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

Andrew B. Kahng, *Fellow, IEEE,*
Lutong Wang, *Student Member, IEEE,* and Bangqi Xu, *Student Member, IEEE*

***Abstract*—Routing is a crucial stage in a modern design automation tool flow for advanced technology nodes. Works in the recent open literature tend to divide routing into separate global routing and detailed routing steps without addressing the correlation issues (e.g., local nets) between these two steps. In this work, we present TritonRoute-WXL, a unified global-detailed router capable of delivering DRC-clean routing solutions in commercial sub-16nm technologies. The major contributions of TritonRoute-WXL include an end-to-end routing framework that closely connects global routing and detailed routing, and an improved detailed routing flow. With a code release under a permissive open source license, TritonRoute-WXL achieves unparalleled solution quality in terms of detailed routing and global-detailed routing as compared to known best solutions from all published academic routers.**

I. Introduction

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 complex design rules. These complex design rules introduce ever-greater challenges to routing, especially detailed routing. The key element for detailed routing to comprehend such complex design rules is a design rule check (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 older 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 forms standard cell pins which are intended to be accessed by planar (i.e., in-plane) wires. The resulting mix of via accesses and planar accesses for standard cell pins introduces extra challenges for 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 and detailed routing. 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 published 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, in practice it is essential to have an accurate routing resource model that well reflects multiple 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 [21] and ISPD-2008 [20] global routing contests, along with the recent ICCAD-2019 global routing contest [9], have stimulated research efforts on global routing. The ISPD-2018 [19] and ISPD-2019 [16] 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

A. B. Kahng is with the Departments of Computer Science and Engineering, and of Electrical and Computer Engineering, University of California at San Diego, La Jolla, CA, 92093 USA (email: abk@ucsd.edu).

B. Xu is with the Department of Electrical and Computer Engineering, University of California at San Diego, La Jolla, CA, 92093 USA (e-mail: bangqixu@ucsd.edu).

L. Wang is with Cadence Design Systems, San Jose, CA, 95134 USA (e-mail: lutong@cadence.com).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3079268, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

can be directly evaluated with a usable routing flow instead of academic contest-centric evaluation metrics. Towards this end, in this paper we present TritonRoute-WXL, an open-source router for advanced VLSI technologies with integrated DRC engine. 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 (GR) and detailed routing (DR) 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 II provides a brief overview of previous works in the open literature. Section III presents our overall routing flow. Section IV details our global routing methodology. Section V presents our geometry based design rule check engine (DRC engine). Section VI presents our improved detailed routing flow. Section VII presents our experimental results using the official ISPD-2018 and ISPD-2019 benchmark suites. Section VIII concludes our work.

II. Previous Works

We classify relevant previous works on routing into three categories: (i) fundamental routing algorithms, and recent developments in (ii) global routing, and (iii) detailed routing.

Fundamental routing algorithms. Lee's algorithm [13] is the first breadth-first maze search algorithm that guarantees to find a minimum-cost path for a two-terminal routing problem if such a path exists. A* search [22], and its bi-directional form [24], perform maze search focusing the direction towards the destination, hence reducing the effort to find the minimum-cost path. Kahng et al. [12] survey more recent developments in conventional and fundamental routing algorithms.

Recent developments in global routing. Many works have been developed based on the ISPD-2007 [21] and ISPD-2008 [20] global routing contests. NCTU-GR 2.0 [15], NTHU-Route 2.0 [4], NTUgr [5] and FastRoute 4.0 [30] 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 [26] performs global routing on a 3D graph based on Discrete Lagrange Multiplier. GRIP [27] applies integer programming to solve the global routing problem. MGR [29] adopts a multi-level approach to more efficiently explore the large routing solution space. The recent ICCAD-2019 global routing contest [9] evaluates a global routing solution by assessing the corresponding detailed routing solution, to accurately capture routability from a detailed routing perspective. CUGR [17], 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. [25] and Zhou et al. [31] build multivariate regression models. Chan et al. [3] adopts support vector machine for DRC distribution prediction. Xie et al. [28] train a fully convolutional network for such prediction. Recently, Chen et al. [6] propose a fully convolutional network-based plug-in to optimize global routing solutions, thus reducing post-route DRCs.

Recent developments in detailed routing. The ISPD-2018 [19] and ISPD-2019 [16] initial detailed routing contests have stimulated new academic efforts to address the detailed routing problem, using industrial detailed challenges and benchmarks. Kahng et al. [12] survey recent ISPD contest-based works on detailed routing, and present a detailed router that adopts a region-based ripup-and-reroute methodology with comprehensive cost scheme for design rule awareness. We perform routing in DRC-safe, non-overlapping regions in parallel in our present work. Gonçalves et al. [10] present an interval based path search algorithm with design rule awareness.

III. Flow

In this section, we describe our global-detailed routing flow. As shown in Figure 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. Importantly, 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. As compared to [12], the global routing step is new and the detailed routing step is significantly improved thanks to the optimizations enabled by our DRC engine.

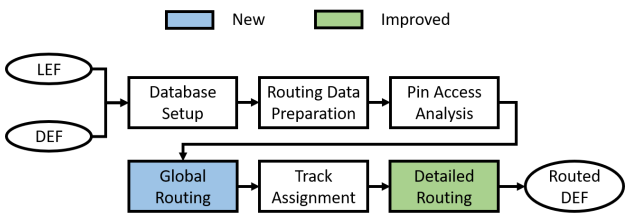

Fig. 1: Overall global-detailed routing flow.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3079268, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

IV. Global Routing

In this section, we describe our global routing flow that operates on the GCells level. As shown in Figure 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.

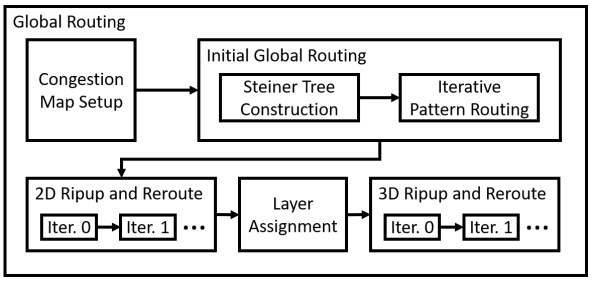

Fig. 2: Global routing flow.

A. 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 [2], 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.

1) Supply: The 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 3. Therefore, for routing layers (e.g., Metal6 in Figure 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.

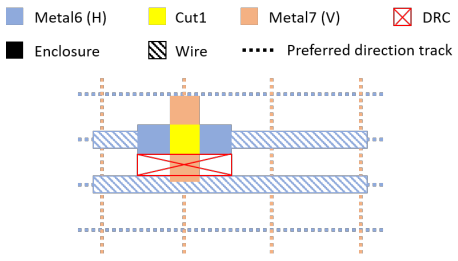

Fig. 3: Illustration of optimism in supply calculation for routing layer with wide via enclosure.

2) 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 [11] 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 illustrates our unified resource model. Figure 4(a) shows the layout within a GCell that consists of two routing wires and a cell pin. Figure 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.

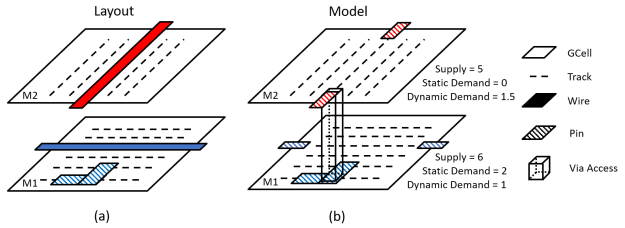

Fig. 4: Illustration of unified routing resource model: (a) the layout of a pin and a routing wire inside a GCell; and (b) the corresponding modeled routing resource with boundary pins and via access.

3) 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 layers to the 2D plane. A GCell is considered as blocked if all of its corresponding GCells in the 3D congestion map are blocked.

B. Initial global routing

The initial global routing consists of (i) Steiner tree construction and (ii) iterative pattern routing. We use FLUTE [7] 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.

C. 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.

1) Region-based ripup-and-reroute: In each *outer* iteration, we partition the design into non-overlapping, 200×200-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 1 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 (details are as given in [12]) 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.

Algorithm 1 Global routing flow

```
1: Input: congestion threshold congThres
2: WorkerDBInit()
3: while currIter < maxIter do
4:     for all net ∈ 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()
```

2) 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 1. The overall relation among wirelength cost, congestion cost and history cost is inspired by [18].

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

$$cost_{cong}(i) = \frac{demand(i)/(supply(i)+1)}{(1+e^{supply(i)-demand(i)})} \quad (2)$$

$$cost_{hist}(i) = histCnt(i) \cdot cost_{cong}(i) \quad (3)$$

$$cost_{overflow}(i) = \begin{cases} 1, & \text{if } demand(i) \geq supply(i) \\ 0, & \text{otherwise} \end{cases} \quad (4)$$

We adopt similar congestion cost function from [17]. 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 are seen in [4] [15] [26]. We adopt similar history cost from [18] and we use a history cost counter *histCnt* and increment the counter each time an overflow is encountered. The history cost counter *histCnt* is decayed (i.e., multiplied by a fractional value less than one) after each iteration. The weights of the cost components are chosen to achieve the following order for a blocked and overflowed GCell: $w_1 \cdot cost_{cong} < w_2 \cdot cost_{hist} < w_4 \cdot cost_{overflow} \ll w_3 \cdot cost_{block}$.

D. Layer assignment and 3D ripup-and-reroute

We adopt a simplified version of dynamic programming based layer assignment from [8]. We sort nets that need layer assignment using the score function from Equation 5 as a "flexibility" measurement which is similar to the one in [30].

$$Score(net) = \frac{HPWL(net)}{|pins(net)|} \quad (5)$$

Note that although an overflow-free 3D routing solution can be constructed based on an overflow-free 2D routing solution using layer assignment [26], layer assignment solution refinement is usually still desired to further improve the solution quality. Unlike the iterative reassignment approach in previous works (e.g., [8]), 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.

V. Geometry-Based Design Rule Check Engine

In this section, we describe our integrated, geometry-based design rule checker (GC).

A. 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 four 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.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3079268, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

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 supports polygon boolean operations (intersecting and merging).

B. Design Rules

The (industry-standard) LEF syntax [34] seen in the ISPD-2018 [19] and the ISPD-2019 [16] benchmark suites is summarized in Table I, where each *italic* word indicates a numerical value. Note the separate metal and cut layer rules.

TABLE I: 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 ;]
```

Minimum width rule specifies the minimum width for a polygon. After 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 this rule is not specified, then we automatically generate one **minimum width** rule per metal layer using the *defaultWidth*. Figure 5 shows polygon slicing and the critical dimension to check against *minWidth*.

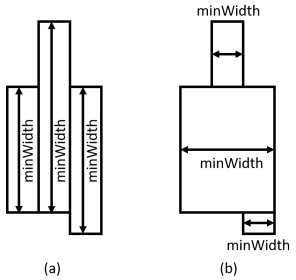

Fig. 5: Minimum width: (a) polygon sliced vertically; and (b) polygon sliced horizontally.

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

Non-sufficient-metal overlap rule specifies the minimum diagonal length in case of metal overlaps. If two max rectangles of the same net overlap, then the overlapping rectangle (i.e., the intersection) must have diagonal length greater than or equal to *minWidth*, as shown in Figure 6.

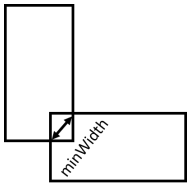

Fig. 6: Non-sufficient-metal overlap.

Parallel run length (PRL) spacing rule 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 7 illustrates the spacing for both positive and negative PRLs.

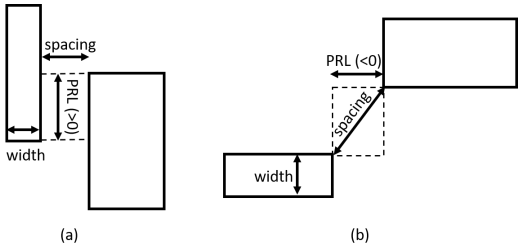

Fig. 7: Parallel run length spacing: (a) positive PRL; and (b) negative PRL.

Minimum step rule 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.

End-of-line (EOL) spacing rule 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 a polygon must be greater than or equal to *eolSpacing* anywhere within (less than) *eolWithin*, as shown in Figure 8(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 8(b).

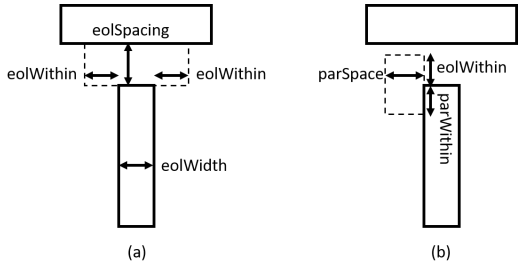

Fig. 8: End-of-line spacing: (a) illustration of *eolWidth*, *eolWithin* and *eolSpacing*; and (b) illustration of *parWithin* and *parSpace*.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3079268, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

Corner spacing rule specifies the spacing from a corner to the exterior of a polygon. CONVEXCORNER (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 rule specifies a short if the two cuts overlap.

Cut spacing rule specifies the minimum spacing between two cuts. If CENTERTOCENTER is specified, then cutSpacing and cutWithin are calculated from cut center to cut center; otherwise, these values are 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*.

C. Data structure

In this subsection, we describe the data structures in GC.

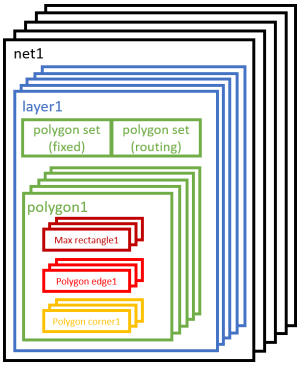

Fig. 9: DRC engine data structure of layout information.

1) Data Structure: Figure 9 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 the boolean OR operation to *non-router-created* shapes. *Polygon set (routing)* is generated similarly from *router-created* shapes. We then merge the two polygon sets into one set, and decompose it into disjoint polygons. Each disjoint polygon holds all of its max rectangles, polygon edges and corners. Each max rectangle, polygon edge or polygon corner is marked with either *fixed* status or *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*.

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 in the number of vertices of all geometries.

2) Region Query: After initialization of data structures, we build region queries for max rectangles and polygon edges. Note that we do not need a separate region query for polygon corners because polygon corners can be queried based on polygon edges.

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 query.

D. Design Rule Checking and Filtering Flow

We now describe the design rule checking and filtering flow. Given an input design database, along with a specified bounding box and layer range, our design rule checking flow first initializes 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 violation bounding box, layer number, net(s) and type.

Input: design database, along with specified bounding box and layer range in which to perform design rule checking
Constraints: design rules
Output: *detailed-routing-fixable* design rule violation markers

The underlying open-source shape engines (e.g., Boost R-tree, which we use) are well-optimized, and further improvement of such shape engines is beyond the scope of this work. For each rule checking algorithm we present below, each early return statement indicates the filtering process, where a match to the rule does not necessarily result in a violation (*corner case*), or the violation is *non-fixable*.

1) Metal Spacing: Metal spacing rules consist of *short* rules, *non-sufficient-metal overlap* rules and *parallel run length spacing* rules. The rule checking starts with a max rectangle m.

Algorithm 2 Check metal spacing

```
1: Input: max rectangle m
2: N ← queryMaxRectangles(m, maxDist)
3: for all n ≠ 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
```

In Algorithm 2, 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 m and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3079268, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

n overlap and belong to the same net, we check non-sufficient metal overlap in Line 6 (details described in Algorithm 4); if m and n overlap but belong to different nets, we check metal short in Line 8 (details described in Algorithm 3); otherwise, we check parallel run length spacing in Line 11 (details described in Algorithm 5).

Algorithm 3 Check metal short

```
1: Input: max rectangles m, n
2: shortRect ← 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 hasRouterCreatedShapes(shortRect) then
10:    return
11: end if
12: addMarker(MetalShort)
```

Algorithm 3 describes the methodology to check short. Line 2 gets the bounding box of metal intersection. In Lines 3–5, we skip the *non-fixable* 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 10. In Lines 9–11, we skip the *non-fixable* violation if *router-created* shapes do not intersect with the short region.

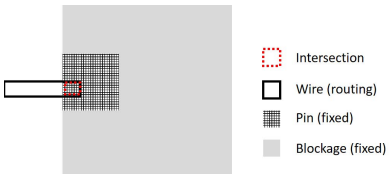

Fig. 10: Metal short filter: short area is within pin.

Algorithm 4 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 *corner case* if any max rectangle has a width less than *minWidth* because such max rectangle is purely the result of polygon decomposition, and does not fully cover any *router-created*, or *non-router-created* shapes. In this work, *minWidth*-related violations are captured by *minWidth* rule checking in Algorithm 6. Note that *minWidth* rule checking is based on sliced rectangles instead of max rectangles. In Lines 9–11, we skip the *corner case* 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 11(a) and (b) illustrate the above two cases.

Algorithm 4 Check non-sufficient metal overlap

```
1: Input: max rectangles m, n
2: nsRect ← getIntersection(m, n)
3: if diagLen(nsRect) ≥ 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)
```

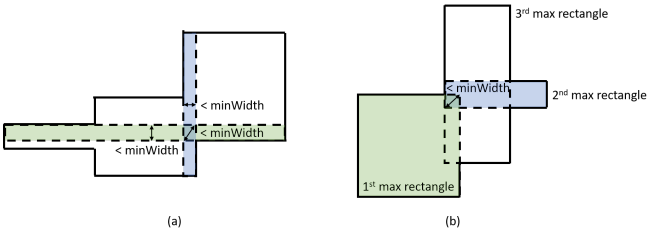

Fig. 11: Non-sufficient-metal overlap filters: (a) max rectangles narrower than *minWidth*; and (b) two max rectangles bridged by a 3rd max rectangle.

Algorithm 5 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 at least one of them is a blockage, 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 *non-fixable* violation if both max rectangles are *fixed*. Line 10 calculates the generalized intersection, i.e., the bounding box formed by the parallel run length and the spacing of the two disjoint max rectangles. In Lines 11–13, we skip the *corner case* 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 are valid. Figure 12(a) shows two same-net max rectangles (light green and light blue) decomposed from a single polygon, with spacing smaller than the required spacing. We skip the *corner case* because in the orthogonal direction of spacing, there is no polygon edge overlapping with the generalized intersecting region. In Lines 14–17, we skip the *non-fixable* violation because *non-router-created* shapes exclusively contribute to the violation. For example, in Figure 12(b), we skip the *non-fixable* violation if no area in the darker green or blue region is exclusively from polygon set (routing).

Algorithm 5 Check parallel run length spacing

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

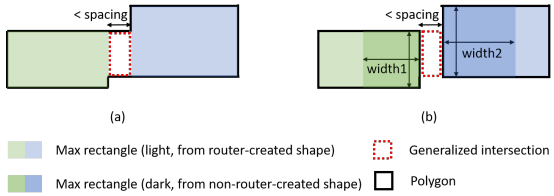

Fig. 12: Parallel run length spacing filters: (a) no valid (vertical) polygon edges overlapped with the generalized intersection, given horizontal spacing direction; and (b) spacing violation contributed exclusively from *non-router-created* shapes.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3079268, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

2) Metal Shape: Metal shape rules consist of minimum width and step. Algorithm 6 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 *non-fixable* violation if the sliced rectangle does not overlap with *router-created* shapes. We repeat the above with slicing in the horizontal direction.

Algorithm 6 Check minimum width (vertical slicing)

```
1: Input: polygon m
2: N ← slicePolygon(m, vertical)
3: for all n in N do
4:     if ySpan(n) ≥ minWidth then
5:         return
6:     end if
7:     if not hasRouting(n) then
8:         return
9:     end if
10:    addMarker(MinimumWidth)
11: end for
```

Algorithm 7 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 of a polygon. 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 17–19, we skip the *corner case* 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 *non-fixable* violation if the bounding box of short edges does not intersect with *router-created* shapes.

Algorithm 7 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 ← e
7: numEdges ← 0
8: while beginEdge ≠ nextEdge(e) do
9:     e ← nextEdge(e)
10:    updateBBox(bbox, endPoint(e))
11:    if length(e) < minStepLength then
12:        numEdges ← 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 hasRouterCreatedShapes(bbox) then
24:    return
25: end if
26: addMarker(MinimumStep)
```

3) End-of-Line Spacing: Algorithm 8 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 11–13, we skip the *corner case* if the generalized intersection of the EOL edge and the opposite edge contains any shape. In Lines 14–16, we skip the *non-fixable* violation if none of the EOL edge, opposite edge, or parallel edge(s), if any, are from *router-created* shapes. Figure 13 shows an EOL violation between two *non-router-created* shapes, given only the existence of a parallel edge from a *router-created* shape.

Algorithm 8 Check end-of-line spacing

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

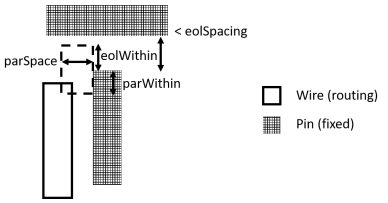

Fig. 13: End-of-line spacing violation between a *non-router-created* EOL edge, and a *non-router-created* opposite edge, given the existence of a parallel edge from a *router-created* shape.

4) Cut Spacing: Check cut spacing follows a similar high-level procedure as shown in Algorithm 2 to first identify neighboring cuts, and then to identify whether two neighboring cuts potentially short or violate cut spacing. Algorithm 9 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 *non-fixable* violation if both cuts are *fixed*. In Lines 10–12, we check whether the layout satisfies ADJACENTCUTS/PARALLELOVERLAP/AREA conditions (if specified).

Algorithm 9 Check cut spacing

```
1: Input: cuts m, n
2: actVal ← getActualSpacing(m, n)
3: reqVal ← getRequiredSpacing(m, n)
4: if actVal ≥ 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) OR not hasParallelOverlap(m, n) OR not hasArea(m, n) then
11:    return
12: end if
13: addMarker(CutSpacing)
```

5) Corner Spacing: Algorithm 10 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3079268, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

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 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 10 Check corner spacing

```
1: Input: polygon corner c
2: if c.type ≠ 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 ← 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 ← getIntersection(c, n)
17:    reqVal ← getRequiredSpacing(n.width)
18:    actVal ← maxXY(prlRect)
19:    if actVal ≥ reqVal then
20:        continue
21:    end if
22:    addMarker(CornerSpacing)
23: end for
```

E. 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 (e.g., our queue-based ripup-and-reroute strategy, via swapping, etc.). 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 2–9 does not belong to the modified net. In the following, we denote incremental DRC checking of a *net* with "GC(*net*)" and denote DRC checking for all nets with "GC()".

VI. Improved ripup-and-reroute in detailed routing

We now present our improved ripup-and-reroute methodology in detailed routing. We first illustrate potential inefficiency in existing ripup-and-reroute flow. We then describe our queue-based ripup-and-reroute flow that improves DRC convergence.

A. Inefficiency in existing ripup-and-reroute flow

Use of ripup-and-reroute to resolve DRC can rely heavily on net ordering. Figure 14 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 14. Hence, simple net ordering heuristics (e.g., shuffling) may not efficiently converge to a feasible solution. Figure 14(a) illustrates how ripup-and-reroute flows in our [12] 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.

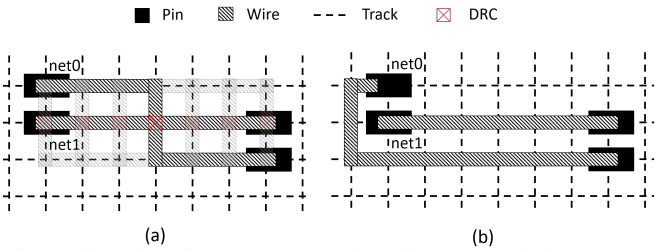

Fig. 14: Illustration of DRC convergence depending on net ordering: (a) seven routing solutions with DRC and (b) a DRC-clean routing solution.

B. 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 Section V-E. 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. The next paragraphs discuss details of our ripup-and-reroute queue.

To illustrate how we push nets to the queue, we introduce the concept of **aggressor** and **victim**. Recall that in the Figure 14, *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 11. Line 2 first initializes the worker database. Line 3 initializes the ripup-and-reroute *queue* with existing DRC

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3079268, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

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 11 Queue-based routing flow

```
1: Input: database, DRC markers markers
2: WorkerDBInit()
3: queue.update(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 15 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 15(e) and (f), for example). Figure 15(a) shows that the three nets are initially routed in the order of *net0*, *net1* and *net2*. Figure 15(b) illustrates the layout and *queue* after *net0* is routed. Figure 15(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 15(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 15(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 15(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.

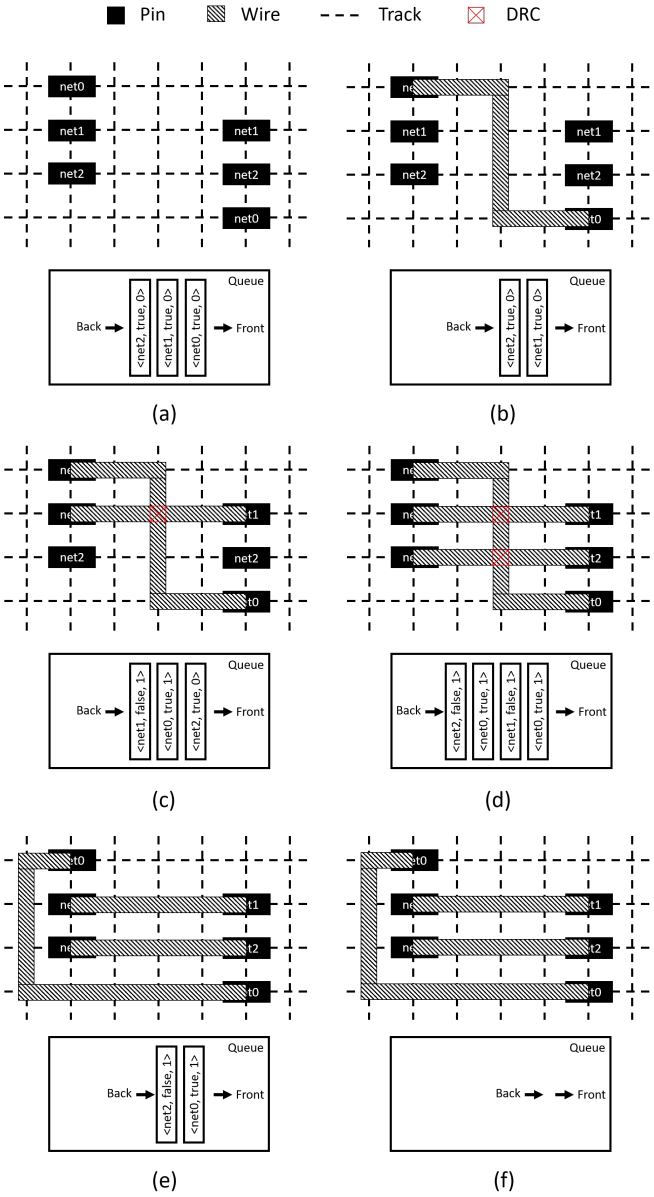

Fig. 15: Illustration of ripup-and-reroute queue on three two-pin nets.

C. Ripup-and-reroute queue update

Algorithm 12 describes the procedure to populate the ripup-and-reroute *queue* based on a given list of DRC markers. 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 markers, 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.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3079268, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

Algorithm 12 Update ripup-and-reroute queue

```
Input: ripup-and-reroute queue queue, DRC markers markers
uniqueAggressors = ∅
uniqueVictims = ∅
for all marker ∈ markers do
    for all aggressor ∈ marker.getAggressors() do
        uniqueAggressors.insert(aggressor)
    end for
    uniqueVictims.insert(marker.getVictim())
end for
for all aggressor ∈ uniqueAggressors do
    queue.push_back(<aggressor, true, 0>)
end for
for all victim ∈ uniqueVictims do
    queue.push_back(<victim, false, 0>)
end for
```

VII. Experiments

We implement our router in C++ with LEF/DEF parser [34] and Boost C++ libraries [32]. We enable multi-threading with OpenMP [35]. We perform experiments using the ISPD-2018 and ISPD-2019 benchmark suites [16][19] 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 which make the testcases more challenging and closer to real-world routing problems. We summarize the benchmark information in Table II. Additionally, we perform an experiment with a foundry 14nm technology node and a commercial 14nm library.

TABLE II: Benchmark information [16][19].

Benchmark	#std	#blk	#net	#pin	#layer	Tech.
ISPD-2018						
ispd18_test1	8879	0	3153	0	9	$45nm$
ispd18_test2	35913	0	36834	1211	9	$45nm$
ispd18_test3	35973	4	36700	1211	9	$45nm$
ispd18_test4	72094	0	72401	1211	9	$32nm$
ispd18_test5	71954	0	72394	1211	9	$32nm$
ispd18_test6	107919	0	107701	1211	9	$32nm$
ispd18_test7	179865	16	179863	1211	9	$32nm$
ispd18_test8	191987	16	179863	1211	9	$32nm$
ispd18_test9	192911	0	178857	1211	9	$32nm$
ispd18_test10	290386	0	182000	1211	9	$32nm$
ISPD-2019						
ispd19_test1	8879	0	3153	0	9	$32nm$
ispd19_test2	72094	4	72410	1211	9	$32nm$
ispd19_test3	8283	4	8953	57	9	$32nm$
ispd19_test4	146442	7	151612	4802	5	$65nm$
ispd19_test5	28920	6	29416	360	5	$65nm$
ispd19_test6	179881	16	179863	1211	9	$32nm$
ispd19_test7	359746	16	358720	2216	9	$32nm$
ispd19_test8	539611	16	537577	3221	9	$32nm$
ispd19_test9	899341	16	895253	3221	9	$32nm$
ispd19_test10	899404	16	895253	3221	9	$32nm$

In the following, based on the ISPD-2018 and ISPD-2019 benchmark suites, we perform (i) DRC convergence comparison between detailed routing (DR) results with and without ripup-and-reroute queue, (ii) comparison between our DR work and known best DR 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 [23] in 14nm. All experiments are performed using eight threads on an Intel Xeon 2.4GHz server.

A. Queue-based ripup-and-reroute DRC convergence study

In this subsection, we compare the detailed routing based on ISPD-2018 and ISPD-2019 benchmark testcases using our detailed router with and without the ripup-and-reroute queue enablement that is described in Section VI-B. For the version that is without ripup-and-reroute queue, we use the ripup-and-reroute strategy in [12] while keeping 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 III 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 testcases where both versions converge on DRC, ripup-and-reroute queue can reduce runtime by an average of 33.5% (up to 85.4%). Since the detailed routing runtime is closely proportional to the overall number of ripup-and-reroutes, the queue-based ripup-and-reroute strategy's more adaptive control on net ordering can achieve DRC convergence more efficiently.

B. DR comparison to known best solutions

In this subsection, we compare our detailed routing (DR) solution 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) [12]. For all ISPD-2019 benchmark testcases, the known best DR solutions are from Dr. CU 2.0 (CU) [14]. Table IV 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.

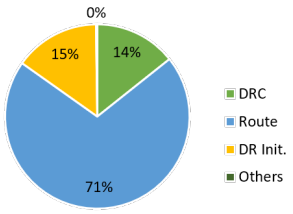

Fig. 16: Detailed routing runtime breakdown.

We now discuss the runtime and multithread scalability of our current work. For runtime study, Figure 16 illustrates the breakdown of the overall detailed routing runtime of four parts – initialization, routing, design rule checking and others. For multithread scalability study, we measure both single-thread and eight-thread runtime and calculate the eight-thread (8T) speedup. Table IV 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) [12] does not have multithreading support and Dr. CU 2.0 (CU) [14] has multithreading capability.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3079268, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

TABLE III: Detailed routing comparison of wirelength, via count, DRC count and runtime between TritonRoute-WXL (TR-WXL) and 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

TABLE IV: 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. For 8T speedup ($\times$) over 1T, note that ispd18_test1–ispd18_test10 results are from [12] without multithreading support and ispd19_test1–ispd19_test10 results are from [14] with multithreading capability.

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

C. 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. Note that the ISPD Contest GR solutions are produced from a commercial routing tool [16][19]. The Contests include both high-quality solutions, which "contains DRC-free solution strictly within the GR solution", and low-quality solutions, which "has congestion issue that needs detailed routing to escape from the GR solution to fix DRC violations". Table V 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. Our results suggest that using consistent routing data (e.g., pin access location, pin access layer, etc.) is essential to improve global-detailed routing convergence.

D. GR-DR flow comparison

In this subsection, we compare our global-detailed routing flow to an academic global-detailed routing flow composed of

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3079268, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

TABLE V: 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

TABLE VI: 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

CUGR and Dr. CU 2.0. Table VI 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 better, 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 the CUGR-and-Dr. CU 2.0 flow.

E. 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 [1] in a 14nm foundry technology node using a commercial 14nm library. We perform our experiment using a global routed RISC-V core [23] (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 17 shows the layout of the routed design.

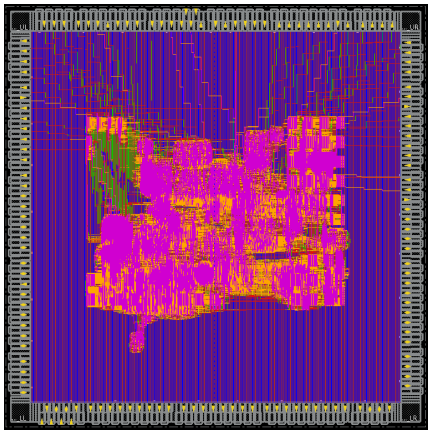

Fig. 17: Illustration of DRC-clean routing of a RISC-V core in 14nm.

VIII. Conclusion and Future Work

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%. Besides fulfilling the future works in [12], we present an end-to-end global-detailed routing flow. 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 solution for sub-16nm foundry technology nodes. Our future research directions include (i) more sophisticated global routing net ordering and (ii) topology control during ripup-and-reroute in global routing.

IX. Acknowledgments

We thank Dr. Wen-Hao Liu for providing valuable feedback.

References

[1] 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. DAC*, 2019, pp. 76:1-76:4.

[2] C. J. Alpert, M. D. Moffitt, G. J. Nam, J. A. Roy and G. Tellez, "What Makes a Design Difficult to Route", *Proc. ISPD*, 2014, PP. 7-12.

[3] 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. ISPD*, 2017, pp. 15-21.

[4] Y.-J. Chang, Y.-T. Lee and T.-C. Wang, "NTHU-Route 2.0: A Fast and Stable Global Router", *Proc. ICCAD*, 2008, pp. 338-343.

[5] H.-Y. Chen, C.-H. Hsu and Y.-W. Chang, "High-Performance Global Routing with Fast Overflow Reduction", *Proc. ASP-DAC*, 2009, pp. 582-587.

[6] 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. ICCAD*, 2020, pp. 1-8.

[7] C. Chu and Y. Wong, "FLUTE: Fast Lookup Table Based Rectilinear Steiner Minimal Tree Algorithm for VLSI design", *IEEE Trans. on CAD* 27(1) (2008), pp. 70-83.

[8] 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 Trans. on VLSI* 20(3) (2012), pp. 459-472.

[9] S. Dolgov, A. Volkov, L. Wang and B. Xu, "2019 CAD Contest: LEF/DEF Based Global Routing", *Proc. ICCAD*, 2019, pp. 1-4.

[10] 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 Trans. on DAES* 26(2) (2020), article 9.

[11] A. B. Kahng, L. Wang and B. Xu, "The Tao of PAO: Anatomy of a Pin Access Oracle for Detailed Routing", *Proc. DAC*, 2020.

[12] A. B. Kahng, L. Wang and B. Xu, "TritonRoute: The Open Source Detailed Router", *IEEE Trans. on CAD* (2020).

[13] C. Y. Lee, "An Algorithm for Path Connections and Its Applications", *IRE Trans. on Electro. Comp.* 10(3) (1961), pp. 346-365.

[14] 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. ICCAD*, 2019, pp. 1-7.

[15] 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. on CAD* 32(5) 2013, pp. 709-722.

[16] 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. ISPD*, 2018, pp. 140-143.

[17] J. Liu, C.-W. Pui, F. Wang and E. F. Y. Young, "CUGR: Detailed-Routability-Driven 3D Global Routing with Probabilistic Resource Model", *Proc. DAC*, 2020.

[18] L. McMurchie and C. Ebeling, "Pathfinder: A Negotiation-Based Performance-Driven Router for FPGAs", *Proc. ISFPGA*, 1995, pp. 111-117.

[19] S. Mantik, G. Posser, W.-K. Chow, Y. Ding and W.-H. Liu, "ISPD 2018 Initial Detailed Routing Contest and Benchmarks", *Proc. ISPD*, 2018, pp. 140-143.

[20] G.-J. Nam, C. Sze and M. Yildiz, "The ISPD Global Routing Benchmark Suite", *Proc. ISPD*, 2008, pp. 156-159.

[21] G.-J. Nam, M. Yildiz, D. Z. Pan and P. H. Madden, "ISPD Placement Contest Updates and ISPD 2007 Global Routing Contest", *Proc. ISPD*, 2007, pp. 167.

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

[23] 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 and M. B. Taylor, "BlackParrot: An Agile Open Source RISC-V Multicore for Accelerator SoCs", *IEEE Micro* 40(4) (2020), pp. 93-102.

[24] I. Pohl, "Bi-Directional Search", *Machine Intelligence* (1971), pp. 127-140.

[25] Z. Qi, Y. Cai and Q. Zhou, "Accurate Prediction of Detailed Routing Congestion using Supervised Data Learning", *Proc. ICCD*, 2014, pp. 97-103.

[26] J. A. Roy and I. L. Markov, "High-Performance Routing at the Nanometer Scale", *IEEE Trans. on CAD* 27(6) 2008, pp. 1066-1077.

[27] T.-H. Wu, A. Davoodi and J. T. Linderoth, "GRIP: Scalable 3D Global Routing using Integer Programming", *Proc. DAC*, 2009, pp. 320-325.

[28] 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. ICCAD*, 2018, pp. 1-8.

[29] Y. Xu and C. Chu, "MGR: Multi-Level Global Router", *Proc ICCAD*, 2011, pp. 250-255.

[30] Y. Xu, Y. Zhang and C. Chu, "FastRoute 4.0: Global Router with Efficient Via Minimization", *Proc. ASP-DAC*, 2009, pp. 576-581.

[31] Q. Zhou, X. Wang, Z. Qi, Z. Chen, Q. Zhou and Y. Cai, "An Accurate Detailed Routing Routability Prediction Model in Placement", *Proc. ASQED*, 2015, pp. 119-122.

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

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

[34] LEF/DEF reference 5.7. http://www.si2.org/openeda.si2.org/projects/lefdefnew

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

Andrew B. Kahng photo and bio not available.

Lutong Wang received the Ph.D. degree in electrical and computer engineering from the University of California at San Diego, La Jolla, in 2019. He is currently with Cadence Design Systems, Inc. His research interests include physical design implementation and DFM methodologies.

Bangqi Xu received the BSEE degree from the University of Michigan, Ann Arbor, MI, USA in 2015 and the M.S. in ECE from the University of California at San Diego, La Jolla, in 2017. He is currently pursuing the Ph.D. degree at the University of California at San Diego, La Jolla. His interests include detailed placement, routing methodology and optimization.