm^2	45nm
ispd18_test4	72094	0	72401	1211	9	0.89×0.61mm^2	32nm
ispd18_test5	71954	0	72394	1211	9	0.93×0.92mm^2	32nm
ispd18_test6	107919	0	107701	1211	9	0.86×0.53mm^2	32nm
ispd18_test7	179865	16	179863	1211	9	1.36×1.33mm^2	32nm
ispd18_test8	191987	16	179863	1211	9	1.36×1.33mm^2	32nm
ispd18_test9	192911	0	178857	1211	9	0.91×0.78mm^2	32nm
ispd18_test10	290386	0	182000	1211	9	0.91×0.87mm^2	32nm
ISPD-2019							
ispd19_test1	8879	0	3153	0	9	0.15×0.15mm^2	32nm
ispd19_test2	72094	4	72410	1211	9	0.87×0.59mm^2	32nm
ispd19_test3	8283	4	8953	57	9	0.20×0.20mm^2	32nm
ispd19_test4	146442	7	151612	4802	5	1.60×1.55mm^2	65nm
ispd19_test5	28920	6	29416	360	5	0.91×0.91mm^2	65nm
ispd19_test6	179881	16	179863	1211	9	1.36×1.33mm^2	32nm
ispd19_test7	359746	16	358720	2216	9	1.58×1.52mm^2	32nm
ispd19_test8	539611	16	537577	3221	9	1.80×1.71mm^2	32nm
ispd19_test9	899341	16	895253	3221	9	2.01×2.15mm^2	32nm
ispd19_test10	899404	16	895253	3221	9	2.01×2.15mm^2	32nm

In the following, based on the ISPD-2018 and ISPD-2019 benchmark suites, we perform (i) DRC convergence comparison between detailed routings with and without ripup-and-reroute queue, (ii) comparison between our detailed routing work and known best detailed routing solutions from all published academic detailed routers, (iii) DRC convergence comparison between our global routing solutions and contest global routing solutions, and (iv) comparison between our global-detailed routing flow and the other academic global-detailed routing flow. We perform additional detailed routing experiment with a RISC-V processor [78] in 14nm. All experiments are performed using eight threads on an Intel Xeon 2.4GHz server.

Queue-based ripup-and-reroute DRC convergence study

In this subsection, we compare the detailed routing (DR) based on ISPD-2018 and ISPD-2019 benchmark testcases using our detailed router with and without the ripup-and-reroute queue enablement described in Section 4.1.6. For the version that is without ripup-and-reroute queue, we use the ripup-and-reroute strategy in [51] while keep every other aspect the same as the version using ripup-and-reroute queue. For this experiment, we set a runtime limit of 24 hours. Table 4.7 gives the wirelength, via count, DRC count and runtime comparisons. We can observe that with the ripup-and-reroute queue, we are able to converge on DRC (i.e., #DRC $\leq$ 50) for all testcases. Moreover, for the testcases that both versions converge on DRC, ripup-and-reroute queue can reduce runtime by an average of 33.5% (up to 85.4%). Since the DR runtime is closely proportional to the overall number of reroutes, the queue-based strategy can achieve DRC convergence more efficiently thanks to its more adaptive control on net ordering.

Table 4.7: Detailed routing comparison of wirelength, via count, DRC count and runtime between queue-based ripup-and-reroute strategy in TritonRoute-WXL (TR-WXL) and non-queue-based ripup-and-reroute strategy in TritonRoute (TR).

Benchmark	**Wirelength** (μm)		**Via count**		**DRC count**		**Runtime** (s)	
	TR-WXL	TR	TR-WXL	TR	TR-WXL	TR	TR-WXL	TR
ispd18_test1	**86440**	86533	**35406**	35466	**0**	0	**23**	33
ispd18_test2	**1572819**	1573641	**359982**	360246	**0**	0	**171**	278
ispd18_test3	**1751762**	1752728	**355758**	356233	**0**	0	**352**	1757
ispd18_test4	**2621560**	2623393	**723918**	725856	**4**	10	**1428**	9762
ispd18_test5	**2763875**	2766182	**889397**	891318	**0**	0	**452**	538
ispd18_test6	**3551801**	3555372	**1369517**	1372596	**0**	0	**683**	875
ispd18_test7	**6475058**	6481683	**2228504**	2235910	**0**	0	**1337**	1479
ispd18_test8	**6503655**	6510428	**2245489**	2252179	**0**	1	**1226**	1454
ispd18_test9	**5433658**	5439825	**2238810**	2244617	**0**	0	**1106**	1528
ispd18_test10	**6760047**	6768788	**2419830**	2432820	**1**	927	**1652**	86400
ispd19_test1	**63151**	63194	**37194**	37246	**0**	1	**84**	93
ispd19_test2	**2470886**	2471332	**787289**	790438	**0**	0	**1053**	1289
ispd19_test3	**82414**	82538	**63852**	64532	**0**	1	**221**	488
ispd19_test4	**3001424**	3007376	**1046033**	1073473	**0**	0	**539**	3121
ispd19_test5	**474240**	474846	**165477**	166581	**0**	0	**55**	64
ispd19_test6	**6537203**	6537793	**1928030**	1930705	3	**2**	**2138**	2934
ispd19_test7	**12157089**	12159501	**4511435**	4516760	**0**	0	**4003**	5324
ispd19_test8	**18694589**	18696221	6980714	**6977429**	**0**	0	**5463**	7125
ispd19_test9	**28280152**	28281276	11581559	**11574769**	**0**	0	**8846**	12167
ispd19_test10	27957631	**27955832**	11711427	**11699849**	**2**	12	**9827**	13842

DR comparison to known best solutions

In this subsection, we compare our detailed routing solution (DR) to the known best DR solutions from all published academic detailed routers based on ISPD-2018 and ISPD-2019 benchmark testcases. We determine the known best DR solutions based on the DRC evaluation result from the official ISPD-2019 contest evaluator, which is more comprehensive as compared to the ISPD-2018 contest evaluator. Therefore, for all ISPD-2018 benchmark testcases, the known best DR solutions are from TritonRoute (TR) [51]. For all ISPD-2019 benchmark testcases, the known best detailed routing solutions are from Dr. CU 2.0 (CU) [54]. Table 4.8 gives the wirelength, via count, DRC count and runtime comparisons. We achieve DRC-clean routing solutions for 16 testcases and reach near-DRC-clean (<5) routing solutions for the remaining testcases. For 19 out of the 20 testcases, we complete detailed routing faster than the known best solution. Overall, we achieve an average of 99.93% (up to 100%) DRC reduction with an average of 30.36% (up to 83.69%) runtime reduction.

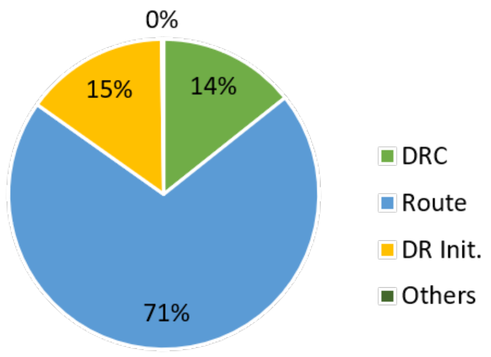


Figure 4.17: Detailed routing runtime breakdown.

We now discuss the runtime and multithread scalability of our current work. For runtime study, Figure 4.17 illustrates the breakdown of the overall detailed routing runtime of four parts – initialization, routing, design rule checking and others. For the multithread scalability study, we measure both single-thread and eight-thread runtime and calculate the eight-thread (8T) speedup. Table 4.8 gives the multithread speedup comparison. We can observe that our work can achieve an average of 5.35× (up to 6.31×) 8T speedup. Note that TritonRoute (TR) [51] does not have multithreading support and Dr. CU 2.0 (CU) [54] has multithreading capability.

Table 4.8: Detailed routing comparison of wirelength, via count, DRC count, runtime and eight-thread (8T) runtime speedup between TritonRoute-WXL (TR-WXL) and known best (K.B.) detailed routing solution. Runtime (s) is obtained with eight threads.

Benchmark	**Wirelength** (μm)		**Via count**		**DRC count**		**Runtime** (s)		**8T speedup** ($\times$)	
	TR-WXL	K.B.	TR-WXL	K.B.	TR-WXL	K.B.	TR-WXL	K.B.	TR-WXL	K.B.
ispd18_test1	86440	**86025**	35406	**32912**	**0**	0	**23**	61	**4.14**	1.00
ispd18_test2	1572819	**1570651**	359982	**319855**	**0**	17	**171**	614	**5.95**	1.00
ispd18_test3	1751762	**1750028**	355758	**319456**	**0**	142	**352**	824	**4.70**	1.00
ispd18_test4	2621560	**2620890**	723918	**695901**	**4**	326	**1428**	1866	**2.77**	1.00
ispd18_test5	2763875	**2763186**	889397	**831775**	**0**	2	**452**	1722	**5.83**	1.00
ispd18_test6	**3551801**	3557744	1369517	**1241673**	**0**	8	**683**	2682	**5.79**	1.00
ispd18_test7	**6475058**	6482066	2228504	**2041794**	**0**	13	**1337**	5023	**5.66**	1.00
ispd18_test8	**6503655**	6513278	2245489	**2062997**	**0**	6	**1226**	4916	**6.25**	1.00
ispd18_test9	**5433658**	5442527	2238810	**2049839**	**0**	5	**1106**	4378	**5.89**	1.00
ispd18_test10	**6760047**	6769942	2419830	**2226243**	**1**	1681	**1652**	10129	**5.02**	1.00
ispd19_test1	**63151**	64258	37194	**36797**	**0**	183	**84**	118	**4.07**	2.91
ispd19_test2	**2470886**	2496133	**787289**	811080	**0**	10475	**1053**	1260	**6.02**	4.52
ispd19_test3	82414	**84216**	**63852**	65501	**0**	667	221	**56**	**3.62**	3.03
ispd19_test4	**3001424**	3049119	1046033	**1031333**	**0**	2612	**539**	1328	**5.34**	3.76
ispd19_test5	**474240**	478046	165477	**153504**	**0**	450	**55**	115	5.00	**5.25**
ispd19_test6	**6537203**	6606659	**1928030**	1998487	**3**	8441	**2138**	2213	**6.11**	5.19
ispd19_test7	**12157089**	12255810	**4511435**	4833913	**0**	32067	**4003**	5288	**6.27**	4.97
ispd19_test8	**18694589**	18847259	**6980714**	7365292	**0**	20213	**5463**	7401	**6.22**	4.86
ispd19_test9	**28280152**	28539077	**11581559**	12249476	**0**	36729	**8846**	10166	**6.31**	4.79
ispd19_test10	**27957631**	28217821	**11711427**	12544541	**2**	36930	**9827**	10665	**5.96**	4.83

GR-based DR convergence study

In this subsection, we compare the detailed routing convergence for all ISPD benchmark testcases. Using our detailed router, we perform detailed routing based on (i) our global routing solutions and (ii) ISPD GR solutions. Table 4.9 shows the detailed routing results from the two sets of global routing solutions. We can observe that compared to the ISPD contest GR solutions, our GR solutions enable faster DR convergence for 16 out of the 20 ISPD testcases while maintaining a similar final DRC count. Note that although our GR solutions yield less wirelength and more via count for most testcases as compared to the ISPD GR solutions, the DR solutions based on our GR solutions achieve (avg. 1.10%) less wirelength and (10.93%) less via count for the four largest testcases. Overall, the faster convergence based on our GR solutions suggests the importance of correlation between global routing and detailed routing. Therefore, using consistent routing data (e.g., pin access location, pin access layer, etc.) is essential to improve global-detailed routing convergence.

Table 4.9: Detailed routing comparison of wirelength, via count, DRC count and runtime of TritonRoute-WXL (TR-WXL) based on TritonRoute-WXL GR solutions and ISPD (ISPD) contest GR solutions.

Benchmark	**Wirelength** (μm)		**Via count**		**DRC count**		**Runtime** (s)	
	TR-WXL	ISPD	TR-WXL	ISPD	TR-WXL	ISPD	TR-WXL	ISPD
ispd18_test1	**85473**	86440	35944	**35406**	**0**	0	**20**	23
ispd18_test2	**1561031**	1572819	368689	**359982**	**0**	0	**152**	171
ispd18_test3	**1748277**	1751762	366529	**355758**	**0**	0	634	**352**
ispd18_test4	**2608384**	2621560	740861	**723918**	**0**	4	**328**	1428
ispd18_test5	**2739911**	2763875	898203	**889397**	**0**	0	**422**	452
ispd18_test6	**3517814**	3551801	1396604	**1369517**	**0**	0	**588**	683
ispd18_test7	**6419424**	6475058	2282448	**2228504**	**0**	0	**1306**	1337
ispd18_test8	**6450911**	6503655	2362445	**2245489**	2	**0**	1379	**1226**
ispd18_test9	**5387783**	5433658	2343351	**2238810**	**0**	0	**1004**	1106
ispd18_test10	6826702	**6760047**	2565965	**2419830**	**0**	1	**1540**	1652
ispd19_test1	**62910**	63151	38524	**37194**	**0**	0	**52**	84
ispd19_test2	**2460555**	2470886	864450	**787289**	2	**0**	**834**	1053
ispd19_test3	**82209**	82414	63958	**63852**	**0**	0	**184**	221
ispd19_test4	3405837	**3001424**	1177000	**1046033**	**0**	0	2002	**539**
ispd19_test5	491124	**474240**	**154285**	165477	**0**	0	74	**55**
ispd19_test6	**6513294**	6537203	2060740	**1928030**	**0**	3	**1795**	2138
ispd19_test7	**12042594**	12157089	**3895912**	4511435	1	**0**	**3768**	4003
ispd19_test8	**18493958**	18694589	**6426055**	6980714	**0**	0	**4474**	5463
ispd19_test9	**27964407**	28280152	**10673809**	11581559	1	**0**	**7205**	8846
ispd19_test10	**27608084**	27957631	**10038232**	11711427	11	**2**	**8639**	9827

GR-DR flow comparison

In this subsection, we compare our global-detailed routing flow to an academic global-detailed routing flow composed of CUGR and Dr. CU 2.0. Table 4.10 shows the global-detailed routing comparison of wirelength, via count, DRC count and runtime between TritonRoute-WXL and CUGR-and-Dr. CU 2.0 flows. We can observe that TritonRoute-WXL consistently achieves considerably lower DRC count with comparable, if not less, wirelength and via count. Meanwhile, for 15 out of the 20 ISPD testcases, TritonRoute-WXL completes routing with shorter runtimes. Overall, TritonRoute-WXL achieves routing solutions with an average of 99.99% (up to 100%) fewer DRCs with similar average wirelength, via count and runtime compared to CUGR-and-Dr. CU 2.0 flow.

Table 4.10: Global-detailed routing comparison of wirelength, via count, DRC count and runtime between TritonRoute-WXL (TR-WXL) flow and CUGR-and-Dr. CU 2.0 (CU) flow.

Benchmark	**Wirelength** (μm)		**Via count**		**DRC count**		**Runtime** (s)	
	TR-WXL	CU	TR-WXL	CU	TR-WXL	CU	TR-WXL	CU
ispd18_test1	**85473**	85737	35944	**35231**	**0**	1554	24	**13**
ispd18_test2	1561031	**1559703**	368689	**365203**	**0**	19207	167	**143**
ispd18_test3	**1748277**	1753358	366529	**361170**	**0**	20718	653	**201**
ispd18_test4	**2608384**	2634776	740861	**727706**	**0**	865	**395**	494
ispd18_test5	**2739911**	2753732	**898203**	927063	**0**	897	**507**	1038
ispd18_test6	**3517814**	3559424	1396604	**1388121**	**0**	720	**671**	785
ispd18_test7	**6419424**	6488953	**2282448**	2289149	**0**	831	**1503**	2114
ispd18_test8	**6450911**	6549767	2362445	**2346013**	**2**	897	**1600**	2057
ispd18_test9	**5387783**	5436327	2343351	**2341125**	**0**	212	**1093**	1416
ispd18_test10	6826702	**6811827**	2565965	**2496257**	**0**	1279	**1730**	2378
ispd19_test1	**62910**	64101	**38524**	40687	**0**	126	**55**	116
ispd19_test2	**2460555**	2500531	864450	**842725**	**2**	9500	**876**	1349
ispd19_test3	**82209**	83901	**63958**	66492	**0**	491	190	**98**
ispd19_test4	3405837	**2994923**	1177000	**917094**	**0**	2677	3756	**3081**
ispd19_test5	491124	**481224**	154285	**138834**	**0**	492	**300**	320
ispd19_test6	**6513294**	6629404	**2060740**	2190998	**0**	3223	**1920**	2180
ispd19_test7	**12042594**	12243117	**3895912**	4073497	**1**	19578	**3987**	5381
ispd19_test8	**18493958**	18721818	**6426055**	6830217	**0**	13463	**4754**	7458
ispd19_test9	**27964407**	28301705	**10673809**	11394780	**1**	27058	**7645**	10290
ispd19_test10	**27608084**	28047248	**10038232**	10331459	**11**	32292	**9181**	10297

Detailed routing a RISC-V core in 14nm

We perform a detailed routing experiment by integrating our detailed router with OpenROAD physical design tool flow [2] in a $14nm$ foundry technology node using a commercial $14nm$ library. We perform our experiment using a global routed RISC-V core [78] (517K instances; runtime 20361 sec). The result confirms that our router is capable of delivering DRC-clean routing result in the sub-$16nm$ commercial context. Figure 4.18 shows the layout of the routed design.

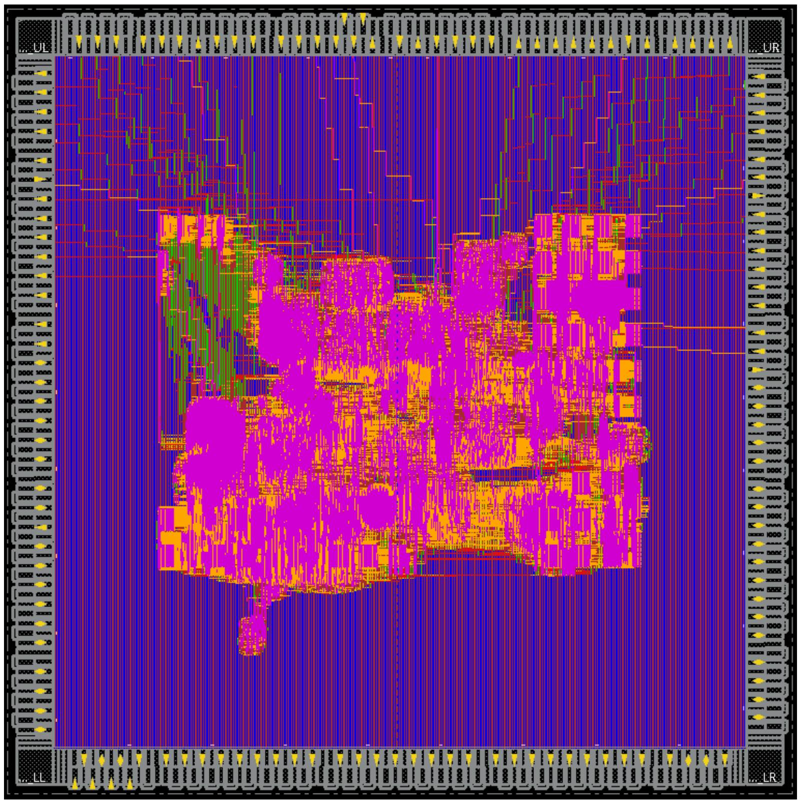


Figure 4.18: Illustration of DRC-clean routing of a RISC-V core in 14nm.

4.1.8 Conclusion

In this work, we present TritonRoute-WXL, an open source router. For detailed routing, with an integrated design rule check engine along with the optimizations enabled by the DRC engine in detailed routing, we deliver DRC-clean detailed routing solutions for 16 of the 20 ISPD contest benchmark testcases. This translates to an average of 99.93% reduction of DRCs as compared to known best detailed routing solutions from all published academic detailed routers, along with an average runtime reduction of 30.36%. For global-detailed routing, compared to the other academic global-detailed routing flow, TritonRoute-WXL achieves an average of 99.99% reduction of DRCs. Our preliminary study also shows that TritonRoute-WXL is capable of delivering DRC-clean routing solutions for sub-16nm foundry technology nodes. Our future research directions include: (i) improving ripup-and-reroute stratetgy for global routing; (ii) timing-driven global-detailed routing; (iii) support of non-default rule (NDR) routing; and (iv) support of ECO routing.

4.2 Acknowledgments

Chapter 4 contains a reprint of Andrew B. Kahng, Lutong Wang and Bangqi Xu, “TritonRoute: The Open Source Detailed Router”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021. Chapter 4 also contains part of the draft of Andrew B. Kahng, Lutong Wang and Bangqi Xu, “TritonRoute-WXL: The Open Source Router with Integrated DRC Engine”, in submission to *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021. The dissertation author is a main contributor to, and a primary author of, each of these papers.

I would like to thank my coauthors Andrew B. Kahng and Lutong Wang for their support and work.

Chapter 5

Conclusion

This thesis has presented innovative detailed placement and routing optimization methodologies, along with essential building blocks towards DRC-clean routing solutions. Together, these address fundamental challenges that arise in advanced technology nodes. The methods presented (i) improve model correlation for placement; (ii) enable key elements for academic routing tools; and (iii) realize an open-source complete routing flow that narrows the academia-industry gap.

Chapter 2 has presented two detailed placement optimization methodologies. First, we have presented an *optimal* dynamic programming-based single-/double-row detailed placement methodology to minimize diffusion *steps* in sub-10nm VLSI, for improved yield and mitigation of NDE. Our work achieves several improvements as compared to previous works: (i) optimal dynamic programming with support of a richer set of cell movements, i.e., flipping, relocating and enhanced reordering; (ii) optimal double-row dynamic programming *with support of movable and reorderable double-height cells*; and (iii) a novel performance improvement technique using *intentional steps*. The proposed techniques achieve up to 98% reduction of inter-cell diffusion *steps*, with scalable runtime and high die utilization in an N7 node enablement. Second, we have presented an *in-route* dynamic programming-based pin access-driven detailed placement optimization methodology to significantly reduce the detailed routing runtime, with noticeable benefits in initial detailed routing DRC count, timing, and routed wirelength. We show that with integration of our *in-route* placement optimization, the detailed routing runtime can be reduced by up to

31.82% (avg. 15.06%) with up to 10.1% (avg. 1.28%) reduction in initial detailed routing DRCs across a wide spectrum of industry designs and technology nodes. Our ongoing research directions include (i) DRC-driven detailed placement refinement; and (ii) a more comprehensive timing-aware optimization flow considering different cell timing criticality characteristics.

Chapter 3 has presented two essential building blocks towards DRC-clean routing. First, we have presented a new geometry-based design rule checking for detailed routing. Our methodology supports design rule checking in the technology nodes from ISPD initial detailed rouitng contests, and is capable of identifying detailed-routing-fixable violations. We integrate our design rule checking into the open-source TritonRoute. We show up to 100% (avg. 93.54%) DRC reduction and up to 50.28% (avg. 17.84%) detailed routing runtime reduction when the feedback from the DRC engine is considered. Our future research directions include: (i) support of color-aware design rule checking and (ii) DRC engine-based design rule violation fixing for shape-related violations (e.g., minimum step violation). Second, we have presented a multi-level, standard cell- and instance-based, complete, robust, scalable and design rule-aware pin access analysis framework. We describe our robust pin-based access point generation, boundary conflict-aware access pattern generation and cluster-based access pattern selection based on dynamic programming. We achieve 100% DRC-clean pin access and demonstrate a superior final detailed routing solution as compared to the best known results using the ISPD-2018 initial detailed routing contest benchmark suite. Our future research directions include: (i) inter-cluster and inter-row pin access co-optimization for better access point alignment; and (ii) incremental pin access analysis for application in engineering change order (ECO) contexts.

Chapter 4 has presented an end-to-end routing flow. We have presented TritonRoute-WXL, an open source router. For detailed routing, with an integrated design rule check engine along with the optimizations enabled by the DRC engine in detailed routing, we deliver DRC-clean detailed routing solutions for 16 of the 20 ISPD contest benchmark testcases. This translates to an average of 99.93% reduction of DRCs as compared to known best detailed routing solutions from all published academic detailed routers, along with an average runtime reduction of 30.36%. For global-detailed routing, compared to the other academic global-detailed routing flow, TritonRoute-WXL achieves an average of 99.99% reduction of DRCs. Our preliminary study also shows that TritonRoute-WXL is capable of delivering

DRC-clean routing solutions for sub-16nm foundry technology nodes. Our future research directions include: (i) improving ripup-and-reroute stratetgy for global routing; (ii) timing-driven global-detailed routing; (iii) support of non-default rule (NDR) routing; and (iv) support of ECO routing.

As the complexity of design rules and the difficulty of designs themselves increase, the conventional EDA tool flow with separate and isolated tools can no longer meet design productivity and design quality needs. Today, we already observe some look-ahead capabilities (e.g., congestion evaluation using global routing during placement stage) in modern tool flows. However, we are still in the early stage of co-optimizations between different tools. Moreover, with the increasing complexity of design rules, potential improvements in automated tool flows are worth studying. Therefore, looking beyond this thesis, our future research directions to address looming challenges include the following:

- **Iterative global-detailed routing**. With rapid development in advanced technology nodes, existing routability models in the global router may not be able to capture new aspects of routability that arise from new design rules. In order to make the global-detailed routing more robust, iteration between global routing and detailed routing can help resolve such routability issues.

- **Iterative place and route**. In-route placement refinement for detailed routing convergence has been presented above. Another important use case would entail in-route placement refinement for resolving last-mile design rule violations. Last-mile design rule violations may involve aspects of pin accessibility issues or pin access-induced issues that are difficult for the pin accessibility model to capture. Having the capability to refine the placement during routing can potentially resolve the last-mile violations more efficiently.

- **Routing-centric design optimization**. Conventional design optimization, including buffering and sizing, mostly considers the available placement whitespace to evaluate feasibility of buffering and sizing. However, searching potential locations for buffering and sizing based on placement whitespace may lead to congestion in routing, which potentially degrades timing and power. Moreover, evaluating the impact on routing of a given candidate buffering or sizing move can be expensive. Therefore, a more efficient and comprehensive, routing-centric way of determining buffering and sizing locations can be considered. For example, locations in a design with low congestion usually indicate placement whitespace as well; this can serve as a “sufficient” condition for determining candidate locations for buffering and sizing. Hence, a router with the capability of suggesting buffering and sizing locations with timing awareness may improve design convergence considerably.

Bibliography

[1] M. Ahrens, M. Gester, N. Klewinghaus, D. Müller, S. Peyer, C. Schulte and G. Téllez, "Detailed Routing Algorithms for Advanced Technology Nodes", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34(4) (2015), pp. 563-576.

[2] T. Ajayi, V. A. Chhabria, M. Fogaca, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee, U. Mallappa, M. Neseem, G. Pradipta, S. Reda, M. Saligane, S. S. Sapatnekar, C. Sechen, M. Shalan, W. Swartz, L. Wang, Z. Wang, M. Woo and B. Xu, "Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project", *Proc. ACM/IEEE Design Automation Conference*, 2019, pp. 76:1-76:4.

[3] C. J. Alpert, Z. Li, M. D. Moffitt, G.-J. Nam, J. A. Roy and G. Tellez, "What Makes a Design Difficult to Route", *Proc. ACM International Symposium on Physical Design*, 2010, pp. 7-12.

[4] C. J. Alpert, D. P. Mehta and S. S. Sapatnekar (Eds.), *Handbook of Algorithms for Physical Design Automation*, Boca Raton, Auerbach Publications, 2009.

[5] W.-T. J. Chan, P.-H. Ho, A. B. Kahng and P. Saxena, "Routability Optimization for Industrial Designs at Sub-14nm Process Nodes Using Machine Learning", *Proc. ACM International Symposium on Physical Design*, 2017, pp. 15-21.

[6] Y.-J. Chang, Y.-T. Lee and T.-C. Wang, "NTHU-Route 2.0: A Fast and Stable Global Router", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2008, pp. 338-343.

[7] F.-Y. Chang, R.-S. Tsay, W.-K. Mak and S.-H. Chen, "MANA: A Shortest Path Maze Algorithm Under Separation and Minimum Length NAnometer Rules", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32(10) (2013), pp. 1557-1568.

[8] H.-Y. Chen and Y.-W. Chang, "Global and Detailed Routing", Chapter 12 in Wang, Chang and Cheng (Eds.), *Electronic Design Automation: Synthesis, Verification, and Test*, Morgan Kaufmann, 2009, pp. 687-749. http://cc.ee.ntu.edu.tw/~ywchang/Courses/PD_Source/EDA_routing.pdf

[9] H.-Y. Chen, C.-H. Hsu and Y.-W. Chang, "High-Performance Global Routing with Fast Overflow Reduction", *Proc. Asia and South Pacific Design Automation Conference*, 2009, pp. 582-587.

[10] J. Chen, J. Kuang, G. Zhao, D. J.-H. Huang and E. F. Y. Young, "PROS: A Plug-In for Routability Optimization Applied in The State-of-The-Art Commercial EDA Tool using Deep Learning", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2020, pp. 17:1-17:8.

[11] D. C. Chen, G. S. Lin, T. H. Lee, R. Lee, Y. C. Liu, M. F. Wang, Y. C. Cheng and D. Y. Wu, "Compact Modeling Solution of Layout Dependent Effect for FinFET Technology", *Proc. International Conference on Microelectronics Test Structures*, 2015, pp. 110-115.

[12] G. Chen, C.-W. Pui, H. Li, J. Chen, B. Jiang and E. F. Y. Young, "Detailed Routing by Sparse Grid Graph and Minimum-Area-Captured Path Search", *Proc. Asia and South Pacific Design Automation Conference*, 2019, pp. 754-760.

[13] G. Chen, C.-W. Pui, H. Li and E. F. Y. Young, "Dr. CU: Detailed Routing by Sparse Grid Graph and Minimum-Area-Captured Path Search", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39(9) (2020), pp. 1902-1915.

[14] W.-K. Chow, J. Kuang, X. He, W. Cai and E. F. Y. Young, "Cell Density-Driven Detailed Placement with Displacement Constraint", *Proc. ACM International Symposium on Physical Design*, 2014, pp. 3-10.

[15] C. Chu and Y. Wong, "FLUTE: Fast Lookup Table Based Rectilinear Steiner Minimal Tree Algorithm for VLSI Design", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27(1) (2008), pp. 70-83.

[16] K.-R. Dai, W.-H. Liu and Y.-L. Li, "NCTU-GR: Efficient Simulated Evolution-Based Rerouting and Congestion-Relaxed Layer Assignment on 3-D Global Routing", *IEEE Transactions on Very Large Scale Integration Systems* 20(3) (2012), pp. 459-472.

[17] N. K. Darav, I. S. Bustany, A. Kennings and R. Mamidi, "ICCAD-2017 CAD Contest in Multi-Deck Standard Cell Legalization and Benchmarks", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2017, pp. 867-871.

[18] L. R. Darden, "Spacing Violation Checker", *US Patent* 6301689 B1, September 1998.

[19] L. R. Darden and W. J. Livingstone, "Methods for Design Rule Checking with Abstracted Via Obstructions", *US Patent* 7725850 B2, July 2007.

[20] P. Debacker, K. Han, A. B. Kahng, H. Lee, P. Raghavan and L. Wang, "Vertical M1 Routing-Aware Detailed Placement for Congestion and Wirelength Reduction in Sub-10nm Nodes", *Proc. ACM/IEEE Design Automation Conference*, 2017, pp. 51:1-51:6.

[21] Y. Ding, C. Chu and W.-K. Mak, "Self-Aligned Double Patterning Lithography Aware Detailed Routing with Color Preassignment", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36(8) (2017), pp. 1381-1394.

[22] Y. Ding, C. Chu and W.-K. Mak, "Pin Accessibility-Driven Detailed Placement Refinement", *Proc. ACM International Symposium on Physical Design*, 2017, pp. 133-140.

[23] S. Dobre, A. B. Kahng and J. Li, "Mixed Cell-Height Implementation for Improved Design Quality in Advanced Nodes", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2015, pp. 854-860.

[24] S. Dolgov, A. Volkov, L. Wang and B. Xu, "2019 CAD Contest: LEF/DEF Based Global Routing", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2019, pp. 1-4.

[25] Y. Du, Q. Ma, H. Song, J. Shiely, G. Luk-Pat, A. Miloslavsky and M. D. F. Wong, "Spacer-is-Dielectric-Compliant Detailed Routing for Self-Aligned Double Patterning Lithography", *Proc. ACM/EDAC/IEEE Design Automation Conference*, 2013, pp. 1-6.

[26] Y. Du and M. D. F. Wong, "Optimization of Standard Cell Based Detailed Placement for 16nm FinFET Process", *Proc. Design, Automation and Test in Europe*, 2014, pp. 1-6.

[27] J. V. Faricelli, "Layout-Dependent Proximity Effects in Deep Nanoscale CMOS", *Proc. IEEE Custom Integrated Circuits Conference*, 2010, pp. 1-8.

[28] A. Feller, "Automatic Layout of Low-Cost Quick-Turnaround Random-Logic Custom LSI Devices", *Proc. ACM/IEEE Design Automation Conference*, 1976, pp. 79-85.

[29] G.-R. Gao and D. Z. Pan, "Flexible Self-Aligned Double Patterning Aware Detailed Routing with Prescribed Layout Planning", *Proc. ACM International Symposium on Physical Design*, 2012, pp. 25-32.

[30] V. C. Gerousis, S. Zhang, J. Li, S. Mantik and L. Tsai, "Method and System for Implementing Efficient Trim Data Representation for an Electronic Design", *US Patent* 10192018 B1, March 2016.

[31] M. Gester, D. Muller, T. Nieberg, C. Panten, C. Schulte and J. Vygen, "BonnRoute: Algorithms and Data Structures for Fast and Good VLSI Routing", *ACM Transactions on Design Automation of Electronic Systems* 18(2) (2013), pp. 32:1-32:24.

[32] S. M. M. Gonçalves, L. S. da Rosa and F. de S. Marques, "DRAPS: A Design Rule Aware Path Search Algorithm for Detailed Routing", *IEEE Transactions on Circuits and Systems II: Express Briefs* 67(7) (2019), pp. 1239-1243.

[33] S. M. M. Gonçalves, L. S. da Rosa and F. de S. Marques, "An Improved Heuristic Function for A*-Based Path Search in Detailed Routing", *Proc. IEEE International Symposium on Circuits and Systems*, 2019, pp. 1-5.

[34] S. M. M. Gonçalves, L. S. da Rosa and F. de S. Marques, "SmartDR: Algorithms and Techniques for Fast Detailed Routing with Good Design Rule Handling", *ACM Transactions on Design Automation of Electronic Systems* 26(2) (2020), pp. 9:1-9:38.

[35] F. O. Hadlock, "A Shortest Path Algorithm for Grid Graphs", *Networks* 7(4) (1977), pp. 323-334.

[36] C. Han, K. Han, A. B. Kahng, H. Lee, L. Wang and B. Xu, "Optimal Multi-Row Detailed Placement for Yield and Model-Hardware Correlation Improvements in Sub-10nm VLSI", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2017, pp. 667-674.

[37] K. Han, A. B. Kahng and H. Lee, "Evaluation of BEOL Design Rule Impacts Using an Optimal ILP-Based Detailed Router", *Proc. ACM/EDAC/IEEE Design Automation Conference*, 2015, pp. 68:1-68:6.

[38] K. Han, A. B. Kahng and H. Lee, "Scalable Detailed Placement Legalization for Complex Sub-14nm Constraints", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2015, pp. 867-873.

[39] C. Han, A. B. Kahng, L. Wang and B. Xu, "Enhanced Optimal Multi-Row Detailed Placement for Neighbor Diffusion Effect Mitigation in Sub-10nm VLSI", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38(9) (2019), pp. 1703-1716.

[40] A. Hetzel, "A Sequential Detailed Router for Huge Grid Graphs", *Proc. Design, Automation and Test in Europe*, 1998, pp. 332-339.

[41] D. W. Hightower, "A Solution to Line-Routing Problems on the Continuous Plane", *Proc. ACM/IEEE Design Automation Conference*, 1969, pp. 1-24.

[42] D. Hill, "Method and System for High Speed Detailed Placement of Cells Within an Integrated Circuit Design", *US Patent* No. US6370673B1, April 2002.

[43] K.-S. Hu and M.-J. Yang, "Problem B: Routing with Cell Movement", *ICCAD-2020 CAD Contest*, http://iccad-contest.org/2020/

[44] S.-W. Hur and J. Lillis, "Mongrel: Hybrid Techniques for Standard Cell Placement", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2000, pp. 165-170.

[45] A. B. Kahng, "The ITRS Design Technology and System Drivers Roadmap: Process and Status", *Proc. ACM/EDAC/IEEE Design Automation Conference*, 2013, pp. 34-39.

[46] A. B. Kahng, I. L. Markov and S. Reda, "On Legalization of Row-Based Placements", *Proc. ACM Great Lakes Symposium on Very Large Scale Integration*, 2004, pp. 214-219.

[47] A. B. Kahng, P. Sharma and R. O. Topaloglu, "Exploiting STI Stress for Performance", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2007, pp. 83-90.

[48] A. B. Kahng, P. Tucker and A. Zelikovsky, "Optimization of Linear Placements for Wirelength Minimization with Free Sites", *Proc. Asia and South Pacific Design Automation Conference*, 1999, pp. 241-244.

[49] A. B. Kahng, L. Wang and B. Xu, "TritonRoute: An Initial Detailed Router for Advanced VLSI Technologies", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2018, pp. 81:1-81:8.

[50] A. B. Kahng, L. Wang and B. Xu, "The Tao of PAO: Anatomy of a Pin Access Oracle for Detailed Routing", *Proc. ACM/IEEE Design Automation Conference*, 2020, pp. 1-6.

[51] A. B. Kahng, L. Wang and B. Xu, "TritonRoute: The Open Source Detailed Router", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40(3) (2021), pp. 547-559.

[52] C. Y. Lee, "An Algorithm for Path Connections and Its Applications", *IRE Transactions on Electronic Computers* 10(3) (1961), pp. 346-365.

[53] H. K.-S. Leung, "Advanced Routing in Changing Technology Landscape", *Proc. ACM International Symposium on Physical Design*, 2003, pp. 118-121.

[54] H. Li, G. Chen, B. Jiang, J. Chen and E. F. Y. Young, "Dr. CU 2.0: A Scalable Detailed Routing Framework with Correct-by-Construction Design Rule Satisfaction", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2019, pp. 1-7.

[55] S. Li and C.-K. Koh, "Mixed Integer Programming Models for Detailed Placement", *Proc. ACM International Symposium on Physical Design*, 2012, pp. 87-94.

[56] Y. Lin, B. Yu, X. Xu, J.-R. Gao, N. Viswanathan, W.-H. Liu, Z. Li, C. J. Alpert and D. Z. Pan, "MrDP: Multiple-row Detailed Placement of Heterogeneous-sized Cells for Advanced Nodes", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2016, pp. 1-8.

[57] Y. Lin, B. Yu, B. Xu and D. Z. Pan, "Triple Patterning Aware Detailed Placement Toward Zero Cross-Row Middle-of-Line Conflict", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2015, pp. 396-403.

[58] I.-J. Liu, S.-Y. Fang and Y.-W. Chang, "Overlay-Aware Detailed Routing for Self-Aligned Double Patterning Lithography Using the Cut Process", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35(9) (2016), pp. 1519-1531.

[59] W.-H. Liu, C.-K. Koh and Y.-L. Li, "Optimization of Placement Solutions for Routability", *Proc. ACM/EDAC/IEEE Design Automation Conference*, 2013, pp. 153:1-153:9.

[60] W.-H. Liu, W.-C. Kao, Y.-L. Li and K.-Y. Chao, "NCTU-GR 2.0: Multithreaded Collision-Aware Global Routing with Bounded-Length Maze Routing", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32(5) 2013, pp. 709-722.

[61] W.-H. Liu, S. Mantik, W.-K. Chow, Y. Ding, A. Farshidi and G. Posser, "ISPD 2019 Initial Detailed Routing Contest and Benchmark with Advanced Routing Rules", *Proc. ACM International Symposium on Physical Design*, 2018, pp. 140-143.

[62] J. Liu, C.-W. Pui, F. Wang and E. F. Y. Young, "CUGR: Detailed-Routability-Driven 3D Global Routing with Probabilistic Resource Model", *Proc. ACM/IEEE Design Automation Conference*, 2020, pp. 1-6.

[63] W. K. Luk, "A Greedy Switch-Box Router", *Integration* 3(2) (1985), pp. 129-149.

[64] S. Mantik and V. C. Gerousis, "Method, System, and Computer Program Product for Improving Mask Designs and Manufacturability of Electronic Designs for Multi-Exposure Lithography", *US Patent* 9767245 B1, June 2014.

[65] S. Mantik, L. He, S. Kim, J. Lam and J. Li, "Computationally Efficient Design Rule Checking for Circuit Interconnect Routing Design", *US Patent* 7594207 B2, September 2006.

[66] S. Mantik, G. Posser, W.-K. Chow, Y. Ding and W.-H. Liu, "ISPD 2018 Initial Detailed Routing Contest and Benchmarks", *Proc. ACM International Symposium on Physical Design*, 2018, pp. 140-143.

[67] L. McMurchie and C. Ebeling, "Pathfinder: A Negotiation-Based Performance-Driven Router for FPGAs", *Proc. International Symposium on Field-Programmable Gate Arrays*, 1995, pp. 111-117.

[68] Model-Hardware Correlation Team, *Samsung Electronics Co., Ltd.*, November 2016.

[69] M. B. S. Mohan and K. E. Moynihan, "Method and System for Hierarchical Metal-End, Enclosure and Exposure Checking", *US Patent* 6536023 B1, July 2000.

[70] G.-J. Nam, C. Sze and M. Yildiz, “The ISPD Global Routing Benchmark Suite”, *Proc. ACM International Symposium on Physical Design*, 2008, pp. 156-159.

[71] G.-J. Nam, M. Yildiz, D. Z. Pan and P. H. Madden, “ISPD Placement Contest Updates and ISPD 2007 Global Routing Contest”, *Proc. ACM International Symposium on Physical Design*, 2007, p. 167.

[72] T. Nieberg, “Gridless Pin Access in Detailed Routing”, *Proc. ACM/EDAC/IEEE Design Automation Conference*, 2011, pp. 170-175.

[73] N. J. Nilsson, “State-Space Search Methods”, in *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill Book Co., 1971, pp. 43-79.

[74] S.-K. Oh, S.-H. Baek, S.-Y. Lee and T.-J. Song, “Standard Cell Library, Method of Using the Same, and Method of Designing Semiconductor Integrated Circuit”, *US Patent* No. US9830415B2, November 2017.

[75] H.-C. Ou, K.-H. Tseng, J.-Y. Liu, I.-P. Wu and Y.-W. Chang, “Layout-Dependent-Effects-Aware Analytical Analog Placement”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35(8) (2016), pp. 1243-1254.

[76] M. M. Ozdal, “Detailed-Routing Algorithms for Dense Pin Clusters in Integrated Circuits”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28(3) (2009), pp. 340-349.

[77] M. Pan, N. Viswanathan and C. Chu, “An Efficient and Effective Detailed Placement Algorithm”, *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2005, pp. 48-55.

[78] D. Petrisko, F. Gilani, M. Wyse, D. C. Jung, S. Davidson, P. Gao, C. Zhao, Z. Azad, S. Canakci, B. Veluri, T. Guarino, A. J. Joshi, M. Oskin, M. B. Taylor, “BlackParrot: An Agile Open Source RISC-V Multicore for Accelerator SoCs”, *IEEE Micro Special Issue on Agile and Open-Source Hardware* 40(4) 2020, pp. 93-102.

[79] I. Pohl, “Bi-Directional Search”, *Machine Intelligence* (1971), pp. 127-140.

[80] B. T. Preas, M. J. Lorenzetti and B. D. Ackland, *Physical Design Automation of VLSI Systems*, Menlo Park, Benjamin/Cummings, 1988.

[81] Z. Qi, Y. Cai and Q. Zhou, “Accurate Prediction of Detailed Routing Congestion using Supervised Data Learning”, *Proc. IEEE International Conference on Computer Design*, 2014, pp. 97-103.

[82] X. Qiu and M. Marek-Sadowska, “Can Pin Access Limit the Footprint Scaling?”, *Proc. ACM/EDAC/IEEE Design Automation Conference*, 2012, pp. 1100-1106.

[83] J. A. Roy and I. L. Markov, “High-Performance Routing at the Nanometer Scale”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27(6) 2008, pp. 1066-1077.

[84] L. Scheffer, L. Lavagno and G. Martin (Eds.), *EDA for IC System Design, Verification, and Testing*, Boca Raton, CRC Press, 2006.

[85] J. Seo, J. Jung, S. Kim and Y. Shin, “Pin Accessibility-Driven Cell Layout Redesign and Placement Optimization”, *Proc. ACM/EDAC/IEEE Design Automation Conference*, 2017, pp. 54:1-54:6.

[86] N. Sherwani, *Algorithms for VLSI Physical Design Automation*, 3rd edition, New York, Springer, 1999.

[87] E. Shragowitz and S. Keel, "A Global Router Based on a Multicommodity Flow Model", *Integration* 5(1) 1987, pp. 3-16.

[88] J. Soukup, "Fast Maze Router", *Proc. ACM/IEEE Design Automation Conference*, 1978, pp. 100-102.

[89] F.-K. Sun, H. Chen, C.-Y. Chen, C.-H. Hsu and Y.-W. Chang, "A Multithreaded Initial Detailed Routing Algorithm Considering Global Routing Guides", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2018, pp. 82:1-82:7.

[90] T. Taghavi, C. Alpert, A. Huber, Z. Li, G.-J. Nam and S. Ramji, "New Placement Prediction and Mitigation Techniques for Local Routing Congestion", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2010, pp. 621-624.

[91] M. Tarabbia, A. Mittal and N. Hindawy, "Forming FinFET Cell with Fin Tip and Resulting Device", *US Patent* No. US9059093B2, June 2015.

[92] H. Tian, Y. Du, H. Zhang, Z. Xiao and M. D. F. Wong, "Triple Patterning Aware Detailed Placement with Constrained Pattern Assignment", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2014, pp. 116-123.

[93] P.-S. Tzeng, and C. H. Sequin, "Codar: A Congestion-Directed General Area Router", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 1988, pp. 30-33.

[94] C.-H. Wang, Y.-Y. Wu, J. Chen, Y.-W. Chang, S.-Y. Kuo, W. Zhu and G. Fan, "An Effective Legalization Algorithm for Mixed-Cell-Height Standard Cells", *Proc. Asia and South Pacific Design Automation Conference*, 2017, pp. 450-455.

[95] L. Wen and T. Gao, "Generating and Using Route Fix Guidance", *US Patent* 8527930 B2, January 2010.

[96] M.-P. Wong, W.-H. Liu and T.-C. Wang, "Negotiation-Based Track Assignment Considering Local Nets", *Proc. Asia and South Pacific Design Automation Conference*, 2016, pp. 378-383.

[97] G. Wu and C. Chu, "Detailed Placement Algorithm for VLSI Design with Double-Row Height Standard Cells", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35(8) (2016), pp. 1569-1573.

[98] T.-H. Wu, A. Davoodi and J. T. Linderoth, "GRIP: Scalable 3D Global Routing using Integer Programming", *Proc. ACM/IEEE Design Automation Conference*, 2009, pp. 320-325.

[99] Z. Xie, Y.-H. Huang, G.-Q. Fang, H. Ren, S.-Y. Fang, Y. Chen and J. Hu, "RouteNet: Routability Prediction for Mixed-Size Designs using Convolutional Neural Network", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2018, pp. 1-8.

[100] R. Xie, K.-Y. Lim, M. G. Sung and R. R.-H. Kim, "Methods of Forming Single and Double Diffusion Breaks on Integrated Circuit Products Comprised of FinFET Devices and The Resulting Products", *US Patent* No. US9412616B1, August 2016.

[101] Y. Xu and C. Chu, "MGR: Multi-Level Global Router", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2011, pp. 250-255.

[102] X. Xu, B. Cline, G. Yeric, B. Yu and D. Z. Pan, "Self-Aligned Double Pattering Aware Pin Access and Standard Cell Layout Co-Optimization", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34(5) (2015), pp. 699-712.

[103] X. Xu, Y. Lin, V. Livramento and D. Z. Pan, "Concurrent Pin Access Optimization for Unidirectional Routing", *Proc. ACM/IEEE Design Automation Conference*, 2017, pp. 20:1-20:6.

[104] X. Xu, B. Yu, J.-R. Gao, C.-L. Hsu and D. Z. Pan, "PARR: Pin Access Planning and Regular Routing for Self-Aligned Double Patterning", *ACM Transactions on Design Automation of Electronic Systems* 21(3) (2016), pp. 42:1-42:21.

[105] Y. Xu, Y. Zhang and C. Chu, "FastRoute 4.0: Global Router with Efficient Via Minimization", *Proc. Asia and South Pacific Design Automation Conference*, 2009, pp. 576-581.

[106] S. Yang, Y. Liu, M. Cai, J. Bao, P. Feng, X. Chen, L. Ge, J. Yuan, J. Choi, P. Liu, Y. Suh, H. Wang, J. Deng, Y. Gao, J. Yang, X.-Y. Wang, D. Yang, J. Zhu, P. Penzes, SC Song, C. Park, S. Kim, J. Kim, S. Kang, E. Terzioglu, K. Rim and PR. C. Chidambaram, "10nm High Performance Mobile SoC Design and Technology Co-Developed for Performance, Power and Area Scaling", *Proc. Symposium on VLSI Technology*, 2017, pp. T70-T71.

[107] W. Ye, B. Yu, D. Z. Pan, Y.-C. Ban and L. Liebmann, "Standard Cell Layout Regularity and Pin Access Optimization Considering Middle-of-Line", *Proc. ACM Great Lakes Symposium on Very Large Scale Integration*, 2015, pp. 289-294.

[108] T.-C. Yu, S.-Y. Fang, H.-S. Chiu, K.-S. Hu, P. H.-Y. Tai, C. C.-F. Shen and H. Sheng, "Pin Accessibility Prediction and Optimization with Deep Learning-based Pin Pattern Recognition", *Proc. ACM/IEEE Design Automation Conference*, 2019, pp. 220:1-220:6.

[109] B. Yu, X. Xu, J.-R. Gao, Y. Lin, Z. Lee, C. J. Alpert and D. Z. Pan, "Methodology for Standard Cell Compliance and Detailed Placement for Triple Patterning Lithography", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34(5) (2015), pp. 726-739.

[110] Y. Zhang and C. Chu, "CROP: Fast and Effective Congestion Refinement of Placement", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 2009, pp. 344-350.

[111] Y. Zhang and C. Chu, "RegularRoute: An Efficient Detailed Router Applying Regular Routing Patterns", *IEEE Transactions on Very Large Scale Integration Systems* 21(9) (2013), pp. 1655-1668.

[112] P. Zhao, S. M. Pandey, E. Banghart, X. He, R. Asra, V. Mahajan, H. Zhang, B. Zhu, K. Yamada, L. Cao, P. Balasubramaniam, M. Joshi, M. Eller, F. Benistant and S. Samavedam, "Influence of Stress Induced CT Local Layout Effect (LLE) on 14nm FinFET", *Proc. Symposium on VLSI Technology*, 2017, pp. T228-T229.

[113] Q. Zhou, X. Wang, Z. Qi, Z. Chen, Q. Zhou and Y. Cai, "An Accurate Detailed Routing Routability Prediction Model in Placement", *Proc. Asia Symposium on Quality Electronic Design*, 2015, pp. 119-122.

[114] Cadence Innovus Implementation System. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/soc-implementation-and-floorplanning/innovus-implementation-system.html

[115] Dr. CU 2.0, https://github.com/cuhk-eda/dr-cu/releases/tag/v4.1.1

[116] International Technology Roadmap for Semiconductors. http://www.itrs2.net/itrs-reports.html

[117] LEF/DEF 5.7 reference. https://si2.org/oa-tools-utils-libs/

[118] LEF/DEF Language Reference. http://www.ispd.cc/contests/18/lefdefref.pdf

[119] B. Schäling, *The Boost C++ Libraries, 2nd ed.*, XML Press, 2014.

[120] W.-H. Liu, "ISPD 2018 Initial Detailed Routing Contest and Benchmarks" *presentation slides*, http://www.ispd.cc/slides/2018/s7_3.pdf

[121] OpenCores: Open Source IP-Cores. http://www.opencores.org

[122] OpenMP Architecture Review Board, "OpenMP Application Program Interface, Version 4.0".

[123] TritonRoute-WXL: The Open Source Router with Integrated DRC Engine. https://www.github.com/ABKGroup/TritonRoute-WXL

[124] The-OpenROAD-Project/TritonRoute: UCSD Detailed Router, https://github.com/The-OpenROAD-Project/TritonRoute

[125] TritonRoute Version 0.0.6.0, https://github.com/The-OpenROAD-Project/TritonRoute/releases/tag/0.0.6.0

[126] Si2 OpenAccess. https://si2.org/openaccess/

[127] Synopsys Design Compiler. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html