Dr. CU: Detailed Routing by Sparse Grid Graph and Minimum-Area-Captured Path Search

Gengjie Chen, Chak-Wa Pui, Haocheng Li, and Evangeline F. Y. Young

***Abstract*—Different from global routing, detailed routing takes care of many detailed design rules and is performed on a significantly larger routing grid graph. In advanced technology nodes, it becomes the most complicated and time-consuming stage in the very large-scale integration physical design flow. We propose Dr. CU, an efficient and effective detailed router, to tackle the challenges. To handle a 3-D detailed routing grid graph of enormous size, a set of two-level sparse data structures is designed for runtime and memory efficiency. For handling the minimum-area constraint, an optimal correct-by-construction path search algorithm is proposed. Besides, an efficient bulk synchronous parallel scheme is adopted to further reduce the runtime usage. Compared with the other state-of-the-art academic detailed routers, Dr. CU reduces the number of design rule violations by one or two orders of magnitude. At the same time, it uses shorter wire length, fewer vias, and significantly less runtime. The source code of Dr. CU is available at https://github.com/cuhk-eda/dr-cu.**

***Index Terms*—Detailed routing, interconnect, physical design, rip up and reroute.**

I. Introduction

BECAUSE of its enormous computational complexity, very large-scale integration (VLSI) routing is usually performed in two stages: 1) global and 2) detailed. In the *global routing* stage, the routing space is split into an array of regular cells, where a coarse-grained routing solution is generated. It optimizes wire length, via count, routability, timing, and other metrics with a global view. *Detailed routing*, on the other hand, realizes the global routing solution by considering exact metal shapes and positions. It takes care of many complicated detailed design rules [e.g., parallel-run spacing, end-of-line (EOL) spacing, cut spacing, minimum area, etc.]. Its solution quality directly influences various eventual design metrics, such as timing, signal integrity, and chip yield [2]. Meanwhile, its solution space, a 3-D grid graph, is significantly larger than that of global routing. In advanced technology nodes, detailed routing becomes the most complicated and time-consuming stage [3].

During the past decade, many approaches were proposed to complete fast and high-quality global routing with a sustaining progress (e.g., FGR [4], FastRoute [5], BoxRouter [6], Ancher [7], GRIP [8], BonnRoute [9], and NCTU-GR [10]). However, there is insufficient effort for exploring efficient and effective detailed routers in academia. RegularRoute [2] encourages regular routing patterns and exploits a maximum independent set formulation for better design rule satisfaction. MANA [11] considers EOL spacing and minimum length of a wire segment in maze routing. The work in [12] presents the data structures and algorithms for detailed routing used in BonnRoute. Besides, several specific issues in detailed routing have been discussed. For example, methods for the pin access optimization are proposed in [13]–[15]. For others, the impact of various manufacturing technologies have been dealt with, including triple patterning [16]–[18], self-aligned doubling patterning [19], and directed self-assembly [20].

Recently, the ISPD 2018 Initial Detailed Routing Contest [3] stimulates several works on detailed routing. Kahng *et al.* [21] proposed TritonRoute, a detailed router with integer linear programming (ILP)-based intralayer parallel routing. Sun *et al.* [22] presented a detailed routing algorithm with a multistage, rip-up, and reroute scheme. Their approaches suffer from the weakness in both design rule satisfaction and runtime scalability.

As the feature size scales down, not only the problem size but also the complexity of design rules of detailed routing becomes increasingly challenging. Moreover, many detailed routers heavily rely on post processing for fixing design rule violations. Design rule dimensions, however, do not scale well with feature miniaturization (e.g., feature size decreases much faster than minimum area values) and require relatively more spaces for fixing. In this way, a post processing step fails more frequently [12]. Therefore, we proposes Dr. CU, a detailed routing framework that is superiorly scalable in runtime as well as memory usage and provides more correct-by-construction design rule satisfaction. Our contributions can be summarized as follows.

1) We design a set of two-level sparse data structures for a 3-D detailed routing grid graph of enormous size.
2) We develop an optimal correct-by-construction path search that captures the minimum-area constraint.

Manuscript received February 12, 2019; revised May 21, 2019; accepted June 26, 2019. Date of publication July 9, 2019; date of current version August 20, 2020. This work was supported in part by the Research Grants Council of the Hong Kong Special Administrative Region, China, under Project CUHK14208914, and in part by the Hong Kong Ph.D. Fellowship Scheme. The preliminary version has been presented at the Asia and South Pacific Design Automation Conference (ASP-DAC) in 2019 [1]. This paper was recommended by Associate Editor I. H.-R. Jiang. *(Corresponding author: Gengjie Chen.)*

The authors are with the Department of Computer Science and Engineering, Chinese University of Hong Kong, Hong Kong (e-mail: gjchen@cse.cuhk.edu.hk).

Digital Object Identifier 10.1109/TCAD.2019.2927542

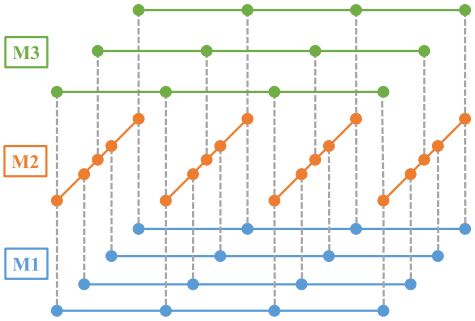

Fig. 1. Example 3-D detailed routing grid graph. In this example, preferred directions of metal 1 (M1) and M3 layers are both horizontal, while that of M2 is vertical.

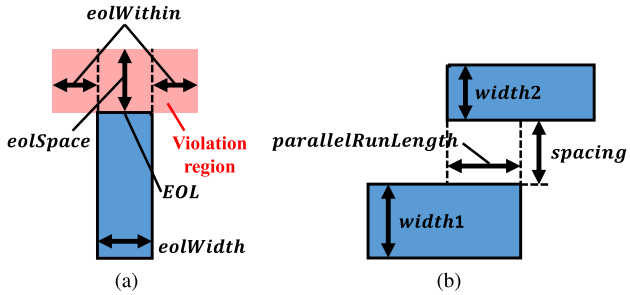

Fig. 2. Example of (a) EOL spacing and (b) parallel-run spacing.

3) We also propose an efficient bulk synchronous parallel scheme to further reduce the turn-around time of the detailed routing process.

The remainder of this paper is organized as follows. Section II introduces the formulation of the VLSI detailed routing problem. Sections III and IV provide the details of our data structures and algorithms, respectively. Section V describes the parallel scheme. In the end, Section VI shows the experimental results, and Section VII concludes this paper.

II. Preliminaries

Before illustrating the details of our data structures and algorithms, the problem formulation of detailed routing is introduced in this section.

A. Routing Space

VLSI routing is on a stack of *metal layers*. A *wire* segment on a layer runs either horizontally or vertically. Each layer has a *preferred direction* for routing, which benefits manufacturability [15], routability, and design rule checking [2]. The preferred directions of adjacent layers are perpendicular to each other in common design practice. Besides, regularly spaced *tracks*, where the majority of wires are routed on, can be predefined according to the wire width and parallel-run spacing constraint. In this paper, wrong-way and off-track wires are considered only for short connections (especially to pins).

Wires on adjacent metal layers can be electrically connected by *vias*. A via is across a *cut layer*, which is between the two metal layers. Note that for vias across a specific cut layer, there may be several via types to be selected from. Different via types have varied metal shapes (usually rectangles with various widths and heights) on the two metal layers. The flexibility provides a way for resolving the spacing violations between vias and obstacles.

The tracks on all metal layers define a 3-D grid graph for detailed routing, as Fig. 1 shows. On each track, there is a series of vertices. Note that a vertex is therefore uniquely defined by a 3-D index, which is a tuple of layer index, track index (in the nonpreferred direction), and relative index along the track (in the preferred direction). A vertex connects downward to the lower layer, upward to the upper layer, or both. Adjacent vertices along a track are also connected. In this way, a same-layer edge represents a possible on-track wire segment, while a cross-layer edge represents a possible via. In this grid graph, an *edge* represents either a wire segment or a via.

Over the chip, there are some routing *obstacles* that vias and wire segments should avoid to prevent short and spacing violations. In detailed routing, the relatively small obstacles within standard cells (e.g., pins and intracell wires) should also be handled.

Assuming that a global routing result is already well optimized for certain metrics (e.g., timing, routability, and power), a detailed router needs to honor the global routing result as much as possible. The optimized metrics are thus kept with detailed design rules handled. In this paper, the 3-D global routing result is referred as *routing guide*, and out-of-guide routing (either wire or via) is penalized.

B. Design Rules

The most fundamental and representative design rules handled by detailed routing are as follows [3].

1) *Short:* A via metal or wire metal cannot overlap with another metal object like via metal, wire metal, obstacle, or pin, except when the two metal objects belong to the same net.
2) *EOL Spacing:* A metal end is an EOL if its width is shorter than *eolWidth*. EOL is required to preserve a spacing greater than or equal to *eolSpace* beyond the EOL anywhere less than the *eolWithin* distance, as Fig. 2(a) shows.
3) *Parallel-Run Spacing:* For two metal objects with *parallelRunLength* (i.e., the projection length between them), there is a spacing requirement, as Fig. 2(b) shows. The value of parallel-run spacing rule depends on the widths of the two metal rectangles.
4) *Cut Spacing:* For vias across the same cut layer, their cut shapes in the cut layer should be sufficiently far away from each other.
5) *Minimum Area:* The area of a metal polygon is required to be above a threshold.

C. Problem Formulation

The detailed routing problem can be formally defined as follows. Given a placed netlist, routing guides, routing tracks, and design rules, route all the nets and minimize a weighted sum of the following.

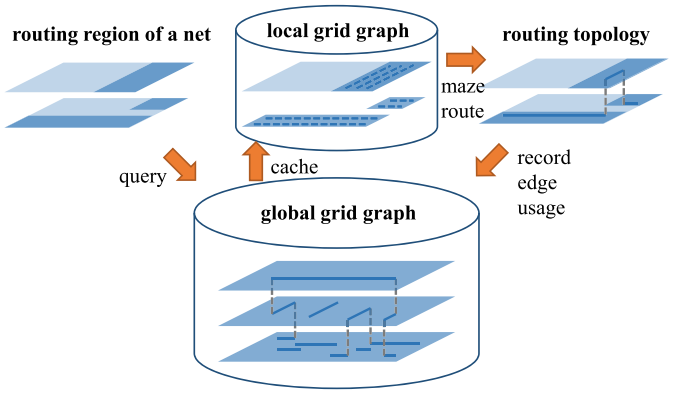

Fig. 3. Overview of the two-level grid graph data structures.

1) Total wire length.
2) Total via count.
3) Nonpreferred usage (including out-of-guide, off-track wires/vias, and wrong-way wires).
4) Design rule violations (including short, spacing, and minimum-area violations).

Note that design rule violations are highly discouraged and suffer much more significant penalty than others.

III. Two-Level Sparse Data Structures

The grid graph for detailed routing is similar to that for global routing in structure, but is significantly more fine-grained and thus has a much larger scale. To support the detailed routing algorithms with both economic memory usage and efficient query, we design a set of two-level data structures for the routing grid graph.

There are a global grid graph and local ones, as Fig. 3 shows. The *global grid graph* data structure stores the graph implicitly without instantiating all vertices. Here, the information of routed edges are stored sparsely by balanced binary search trees (BSTs) and intervals. The *local grid graph*, a local cache of the global one, is created for routing a net. It is a sparse subgraph of the full-chip 3-D grid graph on the routing region of a net, where edge costs are readily available for conducting maze routing.

A. Sparse Global Grid Graph

Edges of routed nets are called *routed edges*. Note that the an edge can be either a via or a wire segment. The global grid graph stores routed edges in the sparse data structure based on BSTs and intervals.

1) BST and Interval-Based Storage: It is very expensive to use a full-chip 3-D direct-address table for storing routed edges. First, its size will be unaffordable (10^9 vertices for just 10 metal layers and 10^4 tracks on each layer) [9]. Besides the time-consuming memory allocation and initialization, some queries are also inefficient if using this data structure. For example, to record, query, or remove the usage of a wire segment (e.g., spanning 10^3 vertices), we need to change or check all the 10^3 vertices on it.

Instead of a 3-D direct-address table, we use a 2-D table for the dimension of layers and the dimension of tracks (i.e., the nonpreferred direction), and use BST and intervals in the third dimension (i.e., the preferred direction) for sparsity. For a track, there are three balanced BSTs, two for storing routed vias and one for storing routed wires. For vias, normal BSTs with indexes in the preferred direction being keys are used. Each via is stored twice, one on the lower track and the other one on the upper track. The duplication benefits the range searches that are needed on both the lower and upper tracks. This will be illustrated in detail later. For wire segments, a BST with nodes representing nonoverlapping intervals is employed. In this way, the memory used is linear to the number of wire segments instead of the number of vertices involved.

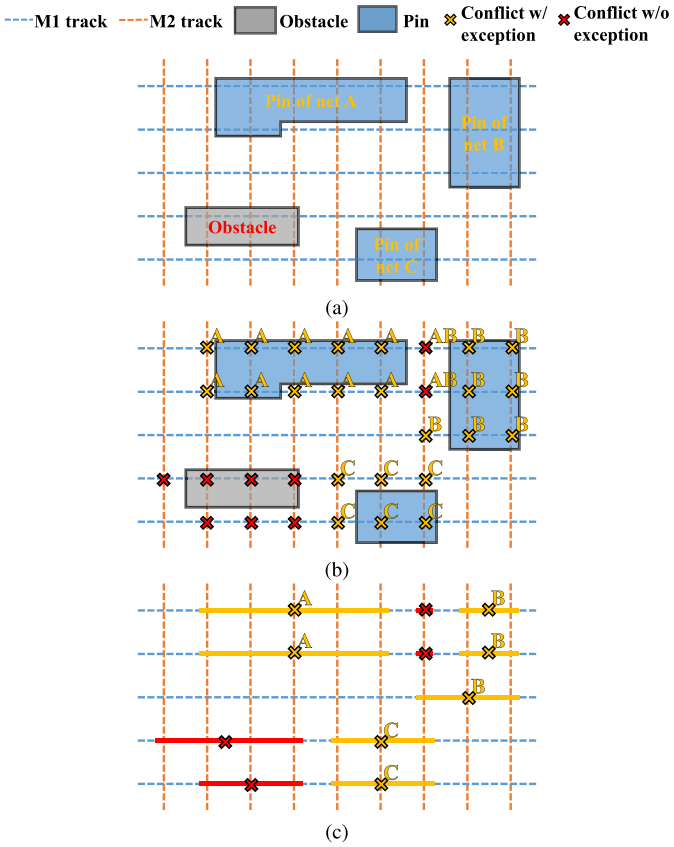

Fig. 4. Wire-obstacle and wire-pin conflicts stored in the global grid graph. (a) Region with an obstacle and three pins. (b) Wires conflicted with obstacles/pins, where a wire-pin conflict is excepted for the net of the pin, but wires conflicted with pins of different nets have no exception. (c) Interval-based storage.

2) Conflicts With Obstacles and Pins: For obstacles and pins with irregular shapes, the vias and wires that may cause short or spacing violations with them are marked in advance in batch. Since obstacles and pins cannot be ripped up, the marking is a one-time effort. Note that a conflict with a pin is net-dependent, because a via or wire is allowed to be close to a pin of the same net. Therefore, some conflicts should be associated with some possibly *excepted net*(*s*).

Fig. 4 shows an example of marking wires conflicted with obstacles and pins. For each obstacle or pin, there are several vertices in the grid graph that will cause short or spacing violations if a wire segment is routed through it. For an obstacle, the conflict applies to all nets [red crosses in Fig. 4(b) indicate

TABLE I
STATISTICS OF VIA-OBSTACLE AND VIA-PIN CONFLICTS ON ISPD18_TEST10

Layer	Metal layer information			Cut layer information								
	# obtacle/pin metal rectangles			Pre-compute?	# via locations						# conflicted intervals	# conflicted intervals / # vias
	Obtacle	Pin	Total		Conflicted (without excepted nets)	Conflicted (with an excepted net)	Conflicted (with multiple excepted nets)	Conflicted (total)	Total	Conflicted / total		
1	839912	2107724	2947636	Yes	41559185	11818934	9608	53387727	189000000	28.247%	16075707	**8.506%**
2	763422	0	763422	Yes	38375423	0	0	38375423	189000000	20.304%	585913	**0.310%**
3	24092	0	24092	Yes	11665650	0	0	11665650	189000000	6.172%	11248534	**5.952%**
4	580772	0	580772	Yes	7537450	0	0	7537450	189000000	3.988%	16040	**0.008%**
5	0	0	0	No	-	-	-	-	-	-	-	-
6	0	0	0	No	-	-	-	-	-	-	-	-
7	0	332	332	No	-	-	-	-	-	-	-	-
8	0	879	879	No	-	-	-	-	-	-	-	-
9	0	0	0	-	-	-	-	-	-	-	-	-

conflicts without exception]; for a pin, the conflict applies to all nets but the net of the pin [yellow crosses in Fig. 4(b) indicate conflicts with exception]. However, the conflicts between a wire and the pins of different nets cannot be excepted. To save memory usage, we use an interval-based storage here as well. Only conflicted vertices are stored, while violation-free vertices are implied. For continuous vertices with the same-type of conflict along a track, they will be stored as an interval, as Fig. 4(c) shows.

Via-obstacle and via-pin violations are more difficult to capture than wire-obstacle and wire-pin violations, because there are several types of vias that can be chosen from. Essentially, all via types need to be attempted. A via location should be penalized if and only if all via types fail to satisfy the spacing requirement with its neighboring obstacles or pins. Note that a via-pin conflict may be excepted for multiple nets due to the via type selection.

When routing a net, the vias that will be considered for using are referred as *candidate vias*. In the preliminary version [1] of this paper, we simply store all the obstacles and pins in R-trees [23] and later query the via-obstacle and via-pin violations from the R-trees. For each candidate via of a net, its neighboring obstacles and pins are queried from the R-trees and checked for possible violations. There is a big drawback with this approach. A via may be treated as a candidate by many nets, resulting in repeated queries and checking processes for a single via. The aforementioned precomputation scheme for conflicted vias can save runtime significantly, which will also be evidenced by the experiments in Section VI.

Three techniques are crucial for enabling such speed-up. First, we only perform the precomputation for metal layers with huge numbers of obstacles and pins.[1] In our implementation, we set a lower bound threshold on the number of obstacle/pin metal rectangles to 10^5. For many designs, it means a precomputation for one or two layers. Second, we store conflicted vias only, while violations-free vias are implied. The third technique is the usage of BST and interval-based storage scheme. The statistics in Table I provides some evidence on the advantages of using these techniques. On `ispd18_test10`, metal layers 1, 2, and 4 have large numbers of obstacles and pins. Therefore, cut layers 1, 2, 3, and 4 need the precomputation of via conflicts (cut layer i connects metal layers i and $i+1$). If storing the conflict situation for all via locations with direct-address tables, it means GB scale memory usage for a single layer (note that we need to store the information of excepted nets). Storing conflicted vias reduces the memory usage to 28.247% for cut layer 1. Using intervals further reduces the usage to 8.506%. For some layers, the reduction can be even much larger (to 0.008%).

[1]We focus the discussion on metal layers for simplicity. In ISPD 2018 benchmarks, which we use for the experiments, there is also no obstacle in cut layers. However, our method is generic and can be easily extended for considering violations in cut layers.

B. Global Grid Graph Query by Look-Up Table

When routing a net, the edges that will be considered for using are referred as *candidate edges*. Their costs (possibly penalized by the short/spacing violations) will be queried from the global grid graph before running maze routing on a net.

Different from the conflict with obstacles, the conflict with routed edges will change during the routing process and cannot be marked in advance. Considering various design rules and a significant number of candidate edges, a proper scheme that can efficiently query their costs is in need. We build several via/wire conflict LUTs to achieve that.

1) Via/Wire Conflict Look-Up Table: For routing a net, the metal short with routed edges can be trivially detected as interval overlapping. For the following spacing violation conflicts, their identification is less straight-forward.

1) *Via–Via Conflicts:* For a specific via, it may conflict not only with vias on the same cut layer (*same-layer vias*) but also with vias on the adjacent cut layers. The conflict between same-layer vias may be due to spacing rules on either cut layer, metal layers, or both. The conflict between different-layer vias is caused by metal spacing requirement.
2) *Via–Wire Conflicts:* A via may have spacing violations with wires on the lower and upper metal layers that it connects.
3) *Wire–Wire Conflicts:* Two wires may be too close to each other at their ends and violate the spacing constraint.

The above violations can be detected during routing. However, these detection operations are extremely frequent and on-the-fly detections are too time consuming. Since, we are working on a relatively regular grid graph, some light-weight LUTs can

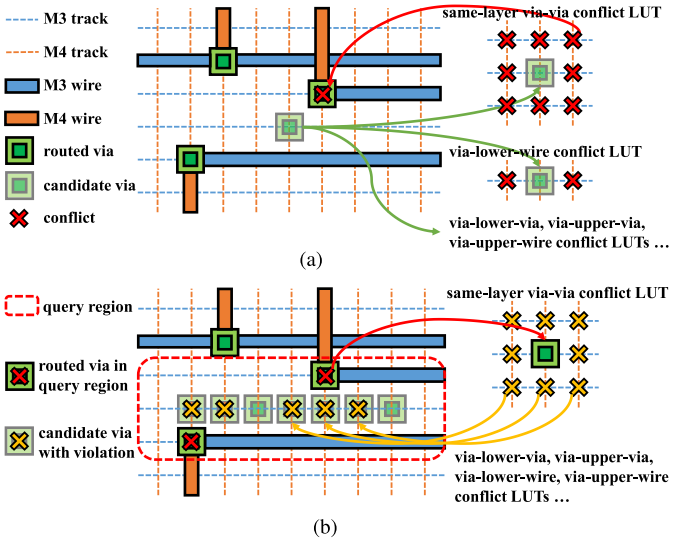

Fig. 5. Query the violations on candidate vias due to the previously routed edges in global grid graph. (a) Query a single candidate via. (b) Query a set of neighboring candidate vias.

accelerate the process. Conceptually, *via/wire conflict LUTs* immediately tells what neighboring edges will conflict with a given edge. There are several types of them: when the given edge is a via e_i, a *via-lower-wire conflict LUT* tells what neighboring wire segments on the lower metal layer of e_i cause conflicts with e_i; similarly, given a wire segment e_j, a *wire-upper-via conflict LUT* tells what vias connecting to the layer above e_j may be conflicted with e_j; so on and so forth. Two conflict LUTs are called the *inverse LUT* to each other if the types of the given edge and the neighboring edges are swapped. For example, the inverse of a via-lower-wire LUT is a wire-upper-via LUT.

Regarding the indexing and sizes of conflict LUTs, we explain the via–via one as an example. For two same-layer vias, their distance is unique for specific track index differences in the lower metal layer and the upper metal layer, because of the equal spacing of the tracks. Therefore, only one LUT is needed for each layer. Such an LUT itself is 2-D and is indexed by the track index differences. For two different-layer vias, three consecutive metal layers are involved. Using their corresponding vertices on the middle metal layer for indexing, their distance in the nonpreferred direction is solely determined by the difference in track indexes. However, in the preferred direction, vertices along a track may have irregular spacing (e.g., M2 in Fig. 1). As a result, a layer needs a series of 2-D LUTs, where each LUT serves for vertices with a specific index in the preferred direction. For each of the 2-D LUT storing the conflicts between a *target via* and its *neighboring vias*, we first need to calculate its size. When the calculation cannot be accurate, we make it pessimistic. An entry of the 2-D LUT represents a neighboring via, of which the distance offset to the target via can be known by the corresponding index offset. Then, for each neighboring via, we test whether there is violation with the target via according to the design rules described in Section II-B and mark the 2-D LUT correspondingly.

2) Single Edge Query: The cost of a candidate edge consists of a unit edge cost and some possible penalty caused by two types of violations. The first type is violations with obstacles and pins, which has been introduced in Section III-A2. The second type is violations with routed edges. The via/wire conflict LUTs tell the neighboring edge positions that will have conflict with the candidate edge. The only thing to do is to check whether the positions are occupied. An example is shown by Fig. 5(a). For the candidate via, a same-layer via–via conflict is detected with the help of the corresponding LUT. Meanwhile, there is no via-lower-wire conflict because no routed wire exists at the two potentially conflicting positions specified by the LUT.

3) Batch/Long Edge Query: Usually, a set of neighboring edges (either vias or wire segments) along a track are all candidate edges for routing a net. If querying them individually, $O(k \log n)$ time is needed with k being the number of candidate edges and n being the BST size.[2] A range search on BST can improve the efficiency. Given a set of candidate edges along a track and the corresponding LUTs, a *query region* where routed edges may have conflicts with can be identified. By the range search on BSTs according to this query region and referring to the inverse LUTs, the conflicted candidate edges can be found. An example on detecting same-layer via–via conflict is illustrated by Fig. 5(b). First, the query region and two routed vias within it are identified. Starting from the two routed vias, the inverse LUT (the same-layer via–via conflict LUT) finds five conflicted candidate vias.

Suppose the number of routed edges within the query region is m. The range search on a BST takes $O(m + \log n)$ time, which can be conducted by finding the first and last tree nodes within the range. Besides, $m = O(k)$. Note that m can be significantly smaller than k because a long routed wire segment is stored as a long interval instead of a bunch of short edges in a BST. Therefore, the time for retrieving the routing cost of the k candidate edges is $O(k)+O(m+\log n) = O(k+\log n)$ instead of $O(k \log n)$. Moreover, the cost of a long wire segment may be queried as a whole, then the time is further improved to $O(m + \log n)$.

In the batch query along a track, routed vias to both lower and upper layers should be considered. As mentioned in Section III-A1, a via is stored twice on both its lower and upper tracks. In this way, efficient BST range search along either track is enabled.

C. Sparse Local Grid Graph

The local grid graph of a net is the subgraph of the full-chip 3-D grid graph within its *routing region* (the routing guide with possibly minor expansion). In terms of data structures, it caches the graph structure and all edge costs of the subgraph by direct-address tables, supporting the maze routing.

Its sparsity is in two aspects. First, only the routing region is considered, which is substantially smaller than the full-chip

[2]To be more rigorous, since multiple BSTs (for vias or wires, for different layers) may all need to be queried, n represents the largest size of all BSTs.

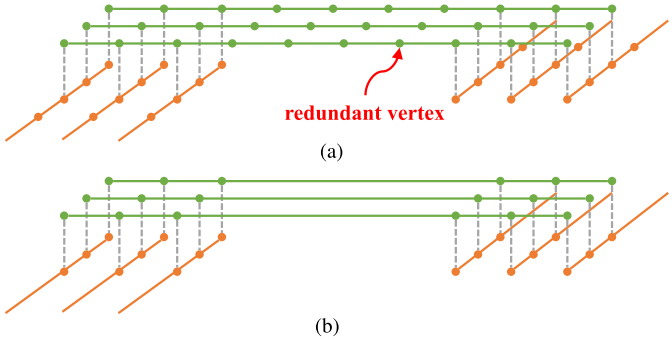

Fig. 6. Long edges by removing redundant vertices. (a) Before removing. (b) After removing.

region. Second, many vertices become redundant in this subgraph and are removed.

1) Routing Region: When routing a net, only the region around its routing guide is considered due to two reasons. First, detailed routing should honor global routing solution, i.e., routing guides, because many objectives (e.g., timing and routability) are optimized in global routing. For some local congestions, global routing may not be able to model and resolve, so minor out-of-guide routes may be necessary. However, such disturbance should be minimized. Second, maze routing on the full-chip 3-D grid graph will suffer from prohibitive runtime due to its enormous scale. In our implementation, the routing region of a net is expanded by a small margin from its routing guide. All out-of-guide edges are penalized. For difficult-to-route nets, the expansion margin may be increased.

2) Long Edge: Conceptually, the local grid graph is simply a subgraph induced by vertices within the routing region. However, many vertices in the subgraph have only two neighbors remained and become redundant, as Fig. 6(a) shows. In this snippet of the subgraph, many vertices originally have neighbors on adjacent layers that are out of the routing region now. They have thus only two neighbors left on the track. In this way, as long as such a vertex does not belong to a pin, it can be safely removed with the two connected edges merging into one. This compressing step cuts down the problem size without affecting the final results. Both memory usage and runtime can be reduced.

3) Wrong-Way Edge: Wrong-way edges are discouraged due to three reasons. First, more regular designs with fewer wrong-way usage is beneficial to manufacturability [15]. Second, a long wrong way edge will block many tracks, which hurts routability. Third, for routing a single net, heavy usage of wrong-way edges leads to a significantly larger solution space and thus runtime overhead.

But it should be allowed. In the preliminary version [1] of this paper, we only try using wrong-way edges in some post processing steps. However, it turns out that adding some wrong-way edges in the local grid graph can greatly benefit escaping congested tracks. In our implementation, we add wrong-way edges densely in the small regions around pins. Besides, along two neighboring tracks, a wrong way edge is added for every ten vertices. Since we store wires as intervals along tracks in the global grid graph, a wrong-way wire will be segmented and stored as degenerated intervals (i.e., points) on the tracks that it spans. This storage scheme is still efficient in general because wrong-way usage is the minority. The improvement due to the wrong-way consideration will be shown in Section VI.

4) Explicit Storage: In the global grid graph, vertices are implied by 3-D indexes but are not instantiated. To support efficient vertex-wise operation in maze routing (e.g., recording the prefix and cost, and propagating to neighbors), the local graph instantiates all its vertices and edges. To be more specific, vertices are assigned with continuous indexes starting from zero, and adjacency lists are also created. In this way, any vertex/edge information can be efficiently stored and retrieved by direct-address tables (instead of hash tables or BSTs).

IV. Routing Algorithm

In routing (especially detailed routing), sequential maze routing is widely adopted due to its scalability (compared with concurrent methods like [8] and [8]) and flexibility (for capturing various objectives and violations). Recall from Fig. 3 that our local grid graph is sparse because of the routing guide and long edges, which enhances the efficiency of our maze routing. We follow the convention of sequential maze routing. Essentially, nets are routed one after another, where previously routed nets are treated as blockages. After routing all nets with possible violations, several rounds of rip-up and reroute (RRR) help to clean them up.

A. Edge Cost in Local Grid Graph

The cost $w(e)$ of each edge e in the local grid graph $G(V, E, w)$ is a weighted sum of the following.

1) Basic wire cost (by length).
2) Basic via cost (by count).
3) Out-of-guide penalty.
4) Short/spacing violation penalty.

In this way, a path search (like Dijkstra's algorithm [25]) running on the grid graph will optimize these objectives automatically. The basic via/wire cost together with the short/spacing violation penalties are queried from the sparse global grid graph in batch. The out-of-guide penalty is charged according to the routing guide after the query.

Note that it is not determined by a single edge whether the minimum-area rule is violated or not. The minimum-area violation thus cannot be reflected as expensive edges like short/spacing violations and can only be captured by the path search algorithm.

B. Minimum-Area-Captured Path Search

For wires with a specific width, a minimum area implies a minimum-length constraint $l_{\min}$. A straight-forward idea for fixing the violation after maze routing is to extend the wire segments that are not long enough. Such a greedy method may suffer from excessive wire length [e.g., Fig. 7(b) compared with Fig. 7(c)] and even insufficient spare space for extension. Another method, multilabel path search [12], forces

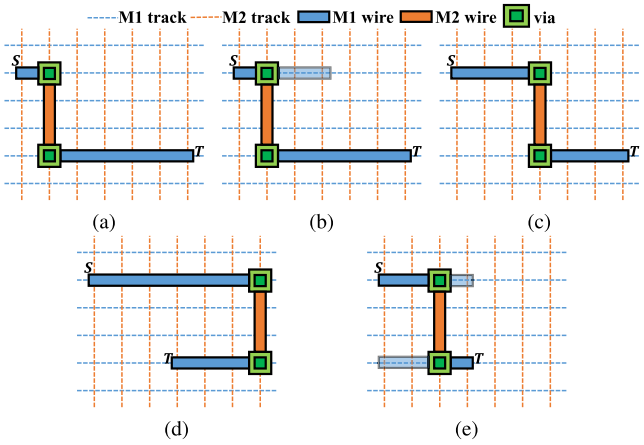

Fig. 7. Capture minimum area cost in path search. Suppose the minimum area implies a length of three pitches. A path from source S to sink T is needed. (a) Normal path search without considering minimum-area violation. (b) Post fixing by extending wire. (c) Forcing the minimum length of wire segment in path search. (d) Detor due to the forcing. (e) Path search with wire extension considered.

the minimum length for every wire segment without considering the possibility of extension. In this way, significant but unnecessary detor may be paid [Fig. 7(d)]. By capturing the minimum-area violation and its possible fixing during the path search, a better solution can be obtained [Fig. 7(e)].

We extend the conventional Dijkstra's algorithm [25] to comprehensively handle the minimum-area rule. In Dijkstra's algorithm, the cost/distance of a path can be directly incremented. That is, the cost of a path from vertex v_1 via v_2 to v_3 is simply the sum of the cost of the two partial paths

$$\text{cost}(v_1 \rightsquigarrow v_2 \rightsquigarrow v_3) = \text{cost}(v_1 \rightsquigarrow v_2) + \text{cost}(v_2 \rightsquigarrow v_3).$$

The challenge for considering the minimum area constraint is an uncertain cost of a partial path, which is unknown until the path turns or stops. At vertex v_2, it is unknown whether a minimum-area overhead (either wire extension or violation penalty) is needed, which depends on the future propagation of the path. However, for a path up to a certain wire segment, bounds on its cost can be calculated as follows.

1) *Lower Bound Cost:* Sum of edge costs plus the minimum-area overhead on all the previous wire segments.
2) *Upper Bound Cost:* Lower bound cost plus the potential minimum-area overhead on the current wire segment.

Our path search is detailed by Algorithm 1. The process is still based on a priority queue Q, but the operation domain is generalized from vertices to paths, because each vertex may have several candidate paths now. The information stored for a partial path P' includes the following.

1) Prefix path P'.prefix and current vertex P'.vertex. Note that such incremental storage requires $O(1)$ memory only for each propagated path, instead of $O(|P'|)$ with $|P'|$ being the number of vertices in P'.
2) The lower bound P'.costLB and upper bound P'.costUB of the path cost.
3) Length of the current wire segment P'.length. It is needed for calculating the minimum-area overhead.

Algorithm 1 Optimal Minimum-Area-Captured Path Search

Require: A local grid graph $G(V, E, w)$, source and sink vertices s and t, minimum length l_{min} of wire segment (implied by the minimum-area constraint).
Ensure: $s - t$ path P.

```
1:  Q ← an empty priority queue for storing paths
2:  v.costUB ← ∞, ∀v ∈ V
3:  Initialize path P' at s (P'.prefix ← null, P'.vertex ← s,
    P'.costLB ← 0, P'.costUB ← 0, P'.length ← l_min)
4:  Push P' into Q
5:  while Q is not empty do
6:      Pop the path P' with smallest P'.costLB from Q
7:      if P'.vertex = t then
8:          return P'
9:      end if
10:     for v ∈ P'.vertex.neighbors do
11:         RELAX(P', v)
12:     end for
13: end while

14: function RELAX(P', v)                    ▷ Extend path P' to v
15:     P''.prefix ← P'
16:     P''.vertex ← v
17:     if P'.vertex.layer ≠ v.layer then
18:         P''.costLB ← P'.costUB + w(P'.vertex, v)
19:         P''.length ← 0
20:     else
21:         P''.costLB ← P'.costLB + w(P'.vertex, v)
22:         P''.length ← P'.length + dist(P'.vertex, v)
23:     end if
24:     P''.costUB ← P''.costLB+
    MINAREAOVERHEAD(P''.length, v.hasSpace)
25:     if P''.costLB < v.costUB then
26:         Push P'' into Q
27:         if P''.costUB < v.costUB then
28:             v.costUB ← P''.costUB
29:         end if
30:     end if
31: end function
```

The information stored at each vertex v is the smallest upper bound cost v.costUB among all the paths reaching it.

In each iteration, the path P' with the smallest lower bound cost in the priority queue Q is popped out (line 6). It will be considered for propagating to the neighbors of P'.vertex. For an extended path P'' to a neighbor $v \in P'$.vertex.neighbors, satisfying P''.costLB $< v$.costUB means that P'' is a potentially optimal path and should be considered for further propagation (line 25). If P''.costLB $\geq v$.costUB, P'' can be pruned. The algorithm stops when a sink vertex is reached (line 7). Note that for a sink vertex, the pin metal is sufficiently large and thus can guarantee that P'.costLB is achievable (i.e., no minimum-area overhead charged).

The overhead due to the minimum-area rule depends on the length of the current wire segment P''.length, whether vertex v has sufficient spare space for wire extension (v.hasSpace),

and the minimum length requirement $l_{\min}$ (line 24). To be more specific

$$\text{MINAREAOVERHEAD}(P''.\text{length}, v.\text{hasSpace}) = \begin{cases} 0, & \text{if } P''.\text{length} \geq l_{\min} \\ w_{\text{wire}} \cdot (l_{\min} - P''.\text{length}), & \text{if } P''.\text{length} < l_{\min} \text{ and } v.\text{hasSpace} \\ w_{\text{minArea}}, & \text{otherwise} \end{cases}$$

where w_{wire} is the unit-length basic cost for wires, and w_{minArea} is the penalty for each minimum-area violation. Note that the flag v.hasSpace for all the vertices in the local grid graph can be queried from the global grid graph in batch. The flags are then stored explicitly in the direct-address table mentioned in Section III-C4.

Theorem 1 states the optimality of Algorithm 1. The proof is similar to that of the original Dijkstra's Algorithm (see [26]).

Theorem 1: For a given local grid graph $G(V, E, w)$, Algorithm 1 gives an optimal $s - t$ path P satisfying the minimum length constraint $l_{\min}$.

The path search algorithm in MANA [11] also captures the minimum length constraint in a similar manner. The strengths of our approach over MANA are twofold. First, our framework allows minimum-area violations to exist in earlier RRR iterations. The minimum-area penalty serves as the Lagrange multiplier [4] and helps to build a smooth RRR optimization process. It avoids satisfying minimum-area constraint at a huge price of sacrificing other metrics (e.g., wire length) in early iterations but still leads to almost zero minimum-area violation eventually. Second, we query the flag v.hasSpace in batch from our global grid graph, which is more efficient.

For a multiple-pin net, path search starts from a source pin s. When reaching the first other pin, all vertices on the path are regarded as source for searching a next pin, until all pins are reached [27].

C. Rip-Up and Reroute

One round of sequential maze routing usually cannot generate a violation-free solution for all the nets. Several rounds of RRR help to iteratively reduce the number of violations. Our RRR strategy is similar to those widely used in global routing (e.g., NCTU-GR [10]) with two major differences. First, only nets with violations are ripped up to save runtime, considering that detailed routing is more time consuming. Second, for ripped-up nets, their routing regions will be slightly expanded for attempting a larger solution space in the next iteration.

For the wires and vias with design rule violations in previous RRR iterations, a history cost is recorded. Note that for a wire segment with violations, history cost is charged only for the intervals with violations on it. In this way, the actual situation of resource competition can be reflected. Regarding the value of history cost, it is discounted compared with the design rule violations showing in the current iteration because current violations are definite. Besides, there is a fading factor so that history violations more iterations ago will have less impact. Such a negotiation-based RRR results in a better and faster convergence, as Section VI will show.

V. PARALLELISM

Detailed routing is time-consuming in general. There are many jobs during the whole process that can be easily parallelized. For example, the initialization of conflicted wires and vias in the global grid graph can be conducted in parallel for different layers and different regions of a chip. However, the major runtime bottleneck of Dr. CU is to construct the local grid graph, run maze routing, and update the global grid graph for each net.

The turn-around time of detailed routing can be further shortened by routing different nets in parallel. The challenge here is that the routing regions of different nets may overlap. We design an efficient bulk synchronous parallel scheme [28]. It routes batches of independent nets one after another. Note that such independence, together with a deterministic scheduling of batches, can ensure deterministic routing results.

For nets in the same batch, their routing regions do not overlap. Here, a *safety margin* is also considered, which captures spacing rules and possible wire extension for minimum-area compliance. There are two phases for each batch. The *routing phase* queries nets from the global grid graph, constructs the local grid graphs, and runs maze routing; the *committing phase* records routed edges into the global grid graph (see Fig. 3), which can be regarded as a data synchronization needed by later batches. The parallelism for the independent jobs in either the routing or committing phase is trivial: each thread keeps consuming a net from a pool of unprocessed nets until the pool becomes empty. With runtime dominated by the routing phase, the reason for having a separate committing phase is to avoid a heavy usage of mutual exclusion (mutex) [29] among threads. Routed edges in the global grid graph are stored by BSTs. A BST cannot be accessed when it is being modified by another thread, even if the ranges of access and modification do not overlap. One solution is to set up locks. Its drawback is that reading BSTs is significantly more frequent than writing. Note that for a net, reading BSTs is performed on its routing region, while writing is only performed for the solution paths, which comprises just a small part of the whole routing region. By separating the committing phase, the BST read in the routing phase becomes lock-free and thus can be performed faster.

A scheduling of all the batches will be performed in the beginning of an RRR iteration by Algorithm 2. Nets are assigned one after another by trying to join an existing batch (lines 4–9) and thus minimizing the number of batches. R-trees are used to detect the conflict between a net and a candidate batch. For a batch of nets, there are several R-trees storing their rectangular routing regions, one for each layer. In this way, the scheduling is very efficient and empirically only takes 1.02%–2.07% of the total running time. Fig. 8(a) shows the runtime profile of all the batches on a test case. Note that in a batch, different threads may finish their last jobs at different time and thus have various durations. The maximum duration of all the threads is the time that a batch needs, while the average duration is the runtime lower bound that can be achieved by an ideal scheduling. Their small difference shown in Fig. 8 justifies the good quality of our scheduling.

Algorithm 2 Scheduling for Parallel Routing

Require: Nets
Ensure: *batchList*
1: Sort all nets in decreasing size of routing region
2: *batchList* $\leftarrow \emptyset$
3: **for** each net n_i **do**
4: **for** each batch b_j in *batchList* **do**
5: **if** n_i has no conflict with b_j **then** ▷ By R-trees
6: Add n_i into b_j
7: Break
8: **end if**
9: **end for**
10: **if** n_i has not been assigned to any batch **then**
11: Append a single-net batch with n_i to *batchList*
12: **end if**
13: **end for**
14: Reverse the order of batches in *batchList*
15: **for** each batch b_j in *batchList* **do**
16: Sort nets in b_j by decreasing size of routing region
17: **end for**

Moreover, we apply three techniques to further improve the effectiveness of scheduling. The first two techniques are to enhance the load balancing.

1) *Within-Batch Balancing (WBB, Algorithm 2 Lines 15–17):* The workload of different threads in a batch can be more balanced by processing larger nets first. The improvement is evidenced by the smaller gaps between the maximum and the average durations of each batch in Fig. 8(b).
2) *Interbatch Balancing (IBB, Line 1):* Attempting larger nets first during the scheduling can improve the parallelism, as Fig. 8(c) shows. The benefits are in threefold. First, larger nets are more likely to have overlap with the existing nets in a candidate batch. Therefore, IBB can help to reduce the number of batches by increasing the success rate of larger nets (e.g., reduced from 123 to 96 for the first RRR iteration on `ispd18_test9`). Second, our scheduling algorithm tends to make later batches with fewer nets and thus worse load balancing among threads. IBB remedies the problem by making later batches have fewer nets and by making nets in later batches smaller. Third, some nets may be very huge and need a long time to be routed. If the other nets in its batch do not take a sufficiently long time in total, there will be a single thread routing the huge net with other threads idle [seen by the long "pulse" in Fig. 8(b)]. IBB can gives more load to the batch of huge nets (usually the first several batches) and avoid such an issue.

The third technique is *batch with small nets first* (BSF, line 14). IBB also lets large nets be routed earlier. The problem is that small nets are less flexible in maze routing than large nets due to their smaller solution space. Routing large nets first makes later small nets even more difficult to be routed. BSF reverses the order of all batches and avoids the problem. In terms of runtime, it leads to fewer nets with violations in an RRR iteration, reroutes fewer nets in the next RRR iteration, and thus saves the runtime, which can be seen from Fig. 8(d).

The eventual runtime benefits of the three techniques will be shown in Section VI.

TABLE II
METRIC WEIGHTS IN ISPD 2018 CONTEST BENCHMARKS

Metric		Weight
Basic cost	wire length	0.5
	# vias	2
Non-preferred usage	out-of-guide wire length	1
	# out-of-guide vias	1
	off-track wire length	0.5
	# off-track vias	1
	wrong-way wire length	1
Design rule violations	short metal area	500
	# spacing violations	500
	# min-area violations	500

VI. EXPERIMENTAL RESULTS

Dr. CU is implemented in C++ with the boost geometry library [30] for R-tree query and Rsyn [31] as parser. The experiments are performed on a 64-bit Linux workstation with Intel Xeon Silver 4114 CPU (2.20 GHz, 40 cores) and 256 GB memory. Benchmarks are from the ISPD 2018 Initial Detailed Routing Contest [3]. The metric weights for the total quality score and the benchmark characteristics are shown by Tables II and III, respectively. Consistent with the contest, eight threads are used by default. The result reporting is conducted by Cadence Innovus 17.1 [32] and the official evaluation script [33].

The result statistics of Dr. CU is illustrated by Table IV. Fig. 9 shows a GUI view of the solution on `ispd18_test10`.

A. Effectiveness of Quality Enhancement

Fig. 10 shows the score breakdown of Dr. CU on `ispd18_test9` and `ispd18_test10` across the four RRR iterations. The score is calculated under the metric of ISPD 2018 Contest and divided into three categories—basic cost, nonpreferred usage, and design rule violations. During the RRR process, even though the nonpreferred usage (especially out-of-guide wire length) may slightly increase, the design rule violations can be significantly reduced. This demonstrates the effectiveness of our RRR scheme. We set the number of RRR iterations to four for all the ten cases as a proper tradeoff between quality and runtime. A fifth iteration can improve the total quality score by 0.5% but needs 31% more runtime on average.

Fig. 11 shows the enhancement due to three other techniques. First, adding some wrong-way edges in the local grid graph (Section III-C3) helps to enlarge the solution space and thus alleviate the congestion problem, which brings 1.6%–38.6% score improvement with the average being 7.7%. Second, the minimum-area-captured path search (Section IV-B) provides more correct-by-construction design

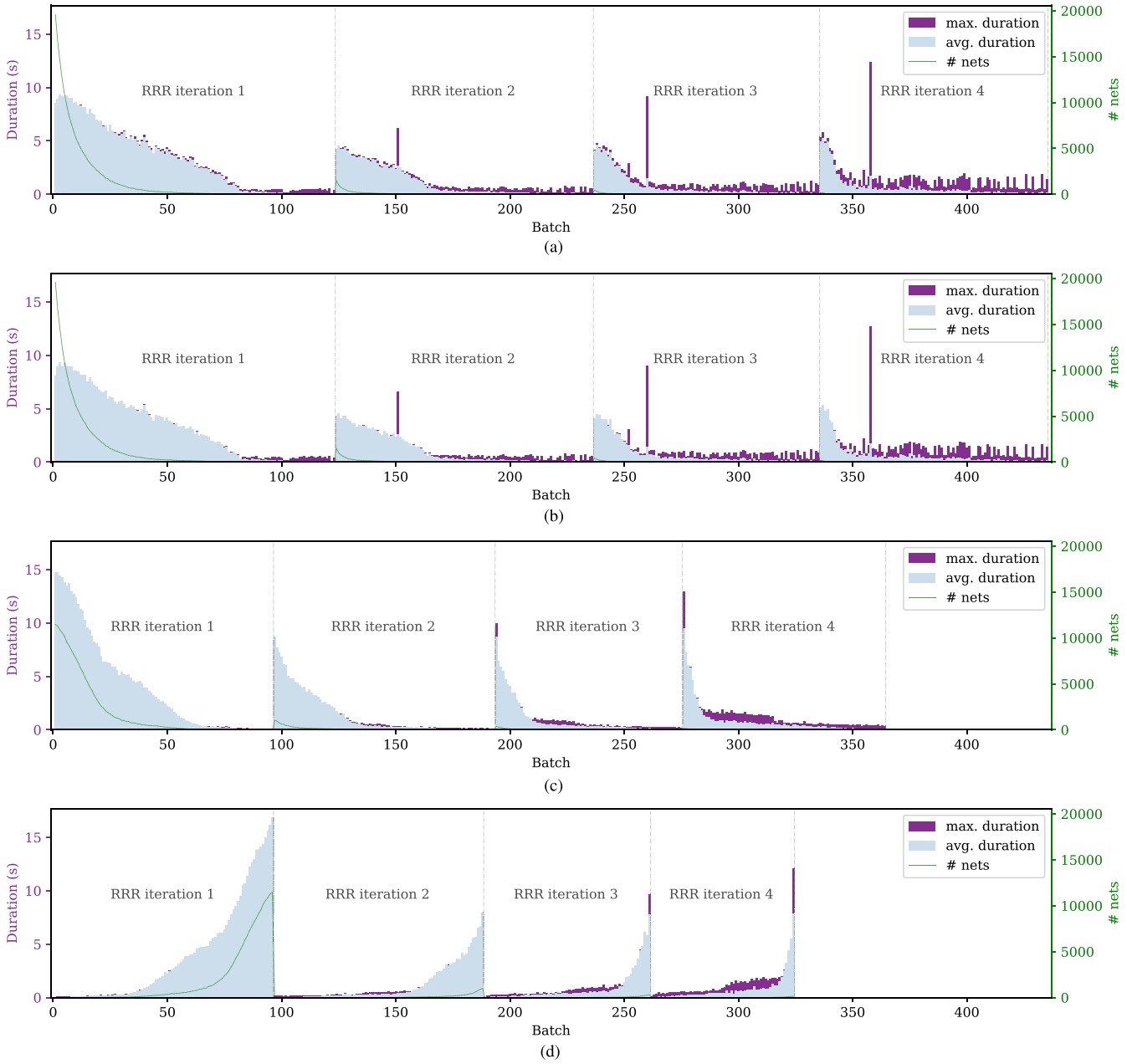

Fig. 8. Better parallelism by WBB, IBB, and BSF. The result is on `ispd18_test9` and across four RRR iterations. (a) None (routing phase 937 s). (b) WBB (routing phase 906 s). (c) WBB + IBB (routing phase 827 s). (d) WBB + IBB + BSF (routing phase 749 s).

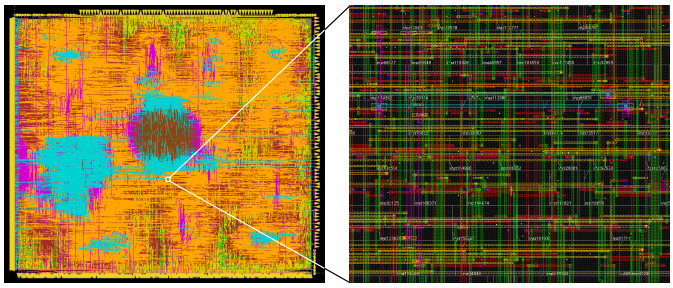

Fig. 9. Solution of Dr. CU on `ispd18_test10`.

rule satisfaction. To be more specific, it reduces the number of minimum area violations by up to 100% and on average 82.6%. The total score is therefore improved by up to 2.6% and on average 0.9%. Third, using history cost in RRR (Section IV-C) improves the quality score by up to 2.0% and on average 1.0% eventually. Meanwhile, it also results in a faster convergence, reducing the total runtime by 6.8% on average.

B. Effectiveness of Runtime Reduction

Fig. 12 shows the speed-up due to precomputing via-obstacle and via-pin conflicts. Here, the turn-around time of the whole detailed routing process is saved by 32.2%–62.8%, which is 49.9% on average.

The acceleration achieved by our parallelism is shown in Fig. 13. Eight threads give around five to six times speed-up compared with single-thread routing. Here, WBB, IBB,

TABLE III
ISPD 2018 CONTEST BENCHMARK CHARACTERISTICS

Benchmark	# std. cells	# block macros	# nets	# pins	# IO pins	# layers	M2 # tracks	M2 pitch (μm)	Die size (mm^2)	Tech. node (nm)
test1	8879	0	3153	17203	0	9	977	0.2	0.20×0.19	45
test2	35913	0	36834	159201	1211	9	3254	0.2	0.65×0.57	45
test3	35973	4	36700	159703	1211	9	4943	0.2	0.99×0.70	45
test4	72094	0	72401	318245	1211	9	8886	0.1	0.89×0.61	32
test5	71954	0	72394	318195	1211	9	9800	0.1	0.93×0.92	32
test6	107919	0	107701	475541	1211	9	5312	0.1	0.86×0.53	32
test7	179865	16	179863	793289	1211	9	13500	0.1	1.36×1.33	32
test8	191987	16	179863	793289	1211	9	13500	0.1	1.36×1.33	32
test9	192911	0	178857	791761	1211	9	13500	0.1	0.91×0.78	32
test10	290386	0	182000	811761	1211	9	13500	0.1	0.91×0.87	32

TABLE IV
COMPARISON WITH STATE-OF-THE-ART ACADEMIC DETAILED ROUTERS ON ISPD 2018 CONTEST BENCHMARKS

		Basic cost		Non-preferred usage					Design rule violations					**ISPD'18 quality score**	Mem (GB)	**Time (s)**
		WL[a]	# vias	Out-of-guide WL[a]	Out-of-guide # vias	Off-track WL[a]	Off-track # vias	Wrong-way WL[a]	# short	Short area[a]	# min area	# spacing	**Total #**			
Dr. CU	test1	433254	32031	1706	446	393	0	4749	4	0.4	0	17	**21**	**296504**	0.33	**11**
	test2	7806294	317160	34194	5948	4937	0	44495	12	1.3	0	73	**85**	**4661740**	1.70	**85**
	test3	8683731	307545	52408	5499	5714	0	45541	346	372.5	0	161	**507**	**5330014**	1.75	**113**
	test4	26033480	658644	132938	16103	9190	0	59579	463	436.8	6	1071	**1540**	**15304156**	3.94	**320**
	test5	27729394	916715	92872	16686	1588	0	44680	406	77.4	10	496	**912**	**16144832**	5.42	**426**
	test6	35595790	1403634	142595	25939	8735	0	69829	168	92.7	21	587	**776**	**21198243**	6.48	**527**
	test7	64994186	2271738	235497	36269	16459	0	106884	772	230.8	38	325	**1135**	**37724327**	10.77	**969**
	test8	65289434	2281513	290418	38596	17082	0	111173	861	249.5	20	399	**1280**	**37990696**	11.73	**1034**
	test9	54602832	2282226	284645	42078	12746	0	108324	297	162.7	28	379	**704**	**32592136**	11.20	**906**
	test10	67907614	2439531	1137257	64535	30527	0	197840	14605	11370.4	44	3910	**18559**	**47909940**	11.95	**1299**
	Avg. ratio	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	**1.00**	**1.00**	1.00	**1.00**
[1]	test1	434914	34443	4352	859	276	0	2363	127	15.3	0	122	249	362725	0.33	22
	test2	7817285	339055	104720	11784	4353	0	22023	1005	1329.9	0	1949	2954	6366885	1.48	114
	test3	8707641	331958	176736	10731	4344	0	22187	2444	1982.1	0	2419	4863	7430091	1.57	128
	test4	26042785	701994	769265	31444	41791	0	89537	6914	26328.8	0	11224	18138	34112927	3.50	443
	test5	27852167	942588	649224	43071	13390	0	63397	5466	4722.2	0	7742	13208	22805759	4.48	692
	test6	35813473	1446807	976672	68656	20357	0	95811	7959	12891.0	0	11023	18982	33908650	6.35	1054
	test7	65360688	2349580	2187794	101866	33105	0	170316	23141	33040.9	0	14880	38021	63816461	10.54	1848
	test8	65668468	2360231	2288159	102982	33373	0	170583	20641	22352.8	0	14384	35025	58501486	10.62	1867
	test9	54993356	2358857	1604576	115465	29620	0	168722	18830	17315.6	0	14470	33300	50010786	10.43	1804
	test10	68282001	2532666	2826908	140343	32865	0	180586	26688	150704.9	0	20837	47525	128141528	11.10	1909
	Avg. ratio	1.00	1.05	5.39	2.34	2.50	-	1.14	31.75	165.37	0.00	21.91	21.76	1.67	0.92	1.66
[21]	test1	464503	39199	5659	1301	64	114	17	4364	1.1	0	120	4484	378304	4.43	101
	test2	8097032	385111	63976	12746	2474	1241	172	29845	36.3	1	1419	31265	5626235	29.19	897
	test3	9013950	389718	42336	691	19905	1117	205	34753	1507.2	0	1755	36508	6971806	38.45	1395
	test4	27165618	847643	267804	50040	180535	1675	1016	42024	17088.6	54	3130	45208	25825193	52.44	6164
	test5	29206112	1142635	278618	54613	20047	9905	1174	145826	1795.2	118	7438	153382	21918275	34.75	2317
	test6	37905264	1768984	408860	80328	31165	16368	2765	152194	1937.0	188	11630	164012	29892011	35.89	3807
	test7	68655629	2866477	677287	131394	93328	23387	4378	243375	9187.7	270	12896	256541	52120721	44.53	6561
	test8	68988139	2879647	696911	135073	88394	23598	4506	238519	8386.2	240	12744	251503	51842741	44.98	6136
	test9	58255989	2872574	620805	127167	51854	23438	4449	264230	2531.7	260	12581	277071	43361290	45.04	5737
	test10	71637851	3055779	957064	138008	232529	27502	5610	340647	12162.8	259	16164	357070	57467840	47.21	12614
	Avg. ratio	1.05	1.25	2.22	2.69	6.25	-	0.02	653.96	20.71	9.15	18.41	189.76	1.35	9.39	9.25
[22][b]	test1	487842	42565	1107	502	167	0	1724	95	4.4	0	773	868	721170	0.29	67
	test2	8322145	403543	34068	5689	2783	0	17897	1218	172.0	0	6803	8021	8514694	1.93	1680
	test3	9212616	398144	16211	5987	1920	0	16672	3919	1365.2	0	8477	12396	10363523	2.09	2194
	test4	27699631	822662	56624	28601	4369	0	56093	5969	8273.2	126	45661	51756	42668733	4.78	7201
	test5	29493060	1072812	130131	13385	15748	0	107889	7037	7681.5	37	96575	103649	69298177	5.46	10017
	test6	38123660	1641879	190819	21811	30645	0	183006	10375	12213.7	48	113048	123471	85411399	7.75	16345
	test7	69033346	2656802	453807	39365	52255	0	302204	19802	20961.3	108	179190	199100	140781447	13.11	51768
	test8	69039670	2621050	468925	39302	53192	0	307267	20944	22601.7	103	182494	203541	143203359	13.53	51328
	test9	58299612	2621857	336589	39504	46044	0	301241	19246	17949.0	74	185270	204590	136740361	13.39	45024
	test10	71636304	2791064	523896	47663	60925	0	358457	35218	221528.4	55	218478	253751	262391463	14.21	71552
	Avg. ratio	1.07	1.21	1.03	1.04	2.71	-	1.73	34.57	70.45	5.55	217.47	110.52	3.45	1.14	33.00
1st place of ISPD 2018	test1	472032	41641	6246	1385	3528	116	3509	4223	0.7	0	107	4330	386190	5.66	100
	test2	8150588	409551	71685	13451	20402	1362	18214	36601	94.9	1	1158	37760	5636272	29.91	831
	test3	9086139	427410	69182	2450	33470	1216	18882	46966	4891.4	0	1387	48353	8645535	41.44	1408
	test4	27514053	858224	240226	8841	150961	1011	224715	349597	52947.1	6	50957	400560	67978775	43.93	4374
	test5	29415618	1158945	342675	31391	46870	10514	194054	431909	28428.7	28	66742	498679	65227110	23.02	1794
	test6	38191983	1800286	471017	42714	151178	17549	281027	628776	31227.5	15	100196	728987	89303688	28.36	2969
	test7	fail	fail	fail	fail	fail	fail	fail	fail	fail	fail	fail	fail	fail	fail	fail
	test8	69559382	2929578	1006247	82478	375236	22294	455824	1058138	76790.0	48	161229	1219415	161426825	40.78	5030
	test9	58803453	2920259	813750	67367	331766	22915	446432	1051112	56580.8	40	158305	1209457	144221468	40.16	4481
	test10	72244024	3110163	1414338	81831	625291	27392	476670	1289359	120966.0	33	177426	1466818	193867712	43.42	5271
	Avg. ratio	1.06	1.30	2.61	1.66	16.74	-	2.70	1628.84	175.34	1.52	138.97	582.42	3.28	9.90	7.60

[a] Unit of length is M2 pitch; unit of area is the square of M2 pitch.
[b] Two versions, with and without spacing-to-short conversion, are reported in [22]. The version without spacing-to-short conversion is shown here because it is more practically meaningful.

and BSF contribute 3.63%, 9.20%, and 6.65% improvement on average, respectively. In total, "WBB+IBB+BSF" saves runtime by 13.9%–32.2% (on average 18.5%).

Fig. 14 shows the runtime breakdown of Dr. CU on ispd18_test10. Before routing, the global grid graph and conflict LUTs are initialized, which takes 4.9% of the total

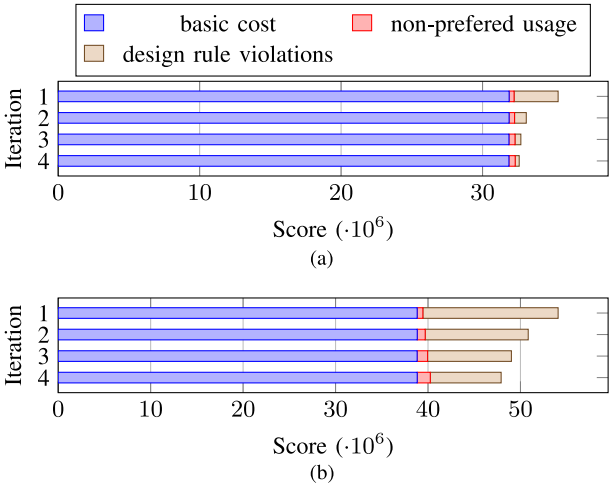

Fig. 10. Improving routing quality by RRR. (a) On `ispd18_test9`. (b) On `ispd18_test10`.

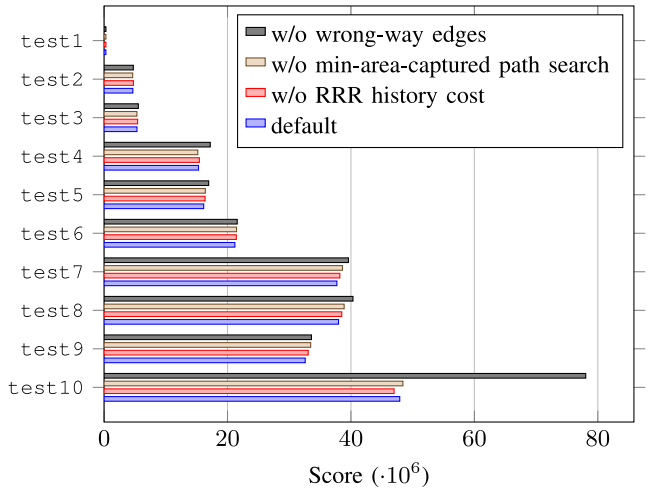

Fig. 11. Improving routing quality by using wrong-way edges, minimum-area-captured path search, and history cost.

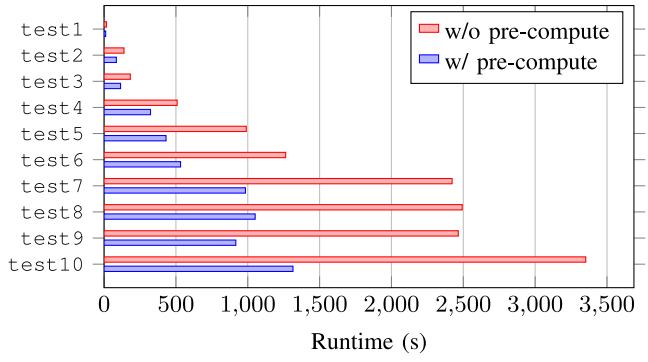

Fig. 12. Speed-up by precomputing via-obstacle and via-pin conflicts.

runtime. We define the process of caching (including querying the global grid graph and constructing local grid graphs) and maze routing as *core routing*, which is the major consumer of runtime (38.4% + 37.3% + 3.7% = 79.4%). In each RRR iteration, core routing is performed under our bulk

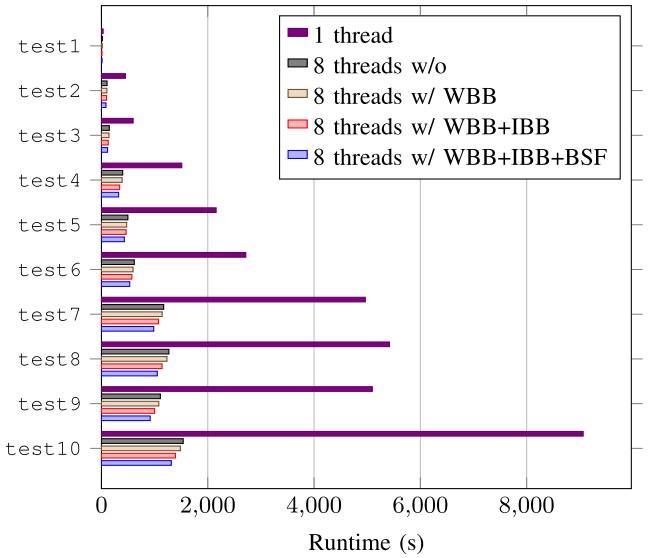

Fig. 13. Speed-up by parallelism.

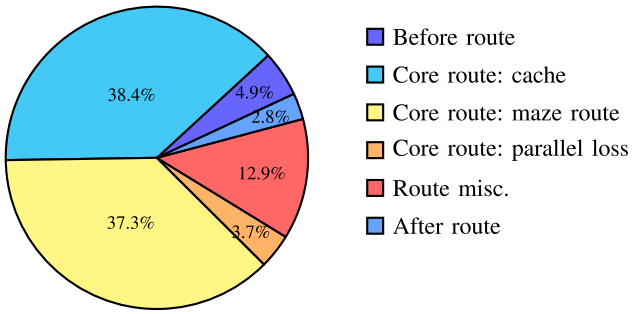

Fig. 14. Runtime breakdown on `ispd18_test10`.

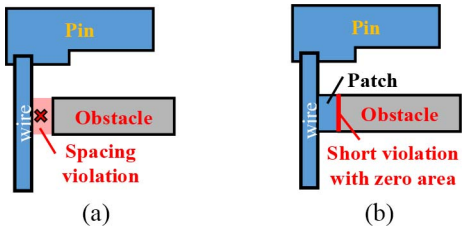

Fig. 15. Spacing-to-short conversion done by some other detailed routers. (a) Spacing violation between a wire segment and an obstacle. (b) Metal patch that converts the spacing violation to a short violation with zero area.

synchronous parallel scheme, where there is a parallel loss.[3] The miscellaneous jobs for routing including the committing phase mentioned in Section V, ripping up violated nets, updating history cost, etc. They take 12.9%. After routing, we write the routing solution to the output file.

C. Comparison With State-of-the-Art Detailed Routers

We also compare Dr. CU with TritonRoute [21], the work [22], and the first place in ISPD 2018 Contest (Table IV).

[3]We divide the total CPU time for caching by the number of threads to get *the equivalent wall time for caching*. Similarly, there is the *equivalent wall time for maze routing*. The parallel loss of core routing is therefore the real wall time of while core routing process minus the equivalent wall time for caching and maze routing.

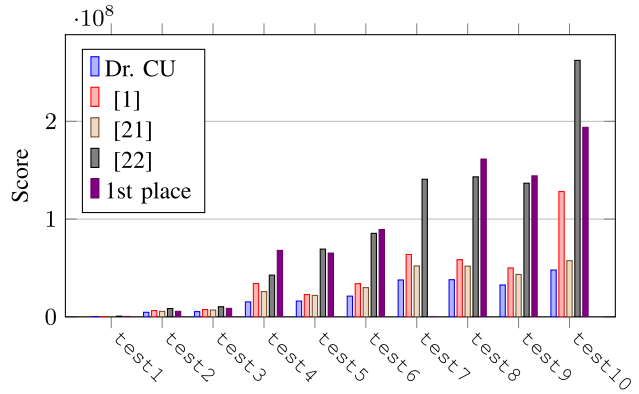

Fig. 16. Comparison with the state-of-the-art detailed routers on quality score under the metric of ISPD 2018 contest.

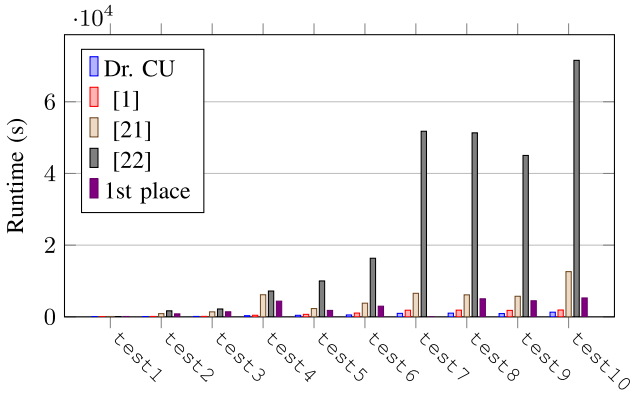

Fig. 18. Comparison with state-of-the-art detailed routers on runtime.

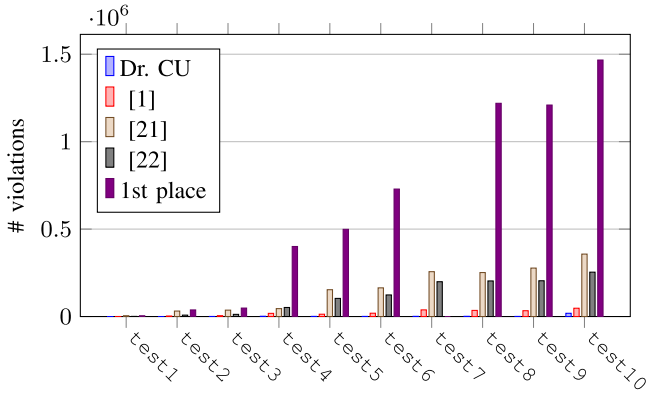

Fig. 17. Comparison with state-of-the-art detailed routers on total number of design rule violations.

For all the detailed routers, we run the binaries provided by the authors on our machine with eight threads. Besides the ISPD 2018 Contest metric, we also report the number of short violations. This is to avoid the misleading due to the abusing of the contest metric. In the design rule verification of Innovus, a spacing violation [e.g., Fig. 15(a)] can be removed by inserting a metal patch between the two violating objects [e.g., Fig. 15(b)]. The patch generates a short violation with zero area, which improves the score under the contest metric but is not beneficial to the real design need.

Regarding the routing quality, Dr. CU shows significantly better scores in many aspects (including wire length, via count, out-of-guide usage, off-track usage, and design rule violations) in most cases. According to the metric of ISPD 2018 contest, our routing quality wins all the other state-of-the-art detailed routers in all test cases, as Fig. 16 summarizes. Compared with the second best [21], the score is improved by 16.6%–40.7%. Regarding the number of design rule violations, our strength is even more obvious (better by one or two orders of magnitude), as Fig. 17 shows. Compared with the second best [22], the number is reduced by 92.7%–99.7%. At the same time, the runtime of Dr. CU is also tremendously better (Fig. 18). To be more specific, there is 9.2×, 33×, and 7.6× speed-up on average compared with [21], [22], and the first place, respectively.

VII. Conclusion

In this paper, we proposed Dr. CU, an efficient and effective detailed router, to tackle the challenges in detailed routing. A set of two-level sparse data structures is designed for the routing grid graph of enormous size. An optimal path search algorithm is proposed to handle the minimum-area constraint. Besides, an efficient bulk synchronous parallel scheme is adopted to further reduce the runtime usage. Compared with the state-of-the-art detailed routers, Dr. CU shows superior routing quality, runtime, and memory usage.

Acknowledgment

The authors would like to thank L. Wang, B. Xu, F.-K. Sun and the other authors of the works [21], [22] for sharing with them the binaries of their detailed routers.

References

[1] 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," in *Proc. IEEE/ACM Asia South Pac. Design Autom. Conf. (ASPDAC)*, 2019, pp. 754–760.

[2] Y. Zhang and C. Chu, "RegularRoute: An efficient detailed router applying regular routing patterns," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 21, no. 9, pp. 1655–1668, Sep. 2013.

[3] S. Mantik, G. Posser, W.-K. Chow, Y. Ding, and W.-H. Liu, "ISPD 2018 initial detailed routing contest and benchmarks," in *Proc. ACM Int. Symp. Phys. Design (ISPD)*, 2018, pp. 140–143.

[4] J. A. Roy and I. L. Markov, "High-performance routing at the nanometer scale," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 6, pp. 1066–1077, Jun. 2008.

[5] Y. Xu, Y. Zhang, and C. Chu, "FastRoute 4.0: Global router with efficient via minimization," in *Proc. IEEE/ACM Asia South Pac. Design Autom. Conf. (ASPDAC)*, 2009, pp. 576–581.

[6] M. Cho, K. Lu, K. Yuan, and D. Z. Pan, "BoxRouter 2.0: A hybrid and robust global router with layer assignment for routability," *ACM Trans. Design Autom. Elect. Syst.*, vol. 14, no. 2, p. 32, 2009.

[7] M. M. Ozdal and M. D. F. Wong, "Archer: A history-based global routing algorithm," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 4, pp. 528–540, Apr. 2009.

[8] T.-H. Wu, A. Davoodi, and J. T. Linderoth, "GRIP: Global routing via integer programming," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 1, pp. 72–84, Jan. 2011.

[9] M. Gester, D. Müller, T. Nieberg, C. Panten, C. Schulte, and J. Vygen, "BonnRoute: Algorithms and data structures for fast and good vlsi routing," *ACM Trans. Design Autom. Elect. Syst.*, vol. 18, no. 2, p. 32, 2013.

[10] 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 Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 32, no. 5, pp. 709–722, May 2013.

[11] 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 Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 32, no. 10, pp. 1557–1568, Oct. 2013.

[12] M. Ahrens *et al.*, "Detailed routing algorithms for advanced technology nodes," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 4, pp. 563–576, Apr. 2015.

[13] M. M. Ozdal, "Detailed-routing algorithms for dense pin clusters in integrated circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 3, pp. 340–349, Mar. 2009.

[14] T. Nieberg, "Gridless pin access in detailed routing," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, New York, NY, USA, 2011, pp. 170–175.

[15] X. Xu, Y. Lin, V. Livramento, and D. Z. Pan, "Concurrent pin access optimization for unidirectional routing," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, Austin, TX, USA, 2017, p. 20.

[16] Q. Ma, H. Zhang, and M. D. F. Wong, "Triple patterning aware routing and its comparison with double patterning aware routing in 14nm technology," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, San Francisco, CA, USA, 2012, pp. 591–596.

[17] Y.-H. Lin, B. Yu, D. Z. Pan, and Y.-L. Li, "TRIAD: A triple patterning lithography aware detailed router," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2012, pp. 123–129.

[18] Z. Liu, C. Liu, and E. F. Y. Young, "An effective triple patterning aware grid-based detailed routing approach," in *Proc. IEEE/ACM Design Autom. Test Europe (DATE)*, 2015, pp. 1641–1646.

[19] Y. Ding, C. Chu, and W.-K. Mak, "Self-aligned double patterning-aware detailed routing with double via insertion and via manufacturability consideration," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 3, pp. 657–668, Mar. 2018.

[20] Y.-H. Su and Y.-W. Chang, "VCR: Simultaneous via-template and cut-template-aware routing for directed self-assembly technology," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2016, pp. 1–8.

[21] A. B. Kahng, L. Wang, and B. Xu, "TritonRoute: An initial detailed router for advanced VLSI technologies," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, San Diego, CA, USA, 2018, pp. 1–8.

[22] 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," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2018, pp. 1–7.

[23] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proc. ACM Conf. Manag. Data (SIGMOD)*, 1984, pp. 47–57.

[24] C. Albrecht, "Global routing by new approximation algorithms for multicommodity flow," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 5, pp. 622–632, May 2001.

[25] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[26] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, MA, USA: MIT Press, 2009.

[27] C. J. Alpert, D. P. Mehta, and S. S. Sapatnekar, Eds., *Handbook of Algorithms for Physical Design Automation*. London, U.K.: CRC Press, 2008.

[28] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[29] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Commun. ACM*, vol. 8, no. 9, p. 569, 1965.

[30] *Boost Geometry Library*. Accessed: Jul. 17, 2019. [Online]. Available: https://www.boost.org/doc/libs/1_67_0/libs/geometry/doc/html/

[31] G. Flach, M. Fogaça, J. Monteiro, M. Johann, and R. Reis, "Rsyn: An extensible physical synthesis framework," in *Proc. ACM Int. Symp. Phys. Design (ISPD)*, 2017, pp. 33–40.

[32] *Cadence Innovus Implementation System*. Accessed: Jul. 17, 2019. [Online]. Available: https://www.cadence.com

[33] *ISPD 2018 Contest*. Accessed: Jul. 17, 2019. [Online]. Available: http://www.ispd.cc/contests/18/

Gengjie Chen received the B.Sc. degree from the Department of Electronic and Communication Engineering, Sun Yat-sen University, Guangzhou, China, in 2015. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, Chinese University of Hong Kong, Hong Kong.

His current research interests include combinatorial optimization, numerical optimization, and electronic design automation.

Mr. Chen was a recipient of the Best Paper Award at ICCAD 2017, the Hong Kong Ph.D. Fellowship from 2015 to 2019, and seven ICCAD/ISPD contest awards. He received the First Place at ACM Student Research Competition in 2019 and the First Place at ACM SIGDA Student Research Competition in 2018.

Chak-Wa Pui received the B.Sc. degree in computer science and technology from Shanghai Jiao Tong University, Shanghai, China, in 2015. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, Chinese University of Hong Kong, Hong Kong.

His current research interests include placement and routing for both ASICs and field-programmable gate arrays.

Mr. Pui was a recipient of the two Best Paper Award nominations in DAC and ISPD and four ISPD/ICCAD contest awards.

Haocheng Li received the B.Eng. degree from the Department of Information and Communication Engineering, Xi'an Jiaotong University, Xi'an, China, in 2016. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, Chinese University of Hong Kong, Hong Kong.

His current research interests include electronic design automation, hardware security, and combinatorial optimization.

Evangeline F. Y. Young received the B.Sc. degree in computer science from the Chinese University of Hong Kong (CUHK), Hong Kong, and the Ph.D. degree from the University of Texas at Austin, Austin, TX, USA, in 1999.

She is currently a Professor with the Department of Computer Science and Engineering, CUHK. Her current research interests include optimization, algorithms, and VLSI CAD. She researches actively on floorplanning, placement, routing, and DFM and EDA on physical design in general.

Dr. Young's research group has won best paper awards from ICCAD 2017, ISPD 2017, SLIP 2017, and FCCM 2018, and several championships and prizes in renown EDA contests, including the 2018, 2016, 2015, 2013, and 2012 CAD Contests at ICCAD, DAC 2012, and ISPD 2019, 2018, 2017, 2016, 2015, 2011, and 2010. She has served on the organization committees of ISPD, ARC, and FPT and on the program committees of conferences, including DAC, ICCAD, ISPD, ASP-DAC, SLIP, DATE, and GLSVLSI. She also served on the editorial boards of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, *ACM Transactions on Design Automation of Electronic Systems*, and *Integration, the VLSI Journal*.