TritonRoute: The Open-Source Detailed Router

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

***Abstract*—Detailed routing is a dead-or-alive critical element in design automation tooling for advanced node enablement. However, very few works address detailed routing in the recent open literature, particularly in the context of modern industrial designs and a complete, end-to-end flow. The ISPD-2018 Initial Detailed Routing Contest addressed this gap for modern industrial designs, using a realistic design rules set. In this work, we present TritonRoute, a detailed router capable of delivering a DRC-clean routing solution. The key contributions of TritonRoute include an in-memory router database, along with an end-to-end detailed routing scheme that is capable of comprehending connectivity and design rule constraints, with every key detail revealed by a code release under a permissive open-source license. We evaluate our router using the official ISPD-2018 benchmark suite and show that TritonRoute achieves an unprecedented solution quality—improved wirelength and via count, and an extremely low level of design rule violations (DRCs). Compared to the known best detailed routing solutions from all published academic detailed routers, TritonRoute improves wirelength by up to 0.8% (avg. 0.4%), via count by up to 16.1% (avg. 9.3%), and DRCs by up to 100% (avg. 92.0%).**

***Index Terms*—Design automation, detailed routing, open source software, physical design, rip up and reroute, routing.**

I. Introduction

DETAILED routing is a dead-or-alive critical element of advanced node enablement. New technology nodes come with smaller feature sizes, while fundamental physical (lithographic patterning, CMP, reliability, variability, etc.) and circuit (crosstalk, delay, etc.) limitations remain. As a result, ever-more complex design rules must be comprehended and satisfied at the detailed routing stage, greatly challenging routability as well as the architecture and strategy of the detailed router itself.

Due to the high complexity and enormous solution space for the very large-scale integration (VLSI) routing problem, the routing is typically split into global routing and detailed routing stages. In global routing, the routing region is divided into rectangular grid cells and represented using a coarse-grained 3-D routing graph. Capacities and various constraints are assigned to the edges and vertices in this 3-D routing graph so that overall routing topology and layer assignment can be optimized considering routability, timing, crosstalk, power, etc. The ensuing detailed routing stage attempts to realize the segments and vias according to the global routing solution, while minimizing design rule violations.

The detailed routing problem has been extensively studied for more than five decades. The fundamental algorithms (e.g., Lee's algorithm, unidirectional and bidirectional A* search, ripup-and-reroute paradigm, etc.) and problem formulations (e.g., channel routing and switchbox routing) have largely remained intact in commercial tools for several decades; see [3] for a thorough review. These algorithms and formulations are elaborated to meet real-world requirements (design-rule correctness, quality of result, scalability, and turnaround time) and widely deployed in today's commercial tools that support foundry N7, N5, or even N3 nodes.

However, only a few academic works [11] even attempt to present an end-to-end detailed routing flow, and almost no works make claims to viability in the real-world IC physical design (P&R) context. Since most detailed routing research focuses on different objectives, such as crosstalk or new-technology contexts, comparison between these works is difficult. Further, direct application of academic codes to modern industrial benchmarks has many hurdles, especially given that commercial tools and industrial designs satisfy far more, and more complex, design rules than any academic tools.

Given the above, it is a highly significant milestone for the field that the ISPD-2018 contest, on the subject of initial detailed routing, has recently exposed industrial detailed routing challenges and benchmarks to the academic community [25], [38]. The ISPD-2018 benchmark suite provides ten testcases in 45 and 32 nm nodes, with up to 290K standard cells and 182K nets. These designs are industrial benchmarks—including large memory cells, off-track pin access, IO ports, and power and macro blockages—with realistic design rules offered in industry-standard input/output formats while keeping problem complexity tractable to academic researchers within the four-month contest timespan. However, even two full years after the initial release of the ISPD-2018 contest, there are only a few works [4], [5], [13], [18], [22], [31] capable of delivering any kind of result; these results have nearly a thousand, if not thousands, of design rule check violations (DRCs) for nearly every testcase. Up until now, no work has come close to approaching the solution

Manuscript received December 4, 2019; revised March 5, 2020; accepted May 4, 2020. Date of publication June 18, 2020; date of current version February 19, 2021. This work was supported in part by DARPA (HR0011-18-2-0032), in part by the Qualcomm FMA Fellowship, in part by NSF (CCF-1564302), and in part by gifts and grants from Qualcomm, Samsung Electronics, NXP Semiconductors, and Mentor Graphics. This article was recommended by Associate Editor L. Behjat. *(Corresponding author: Bangqi Xu.)*

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

Lutong Wang and Bangqi Xu are with the Department of Electrical and Computer Engineering, University of California at San Diego, La Jolla, CA 92093 USA (e-mail: luw002@ucsd.edu; bangqixu@ucsd.edu).

Digital Object Identifier 10.1109/TCAD.2020.3003234

quality we expect from commercial detailed routers, although almost every work utilizes a variant of the five-decades-old path search algorithm.

Based on the ISPD-2018 Initial Detailed Routing contest, this article describes TritonRoute, an open-source detailed router for advanced VLSI technologies. Our main contribution is an end-to-end (i.e., complete, and with collaterals visible in a permissively open-sourced repository) detailed routing framework that aims and achieves beyond all existing academic detailed routers. Highlights of our work are summarized as follows.

1) We propose an end-to-end detailed routing scheme. Our proposed scheme is capable of comprehending connectivity constraints (i.e., opens and shorts) and design rule constraints [i.e., spacing tables, end-of-line (EOL) spacing, minimum area, and cut spacing].
2) We build an in-memory router database that complies with LEF/DEF data models. This noncontest-driven code infrastructure enables future development and leverage of our open-source code toward deeper core optimization, more complete design rule support, and other enhancements.
3) We present a number of key ideas in addition to the well-known A*-based path search. Transparency of our descriptions is aided by all implementation source codes being released under a permissive open-source license.
4) We evaluate our router using the official ISPD-2018 benchmark suite, and show that we reach an unprecedented, extremely low level of DRCs (<20) in seven of ten testcases, which is a 99.3% reduction of DRCs on average as compared to the known best detailed routing solutions from all published academic detailed routers. For the remaining three testcases, we reduce DRCs by 75.1% on average, and by 60.0% at a minimum. Overall, compared to the known best detailed routing solutions, TritonRoute improves wirelength by up to 0.8% (avg. 0.4%), via count by up to 16.1% (avg. 9.3%), and DRCs by up to 100% (avg. 92.0%).
5) To the best of our knowledge, we are the first and the only open-source gridded detailed router which is capable of delivering a DRC-clean detailed routing solution in sub-65-nm technology nodes.

The remainder of this article is organized as follows. Section II provides a brief overview of previous works in the open literature. As noted above, such literature is sparse as far as it gives insight into industry routing tools and how they address modern routing challenges. Section III presents our router database. Section IV details our overall detailed routing flow. Section V presents our detailed routing methodology. Section VI presents our experimental results using the official ISPD-2018 benchmark suite. Section VII gives conclusions and directions for ongoing work.

II. Previous Works

As surveyed in [3], previous works on detailed routing can be categorized into fundamental and conventional algorithms, and recent developments. Further, we summarize the recent works targeting the ISPD-2018 initial detailed routing contest.

Fundamental and Conventional Algorithms: Lee [20] proposed the first maze routing algorithm, i.e., a breadth-first search that guarantees to find a minimum-cost path between two terminals if a path exists. Use of "best-first search," also known as A* search [27], sometimes in its bidirectional [28] form, enables maze-based search to focus itself toward desired targets, and reduces the effort needed to find a minimum-cost feasible path. Hadlock [14] and Soukup [30] applied speedups to Lee's algorithm and others applied the line-search paradigm [17] to improve time and space efficiency as compared to Lee's and A* algorithms. Hetzel [16] developed a sequential routing approach using a shortest path algorithm with respect to the Euclidean distance. Specialized contexts, such as channel routing [9] and switchbox routing [24], along with general frameworks, such as multicommodity flow [29] and ripup-and-reroute [32], have respective subliterature and remain as a fundamental building blocks of the detailed router today (see [11]).

Recent Developments: More recent academic works on detailed routing focus on certain aspects of the modern routing challenge, mainly to address issues arising with advanced nodes. Leung [21] gave an excellent summary of the academia-industry gap for detailed routing as of 2003; much of this gap remains today. Examples of focused recent works include Nieberg [26], which proposes techniques for gridless pin access in detailed routing. Xu *et al.* [34] proposed pin-access planning and regular routing for self-aligned double patterning (SADP). The works of [6], [8], [10], and [23] address the detailed routing problem in an SADP process context. MANA [2] introduces an end-to-end separation and minimum wirelength-aware shortest path algorithm. Han *et al.* [15] developed a framework to reduce various DRCs in advanced nodes using multicommodity flow-based integer-linear programming (ILP). BonnRoute [1], [11] and RegularRoute [35] are two works prominent in the recent literature that present more complete portraits of overall detailed routing solutions.

ISPD Contest-Based Works: Recently, a few works in the open literature attempt to address the gap between modern industrial designs and academic detailed routing flows, based on the ISPD-2018 initial detailed routing contest [25]. Sun *et al.* [31] presented a multistage ripup-and-reroute flow for detailed routing. Kahng *et al.* [18] proposed an ILP-based parallel intralayer and sequential interlayer routing flow. Chen *et al.* [4], [5] and Li *et al.* [22] proposed a detailed routing flow using min-area-captured path search on a sparse grid graph. Gonçalves *et al.* [12], [13] proposed a tunnel-aware A* lower bound, and a design-rule-aware path search algorithm for detailed routing. Although most recent works use correct-by-construction or safe-by-construction approaches to prevent DRCs, none of them is capable of delivering decent solution quality (i.e., in a practical sense) due to the complexity of developing the necessary router infrastructure.

III. Database

In this section, we list all major objects and structures in the routing database. In building this database, we follow the LEF/DEF [40] data model, and reuse the naming convention from OpenAccess [41] as much as possible. The objects from

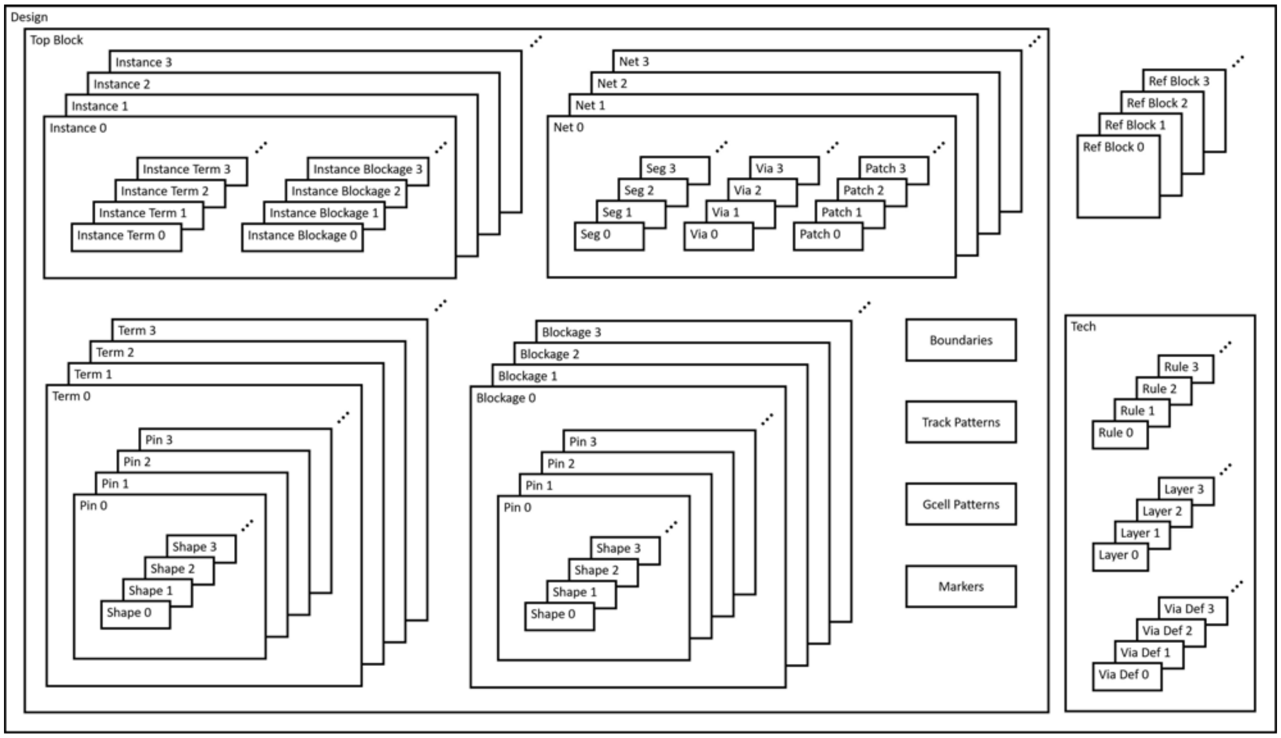

Fig. 1. Major database structures.

TABLE I
DATABASE OBJECTS FROM LEF

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

TABLE II
DATABASE OBJECTS FROM DEF

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

TABLE III
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] ;] ...
// cut layer
{SPACING cutSpacing [CENTERTOCENTER]
 [  ADJACENTCUTS numCuts WITHIN cutWithin [EXCEPTSAMEPGNET]
  | PARALLELOVERLAP
  | AREA cutArea] ;}...
[SPACING cutSpacingSN [CENTERTOCENTER] SAMENET ;]
```

LEF are summarized in Table I, and the objects from DEF are summarized in Table II. The structure of the database is described in Fig. 1. The database is an in-memory, flattened physical design database. In the top level, the database consists of a *technology library*, a *top block*, and several *reference blocks*.

A. Technology Library

Technology library stores all metal and cut *layers*, *viadefs*, and design rule *constraints*. A back-of-end-stack layer consists of basic layer information, i.e., type, direction, pitch, offset, as well as all its applied design rule constraints. A viadef holds one or more *shapes* (rectangles or polygons) on two consecutive metal layers with shape(s) in the middle cut layer, realizing the physical connection between neighboring metal layers at the same *x*–*y* coordinate. We summarize the design rules that we support in Table III. For definitions, examples, and detailed handling methodology of each rule, please refer to [36].

B. Block

The top block describes the flattened logical and physical connections, following the DEF model. There are four major types of objects: 1) *term*; 2) *blockage*; 3) *instance*; and 4) *net*. A reference block is a standard or macro cell from LEF, having the same data structure as the top block, except that only terms and blockages are populated.

1) Term: Terms are IO pins for the top block, and standard or macro cell pins for the reference blocks. Each term consists of one or more physical *pins*. Each pin consists of one or more physical shapes across one or more metal and cut layers.[1]

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

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

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

Other types of objects in a block include *boundary*, *trackPattern*, *gcellPattern*, *marker*, etc. The gcellPattern object defines the global routing cells (GCells) [7] in 2-D grids;[2] and marker object represents a design rule check (DRC) violation, including the bounding box, layer, violation type, and source objects. In our implementation, we also build several assisting objects and structures. Some of the procedures are described in Section IV. A complete picture and details of the database implementation are visible at [37].

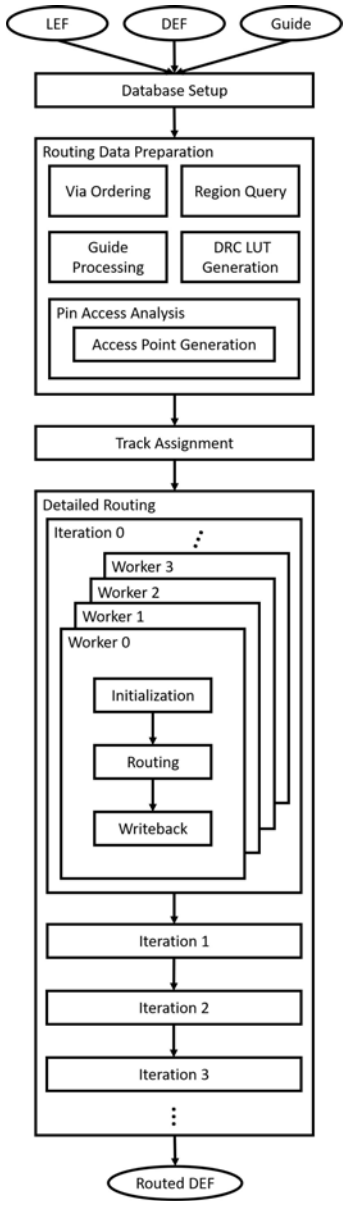

Fig. 2. Overall flow.

IV. Flow

In this section, we describe the detailed routing flow. As shown in Fig. 2, the inputs to the router are LEF, DEF, and guide files. LEF and DEF files are industry-standard formats. The route guide file serves as the global routing solution. Given the inputs, we first set up the design database. Next, we take several data preparation steps. Then, we perform track assignment, multiple iterations of detailed routing, and output a routed DEF.

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

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

A. Data Preparation

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

1) Via Ordering: Via ordering is the step to select default viadef(s) used for pin access and detailed routing. We sort all viadefs according to: 1) the number of cuts; 2) default via property; 3) enclosure direction; 4) enclosure area; and 5) enclosure width. In detailed routing, we only use the minimal-enclosed default single-cut viadef, with both lower- and upper-layer enclosure along the preferred routing direction. In pin access analysis, in addition to the viadef we use in detail routing, we also use the minimal-enclosed default single-cut viadef, with the lower-layer enclosure orthogonal to the preferred routing direction, and the upper-layer enclosure

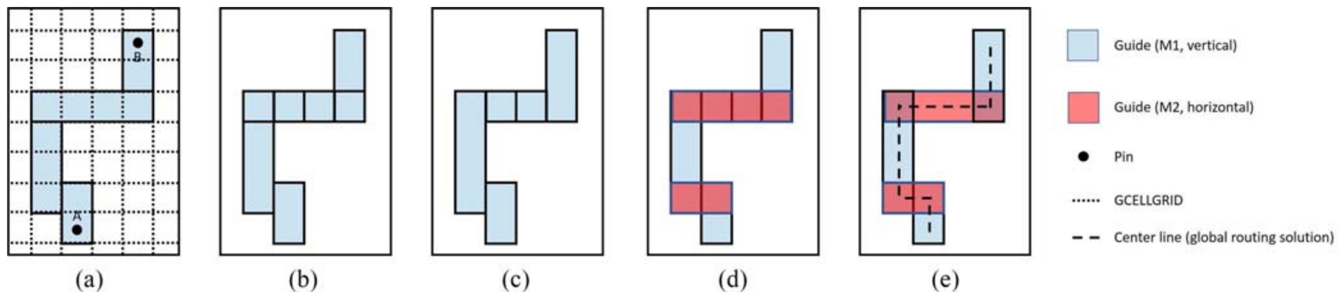

Fig. 3. Preprocessing: (a) initial route guides; (b) splitting; (c) merging; (d) bridging; and (e) preprocessed guides. The preferred direction for M1 is vertical, and for M2 is horizontal.

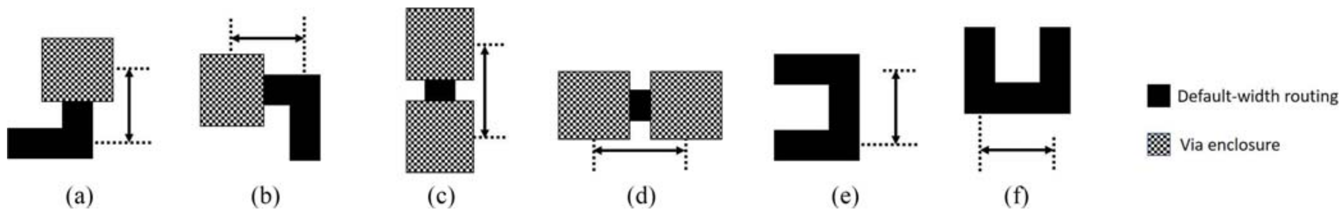

Fig. 4. DRC LUT: (a) via to jog (vertical); (b) via to jog (horizontal); (c) via to via (vertical); (d) via to via (horizontal); (e) jog to jog (vertical); and (f) jog to jog (horizontal).

along the preferred routing direction. Overall, we select one of two viadefs to access the pin, and only use one viadef for all other connections. Fig. 5 illustrates the ordered viadefs for detailed routing, additional viadef for pin access analysis, and a nonpreferred viadef.[3]

2) Guide Processing: Guide processing [7], [18] is the step to transform a set of input route guides into a standardized tree-like global routing solution.[4] A route guide specifies a rectangular region on a specific metal layer. A global routing solution for a net may contain several route guides on some or all of the metal layers. If we abstract the guide by drawing a center line for each guide along the preferred routing direction, we take the center lines to form a connected graph, as shown in Fig. 3(e).

To standardize on a guide dimension that is conducive to form a trimmed tree-like global routing solution, we first extract the most common offset and width of all guides to form GCELLGRIDS [7], then process all route guides with *splitting*, *merging*, and *bridging* techniques. Given the input guides in Fig. 3(a), we first split the guide according to the GCELLGRID along the preferred routing direction for each metal layer, as shown in Fig. 3(b); we then merge touching guides along the preferred routing direction, as shown in Fig. 3(c). Finally, for abutting guides along the nonpreferred routing direction, we bridge them by creating upper-layer (or, otherwise, lower-layer) guides, as shown in Fig. 3(d).

The above procedures guarantee a connected global routing solution as long as the input guides satisfy the assumption described in [7]. To remove redundant edges (i.e., loops) in a global routing solution, we further perform A* search from any pin to all other pins through the processed guides. All off-path guides are removed.

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

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

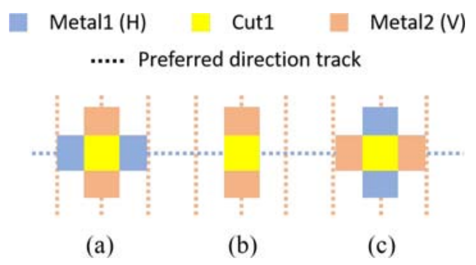

Fig. 5. Illustrations of ordered viadefs: (a) preferred viadef for detailed routing; (b) additional viadef for pin access analysis; and (c) nonpreferred viadef.

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

4) LUT Generation: LUT generation is the step to construct assisting data structure to avoid same-net DRC violations. In grid-based path search, we use the object cost (described in Section V-B) to avoid potential DRCs to existing objects. To prevent DRCs within the current path, i.e., same-net violation, we characterize the minimum default-width routing length between any two-object pair of an up via, a down via, and a jog, on all metal layers, and in all directions. Fig. 4 illustrates three types of minimum length requirement: 1) via to jog; 2) via to via; and 3) jog to jog, in both x and y directions. In our implementation, we characterize separately for the up via and down via. In grid-based path search, we apply an additional cost if the minimum length between vias and/or jogs is not satisfied.

5) Pin Access Analysis: For each pin, we generate at least K access points ($K = 3$ in our implementation) using the pin access analysis methodology from [19]. An access point is an x–y coordinate on a metal layer where the detailed router ends routing. Each access point stores from which direction the router can access the pin. There are six access directions: 1) west; 2) east; 3) south; 4) north; 5) up; and 6) down. For the planar four directions, we check whether a wire can be used to access the pin DRC-free. For the up direction, we check whether the first two vias according to the via ordering can be used to access the pin DRC-free. We do not check or use the down access direction in this work. Each access point may indicate multiple valid access directions. For the up direction, we also store which vias are valid to use, among which one via is primary (preferred to use). The access point must be on the pin shape.

B. Track Assignment

We adopt a simplified version of the greedy track assignment [33]. To reduce the problem size and lay a foundation for future parallel implementation, we perform the track assignment every 50 GCell panels. Each GCell panel has a length along the preferred routing direction and spans 50 GCell heights. The initial track assignment is applied once on all horizontal layers, then on all vertical layers. According to [33], we then perform one iteration of track reassignment to optimize the solution quality.

C. Detailed Routing

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

In the construction of a detailed routing worker, each clip comes with three bounding boxes: 1) *standard*; 2) *DRC*; and 3) *extended box*. The standard box is the above-mentioned 7×7, nonoverlapping GCell-aligned clip. The detailed routing worker can only modify objects with their center lines on or within the standard box. The DRC box is slightly larger than the standard box, enclosing the bounding box of all modifiable objects. We only count and writeback those markers intersecting with the DRC box. The extended box is slightly larger than the DRC box, allowing DRC across the DRC box. In the detailed routing worker database, all objects within the extended box are constructed locally. Only the objects that are on or within the standard box are modifiable, while other objects are fixed. The fixed objects are used for cost calculation and design rule checking.

V. Detailed Routing Worker

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

A. Grid Graph

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

1) Construction: We now describe how to generate the preferred-direction grid lines on each metal layer. We first form all grid lines that are on-track—i.e., align with the DEF TRACKS definitions. Then, we form all grid lines that are off-track—i.e., the center lines along the preferred direction for any existing pathSegs, vias, and pin access points. We also form the grid lines on the boundary. We do not generate the grid lines in the nonpreferred direction. However, bidirectional routing is still available as described in Section V-A2.

Fig. 6 shows how we form the grid lines. Fig. 6(a) shows horizontal Metal1, with seven regular-spaced tracks from DEF. The Metal1 pin has an access point with an off-track y-coordinate. Thus, we create an off-track grid line according to the pin access point location. Fig. 6(b) shows vertical Metal2, with five regular-spaced tracks from DEF. We additionally create an off-track grid on the left boundary. By always creating grid lines along the boundaries of the routing region, we make sure that at least one path exists in the grid graph in any direction, in the case that no on- and off-track grid lines exist (e.g., given a small routing region). Since the center line of the Metal1 pin access point aligns with a Metal2 track, we do not build additional off-track grid lines on Metal2. Similarly, we build grid lines on Metal3. Note that Metal1 and Metal3 grid lines do not necessarily align.

In Fig. 6(d), we show the overlay of x- and y-direction grid lines. The grid vertices are formed by intersecting all x- and y-direction grid lines, and repeating $|Z|$ times along the z direction. Each vertex has six neighbors (except the boundary vertices)—west, east, south, north, down, and up; this is the 3-D grid (projected into the x–y plane) that we use in TritonRoute.

2) Edge: The edge properties are summarized in Table IV. As shown in Fig. 6, not every grid line exists in every metal layer. We use *isEnable* to show whether the edge exists in the path search. A planar edge in the preferred direction is enabled if it is on a current layer grid line. A planar edge in the nonpreferred direction is enabled if it is on an upper-layer grid line (if any, otherwise lower-layer). Via edges are enabled

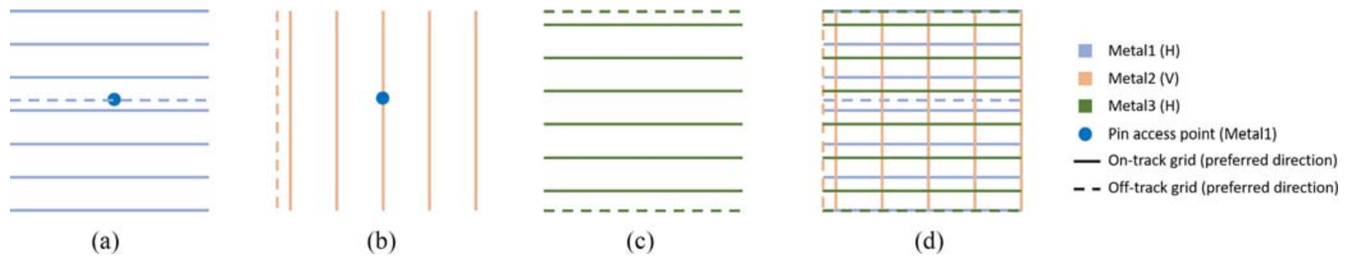
(a)
(b)
(c)
(d)
Metal1 (H)
Metal2 (V)
Metal3 (H)
Pin access point (Metal1)
On-track grid (preferred direction)
Off-track grid (preferred direction)

Fig. 6. Grid graph: (a) preferred-direction grid lines on Metal1; (b) preferred-direction grid lines on Metal2; (c) preferred-direction grid lines on Metal3; and (d) overlay of grid lines (3-D grid graph projected onto the *x*–*y* plane).

TABLE IV
EDGE PROPERTIES

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

TABLE V
VERTEX PROPERTIES

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

between any two preferred-direction grid lines on neighboring metal layers. For each edge, we use *isOnTrack* to show whether the edge is on track; we use *isOnPrefDir* to show whether the edge is on the preferred direction. For a via edge, *specialVia* indicates whether the router should choose a special via instead of the default via. Only pin access points may have this special via property. We preprocess and mark relevant via edges for all up-via pin accesses (using nondefault via). There are two types of costs associated with each edge, object cost, and marker cost. We describe these costs in Section V-B.

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

B. Routing Cost

We use two types of costs: 1) *object cost* and 2) *marker cost*. Overall, the object cost is applied around an existing shape. This cost preemptively guides the path search to go around existing objects to avoid potential DRCs. The marker cost is applied around an existing DRC marker. In the ripup-and-reroute scheme, this cost helps the nets to be routed avoiding the DRC hotspots given the history of DRC data.

1) Object Cost: The object cost is the cost originated from an object, and stored in neighboring edges to the object. We modify this cost whenever the worker database adds or removes an object, e.g., at the time of database initialization, after net ripup, or after routing of one net. We use the object

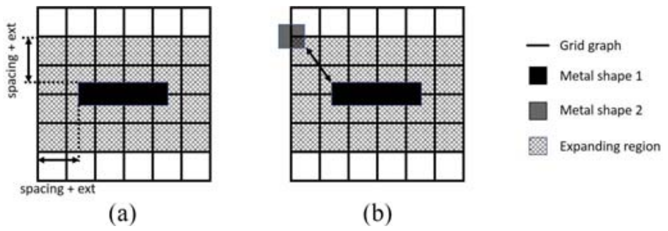

Fig. 7. Object cost from parallel run length spacing: (a) expanding region and (b) shadow object.

cost to prevent potential DRC violations. The evaluation of the object cost is nonprecise but quick, and does not invoke the DRC engine.[5] We support three types of spacing rules for object cost: 1) SPACINGTABLE PARALLELRUNLENGTH; 2) SPACING ENDOFLINE; and 3) SPACING (cut).

For parallel run length spacing, given a *target object*, we first draw an expanding region in which objects on the intersecting edges may cause DRCs, as shown in Fig. 7(a). The expanding region extends beyond the target object up to the maximum required spacing plus half the default width for planar edges, and half the via enclosure for via edges. We then assume a *shadow object* (either a default-width pathSeg or a via) on each of the neighboring planar and via edges, and check against the target object, as shown in Fig. 7(b). For a pathSeg on a planar edge, since the exact length of the shadow object can be arbitrarily longer than the edge length, we add pessimism by assuming maximum parallel run length between the two objects to accelerate convergence. The maximum parallel run length is the length of the target object regardless of the actual parallel run length. For each via edge, we assume a default via or the special via stored with the edge, and check the via enclosure against the target object. The parallel run length between a shadow via enclosure and the target object is calculated by their actual parallel run length. We modify the cost of the edge if there is a violation. Here, the modification of the costs also helps to avoid short violations since the expansion region implicitly includes those edges that may have potential short violations with the target object.

For EOL spacing, we only check the target object if it is a via, and the spacing is only checked along the preferred

[5]We do not have a metric for "precision" of the object cost evaluation. The goals of the quick object cost evaluation, in decreasing priority order, are: 1) quickness and 2) help avoidance of repeated cycles of violations (e.g., arising due to the DRC marker cost in A* search). In practice, we see that our use of quick object cost evaluation—which naturally must be pessimistic—helps avoid cycling.

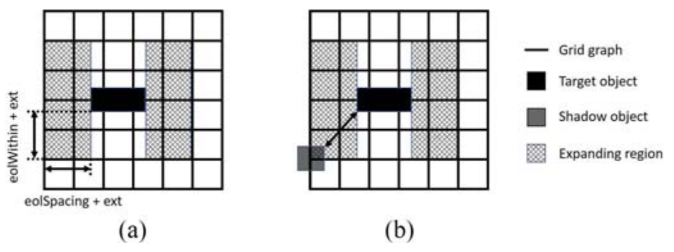

Fig. 8. Object cost from EOL spacing: (a) expanding region and (b) shadow object. The preferred routing direction is horizontal.

routing direction of the metal layer. Spacing orthogonal to the preferred routing direction is not checked to avoid pessimism since almost all jogs end with a preferred-direction routing or a default via, making the line end a non-EOL edge. Fig. 8 illustrates the procedure.

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

The object cost has no history. For example, an object cost is added to the neighboring edges of the target object after the object is created, and subtracted from the neighboring edges of the target object after the object is removed. The object cost calculation supports same-net overriding, blockage spacing overriding and other exceptions. For more details pertaining to this and other parts of our discussion, please refer to [37].

2) Marker Cost: The marker cost is the cost applied according to the DRC markers after each call to the DRC engine. For each marker, we get all objects touching the marker, and add costs to the nearest edge(s) that are used to form the objects. The marker cost has a history within the detailed routing worker. For example, a marker cost is added to an edge and decayed over time (*currIter* in Algorithm 1), but is never subtracted due to the removal of a specific marker. Here, the marker cost history only persists within the detailed routing worker. There is no history between detailed routing iterations shown in Fig. 2.

C. Routing Flow

Now, we describe the routing flow inside a detailed routing worker. In Algorithm 1, line 2 first initializes the worker database from the global database. In this step, we construct a local netlist from the connectivity of routing objects. Fig. 9 shows an example, where a single net passes through the standard box twice, with two parts disjoint. In this case, we construct two subnets so that ripup-and-reroute does not change the connectivity of the net.

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

In lines 9–15, we route each net once according to the net ordering. Before routing, line 10 unreserves the pin access for the current net by subtracting the corresponding object cost of the preferred pin access (up via). Line 11 subtracts the object cost for the boundary objects outside the standard box to avoid unnecessary costs when we connect the net to the boundary pin. Line 12 routes the current net. Line 13 adds the object cost for all the newly routed objects. Line 14 adds back the object cost for boundary objects to prevent design rule violations between these objects to the remaining unrouted nets. Lines 16–19 perform design rule checking and terminate the ripup-and-reroute flow once the clip is clean.

Algorithm 1 Routing Flow

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

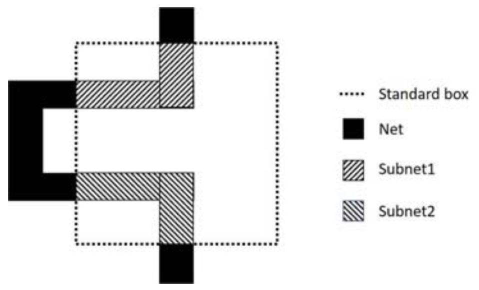

Fig. 9. Local netlist construction: two disjoint subnets constructed in the detailed routing worker from one global net.

[6]Note that the number of iterations is different from the number of "outer" iterations in Fig. 2. For the results that we report in this work, we perform seven (outer) iterations. The *maxIter* number of iterations in Algorithm 1 defines the maximum number of ripup-and-reroute iterations a net inside a DRWorker can undergo. In the current implementation/results represented in this article, we use (1, 4, 4, 4, 4, 4, 4) as the maxIter (for ripup-and-reroute) for each net in the seven "outer" iterations, respectively.

Algorithm 2 Route One Net

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

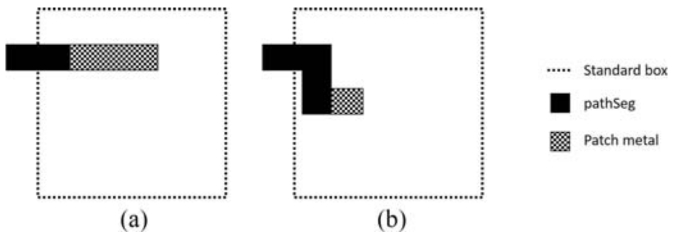

Fig. 10. Minimum area patch metal: (a) patch metal considering area outside of the standard box and (b) patch metal always along the preferred routing direction even if the routing ends in the nonpreferred direction. We assume that the preferred routing direction is horizontal. We do not allow the patch metal to exceed the standard box. If there are more than one patch metal choices, e.g., adding to the left or to the right of a routing object, we choose the one with smaller object cost.

Algorithm 3 Initialization

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

Algorithm 4 Search

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

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

D. Routing One Net

1) Flow: We now describe the methodology to route one net in a detailed routing worker. In our current implementation, in the standard box, a net is either fully routed or unrouted, but not partially routed. Algorithm 2 describes the methodology to route one net. Line 2 gets all unconnected pins, including standard box boundary pins and pins from instTerm and term. Line 3 holds the set of visited grid vertices, and we initialize the set to be empty. Lines 4 and 5 select the source pin to perform path search and remove it from the unconnected pins. To select the source pin, we first calculate the center of gravity for all pins in the x–y plane, then select the pin furthest away from the center of gravity as the source. Line 6 performs the initialization described in Section V-D2. In lines 7–11, we perform the path search as long as there are still unconnected pins. After path search, we update the grid graph in preparation for the next round of path search. The writeDB function backtraces the path to create the routing objects according to the path.

During backtracing, we calculate the total metal area and add necessary patch metals to satisfy the minimum area rule. The patch metals are always created with default routing width along the preferred routing direction. In our implementation, we also build assisting structures to calculate necessary patch metal area for objects connected to the boundary pin. Fig. 10 gives two examples of patch metal addition. The path search is completed once all pins are connected. The path search algorithm is described in Algorithm 4. The update function is described in Algorithm 5.

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

3) Path Search: Algorithm 4 details the path search. The A*-based path search is based on a priority queue. Each element in the priority queue is an element of the search's wavefront, representing that a path exists from the source up to the wavefront grid vertex. In lines 3–5, we first push all visited grids (source) to the queue as the initial wavefront vertices. Then, in lines 6–16, we pop the wavefront vertex with the least cost. We use the previous direction to indicate whether the wavefront vertex has been visited before. Lines 9–11 skip the wavefront vertex if it has been visited before. In lines 12–14, we check whether the wavefront vertex is the destination, and return the path when reaching the destination. Otherwise, we expand the wavefront vertex by pushing its neighbors into the priority queue (with proper cost) as new wavefront vertices.

Here, the cost in the priority queue is the A* cost, consisting of an existing path cost and an estimated future cost, as shown in (1). Whenever we expand from a wavefront vertex to its neighboring vertex, the existing cost is the cost from the wavefront vertex plus the cost to its neighbor, as shown in (2). The cost is the sum of edge length, plus 8× edge length if the edge has a nonzero object cost, and 64× edge length if the edge has a nonzero marker cost. In addition, we

Algorithm 5 Update

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

apply a penalty p if any match to the DRC LUT is found. The estimated future cost is the Manhattan distance to a pre-determined destination, as shown in (3). If there are more than one unconnected pins to be connected, the pre-determined destination is the bounding box of the unconnected pin that is the closest to the bounding box of all visited grids. The Manhattan distance in z direction (between two neighboring metal layers) is calculated as $4\times$ the lower metal layer pitch

$$\text{cost}_{\text{tot}} = \text{cost}_{wf'} + \text{cost}_{est} \tag{1}$$

$$\text{cost}_{wf'} = \text{cost}_{wf} + \text{len}_e + \text{objCost}_e + \text{markerCost}_e + p \tag{2}$$

$$\text{cost}_{est} = \text{dist}_{wf',dst} \tag{3}$$

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

4) Update: Algorithm 5 describes the methodology to update the grid graph. In line 2, we reset the previous direction flag for every grid vertex in preparation of the next path search. In lines 3–6, we set the source flag for every grid vertex along the path. We then add these grid vertices to the visited grids. Here, the source flag and the visited grids serve the same purpose as they both identify the new sources for the next round of path search. However, visited grids are stored in a vector-like container to allow us to initialize the wavefront for the next path search in batches. In lines 7–15, we identify the destination pin that we route to in the current round of path search, remove it from the unconnected pins, and reset the destination flag on all access points of the destination pin.

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

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

In case of disabling feedthrough, special handling methodology is needed for the first source pin of the net, described in lines 17–24. Recall that in line 4 of Algorithm 3, we set the source flag on all access points of the source pin. Given feedthrough disabled, we must reset the source flag on all unused access points of the source pin once the first path search completes.

VI. Experiments

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

A. Setup

We implement our router in C++ with LEF/DEF parser [40] and Boost C++ libraries [39]. We perform experiments using the ISPD-2018 benchmark suite [25]. The ISPD-2018 benchmark suite provides ten testcases in 45 and 32 nm nodes, with up to 290K standard cells and 182K nets. These designs are industrial benchmarks—including large memory cells, off-track pin access, IO ports, and power and macro blockages—with realistic design rules offered in industry-standard input/output formats. The ISPD-2018 benchmark information is summarized in Table VI.

The ISPD-2018 contest evaluation metrics consist of three components: 1) routing, including wirelength and via count; 2) guides and tracks obedience, including out-of-guide wire and vias, off-track wire and vias, and wrong-way wire; and 3) DRCs, including the area of metal shorts, the number of minimum area violations, and the number of spacing violations. However, in the experimental results below, we do not report 2), and make several improvements to 3) according to the following.

1) We do not strictly obey the guides since TritonRoute is not targeting the ISPD-2018 contest. According to the contest organizers, strict guide obedience was never their initial intention although all participating teams and the following published papers all strictly follow the route guides.
2) We do not report the off-track and wrong-way routing although they are already considered throughout the routing flow. In all our reported testcases, such off-track and wrong-way routing account for 0.68% of the total wirelength on average.
3) We report all types of DRCs, including all ISPD-2018 centric DRCs plus (number of) metal short, nonsufficient

TABLE VI
BENCHMARK INFORMATION [25]

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

TABLE VII
COMPARISON OF WIRELENGTH, VIA COUNT, MEMORY USAGE, AND RUNTIME BETWEEN TRITONROUTE (TR) AND DR. CU (CU)

Benchmark	Wirelength (μm)		Via count		Memory (GB)		Runtime (s)	
	TR	CU	TR	CU	TR	CU	TR	CU
ispd18_test1	**86025**	86709	32912	**32402**	**0.08**	0.21	61	**40**
ispd18_test2	1570651	**1566537**	**319855**	325684	**0.43**	1.39	614	**578**
ispd18_test3	1750028	**1743561**	319456	**318309**	**0.47**	1.51	824	**788**
ispd18_test4	**2620890**	2641860	**695901**	729312	**1.09**	5.72	**1866**	3422
ispd18_test5	**2763186**	2780130	**831775**	965544	**1.29**	4.61	**1722**	2383
ispd18_test6	**3557744**	3570351	**1241673**	1480617	**1.71**	5.72	**2682**	3357
ispd18_test7	**6482066**	6517341	**2041794**	2402543	**3.07**	9.87	**5023**	5847
ispd18_test8	**6513278**	6546908	**2062997**	2412121	**3.11**	10.47	**4916**	5932
ispd18_test9	**5442527**	5476029	**2049839**	2410790	**2.71**	10.11	**4378**	4910
ispd18_test10	**6769942**	6809019	**2226243**	2594386	**3.09**	10.58	10129	**9380**

metal overlap, and minimum width. The number of metal short is a good indicator of the strength of the detailed router. Nonsufficient metal overlap and minimum width are two design rules existing in the input, but not considered in the contest evaluation. We believe that the reporting of all types of DRCs effectively forbids any optimization targeting the contest metric.

Among all recently published academic detailed routers [13], [22], [31] that are capable of delivering ISPD-2018 contest solutions, Dr. CU 2.0 [22] dominates the solution quality for all ten testcases in terms of DRCs. Thus, we compare our TritonRoute to Dr. CU 2.0. All experiments are performed using a single thread on an Intel Xeon server.

B. Results

The experimental results are shown in Tables VII and VIII. Table VII gives wirelength, via count, memory consumption, and runtime; and Table VIII gives the details of DRCs.

As a prerequisite, a routing solution is valid only if there are no open nets. All of our reported solutions meet the connectivity requirement. Furthermore, our solution guarantees a loop-free and dangling wire-free solution (except the minimum area patch metals).

We achieve DRC-clean solution for ispd18_test1, and reach an unprecedented, extremely low level of DRCs (<20) in seven of ten testcases while consuming substantially reduced memory, with similar single-threaded runtime. This translates to a 99.3% reduction of DRCs as compared to known best detailed routing solutions from all published academic detailed routers. For the remaining three testcases, we reduce DRCs by 75.1% on average, and by 60.0% at a minimum. Overall, compared to the known best detailed routing solutions, TritonRoute improves wirelength by up to 0.8% (avg. 0.4%), via count by up to 16.1% (avg. 9.3%), and DRCs by up to 100% (avg. 92.0%). TritonRoute completes routing with smaller wirelength and smaller via count, and leaves only a fraction of DRCs compared to all other academic detailed routers.

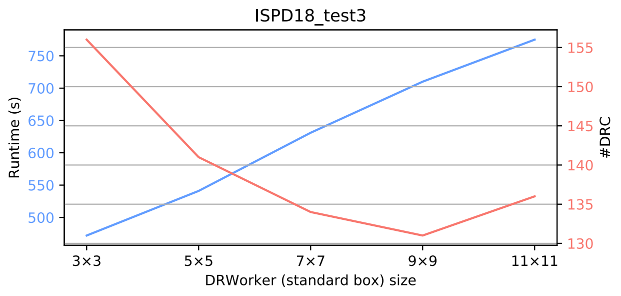

Fig. 11. Illustration of tradeoff between runtime and final DRC count with various DRWorker standard box sizes in unit of GCell.

We have also performed a case-study experiment using different standard box sizes to analyze the tradeoff between runtime and final DRC count. We sweep the standard box size from 3×3 to 11×11 with a step size of 2 on the ISPD18_test3 testcase. The specific testcase that we choose has relatively high *#violation-to-#instance* ratio, which indicates that ISPD18_test3 is a difficult and congested design among the ISPD18 contest benchmarks. Fig. 11 illustrates the tradeoff between runtime and final DRC count with different standard box sizes. We observe that a larger standard box provides a larger solution space for ripup-and-reroute for DRC fixing at the cost of longer runtime for A* search. A standard

TABLE VIII
COMPARISON OF NUMBER OF MINIMUM WIDTH (MINWID), NONSUFFICIENT-METAL OVERLAP (NSMET), MINIMUM AREA (MAR), METAL SHORT (SHORT), CUT SHORT (CSHORT), METAL PARALLEL RUN LENGTH SPACING (METSPC), METAL EOL SPACING (EOLSPC), CUT SPACING (CUTSPC), AND TOTAL DESIGN RULE VIOLATIONS BETWEEN TRITONROUTE (TR) AND DR. CU (CU)

Benchmark	**Design rule violations**																	
	#MinWid		#NSMet		#MAR		#Short		#CShort		#MetSpc		#EOLSpc		#CutSpc		**#Total**	
	TR	CU	TR	CU	TR	CU	TR	CU	TR	CU	TR	CU	TR	CU	TR	CU	TR	CU
ispd18_test1	**0**	0	**0**	1716	**0**	0	**0**	1	**0**	0	**0**	1	**0**	1	**0**	0	**0**	1719
ispd18_test2	**0**	0	**0**	20048	**0**	0	**1**	1	**0**	0	**7**	49	**9**	9	**0**	0	**17**	20107
ispd18_test3	**0**	0	**0**	21224	**0**	0	**112**	219	1	**0**	**17**	86	10	**9**	2	**0**	**142**	21538
ispd18_test4	**0**	10	**2**	17	**0**	32	**190**	287	**0**	0	**132**	289	**2**	164	**0**	142	**326**	941
ispd18_test5	**0**	7	**0**	19	**0**	48	**2**	342	**0**	0	**0**	309	**0**	36	**0**	20	**2**	781
ispd18_test6	**0**	8	**0**	44	**3**	92	**1**	36	**0**	0	**2**	489	**2**	21	**0**	30	**8**	720
ispd18_test7	**0**	0	**0**	11	**5**	127	**4**	604	**0**	0	**4**	129	**0**	7	**0**	60	**13**	938
ispd18_test8	**0**	0	**0**	19	**3**	138	**2**	625	**0**	0	**1**	118	**0**	15	**0**	59	**6**	974
ispd18_test9	**0**	0	**0**	16	**4**	185	**1**	39	**0**	0	**0**	49	**0**	7	**0**	54	**5**	350
ispd18_test10	**0**	0	**0**	26	**4**	228	**1103**	3180	5	**1**	**425**	742	144	**73**	**33**	100	**1714**	4350

box with size of 7×7 GCells can achieve a decent tradeoff between runtime and final DRC count, especially for difficult designs.

VII. CONCLUSION

In this work, we presented TritonRoute, an open-source detailed router. We describe an in-memory router database, and an end-to-end detailed routing scheme. We evaluate our router using the official ISPD-2018 benchmark suite, and show that we reach an unprecedented, extremely low level of DRCs ($<$20) in seven of ten testcases, a 99.3% reduction of DRCs on average compared to known best detailed routing solution from all published academic detail routers. Overall, compared to the known best detailed routing solution, TritonRoute improves wirelength by up to 0.8% (avg. 0.4%), via count by up to 16.1% (avg. 9.3%), and DRCs by up to 100% (avg. 92.0%). Due to its generic nature, our framework can support extensions to new technologies or design rules. Our ongoing work includes: 1) support of multithreading; 2) track assignment improvement; 3) runtime improvement; 4) support of advanced technology nodes (including ISPD-2019 contest benchmarks); and 5) support of via generation and via swapping.

ACKNOWLEDGMENT

The authors would like to thank Dr. Patrick Groeneveld, Dr. Wen-Hao Liu, and Dr. Stefanus Mantik for providing valuable feedback.

REFERENCES

[1] 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.

[2] 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.

[3] H.-Y. Chen and Y.-W. Chang, "Global and detailed routing," in *Electronic Design Automation: Synthesis, Verification, and Test*, L.-T. Wang, Y.-W. Chang and T. K.-T. Cheng, Eds. Amsterdam, The Netherlands: Morgan Kaufmann, 2009, ch. 12, pp. 687–749. [Online]. Available: http://cc.ee.ntu.edu.tw/~ywchang/Courses/PD_Source/EDA_routing.pdf

[4] 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. 24th Asia South Pac. Design Automat. Conf. (ASP-DAC)*, 2019, pp. 754–760.

[5] G. Chen, C.-W. Pui, H. Li, and E. F. Y. Young, "Dr. CU: Detailed routing by sparse grid graph and minimum-area-captured path search," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, early access, Jul. 09, 2019, doi: 10.1109/TCAD.2019.2927542.

[6] Y. Ding, C. Chu, and W.-K. Mak, "Self-aligned double patterning lithography aware detailed routing with color preassignment," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 36, no. 8, pp. 1381–1394, Aug. 2017.

[7] S. Dolgov, A. Volkov, L. Wang, and B. Xu, "2019 CAD contest: LEF/DEF based global routing," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Westminster, CO, USA, 2019, pp. 1–4.

[8] Y. Du *et al.*, "Spacer-is-dielectric-compliant detailed routing for self-aligned double patterning lithography," in *Proc. 50th ACM/EDAC/IEEE Design Automat. Conf. (DAC)*, Austin, TX, USA, 2013, pp. 1–6.

[9] A. Feller, "Automatic layout of low-cost quick-turnaround random-logic custom LSI devices," in *Proc. 13th Design Automat. Conf. (DAC)*, 1976, pp. 79–85.

[10] G.-R. Gao and D. Z. Pan, "Flexible self-aligned double patterning aware detailed routing with prescribed layout planning," in *Proc. ACM Int. Symp. Phys. Design (ISPD)*, 2012, pp. 25–32.

[11] 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 Automat. Electron. Syst.*, vol. 18, no. 2, pp. 1–24, 2013.

[12] S. M. M. Gonçalves, L. S. da Rosa, and F. de S. Marques, "An improved heuristic function for A*-based path search in detailed routing," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Sapporo, Japan, 2019, pp. 1–5.

[13] S. M. M. Gonçalves, L. S. da Rosa, and F. de S. Marques, "DRAPS: A design rule aware path search algorithm for detailed routing," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, early access, Aug. 27, 2019, doi: 10.1109/TCSII.2019.2937893.

[14] F. O. Hadlock, "A shortest path algorithm for grid graphs," *Networks*, vol. 7, no. 4, pp. 323–334, 1977.

[15] K. Han, A. B. Kahng, and H. Lee, "Evaluation of BEOL design rule impacts using an optimal ILP-based detailed router," in *Proc. 52nd ACM/EDAC/IEEE Design Automat. Conf. (DAC)*, San Francisco, CA, USA, 2015, pp. 1–6.

[16] A. Hetzel, "A sequential detailed router for huge grid graphs," in *Proc. Design Automat. Test Eur. (DATE)*, Paris, France, 1998, pp. 332–339.

[17] D. W. Hightower, "A solution to line-routing problems on the continuous plane," in *Proc. 6th Annu. Design Automat. Conf. (DAC)*, 1969, pp. 1–24.

[18] 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.

[19] A. B. Kahng, L. Wang, and B. Xu, "The tao of PAO: Anatomy of a pin access oracle for detailed routing," in *Proc. ACM/EDAC/IEEE Design Automat. Conf. (DAC)*, Jul. 2020.

[20] C. Y. Lee, "An algorithm for path connections and its applications," *IRE Trans. Electron. Comp.*, vol. EC-10, no. 3, pp. 346–365, Sep. 1961.

[21] H. K.-S. Leung, "Advanced routing in changing technology landscape," in *Proc. Int. Symp. Phys. Design (ISPD)*, 2003, pp. 118–121.

[22] 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," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Westminster, CO, USA, 2019, pp. 1–7.

[23] I.-J. Liu, S.-Y. Fang, and Y.-W. Chang, "Overlay-aware detailed routing for self-aligned double patterning lithography using the cut process," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 9, pp. 1519–1531, Sep. 2016.

[24] W. K. Luk, "A greedy switch-box router," *Integra. VLSI J.* vol. 3, no. 2, pp. 129–149, 1985.

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

[26] T. Nieberg, "Gridless pin access in detailed routing," in *Proc. 48th ACM/EDAC/IEEE Design Automat. Conf. (DAC)*, New York, NY, USA, 2011, pp. 170–175.

[27] N. J. Nilsson, "State-space search methods," in *Problem-Solving Methods in Artificial Intelligence*. New York, NY, USA: McGraw-Hill, 1971, pp. 43–79.

[28] I. Pohl, "Bi-directional search," *Machine Intelligence*. Edinburgh, U.K.: Edinburgh Univ. Press, 1971, pp. 127–140.

[29] E. Shragowitz and S. Keel, "A global router based on a multicommodity flow model," *Integr. VLSI J.*, vol. 5, no. 1, pp. 3–16, 1987.

[30] J. Soukup, "Fast maze router," in *Proc. 15th Design Automat. Conf. (DAC)*, 1978, pp. 100–102.

[31] 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)*, San Diego, CA, USA, 2018, pp. 1–7.

[32] P.-S. Tzeng and C. H. Sequin, "Codar: A congestion-directed general area router," in *IEEE Int. Conf. Comput.-Aided Design Dig. Tech. Papers (ICCAD-89)*, Santa Clara, CA, USA, 1988, pp. 30–33.

[33] M.-P. Wong, W.-H. Liu, and T.-C. Wang, "Negotiation-based track assignment considering local nets," in *Proc. 21st Asia South Pac. Design Automat. Conf. (ASP-DAC)*, Macau, China, 2016, pp. 378–383.

[34] X. Xu, B. Yu, J.-R. Gao, C.-L. Hsu, and D. Z. Pan, "PARR: Pin access planning and regular routing for self-aligned double patterning," *ACM Trans. Design Automat. Electron. Syst.*, vol. 21, no. 3, p. 42, 2016.

[35] 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.

[36] (Nov. 2009). *LEF/DEF Language Reference*. [Online]. Available: http://www.ispd.cc/contests/18/lefdefref.pdf

[37] *The-OpenROAD-Project/TritonRoute: UCSD Detailed Router*. Accessed: Dec. 13, 2019. [Online]. Available: https://github.com/The-OpenROAD-Project/TritonRoute

[38] W.-H. Liu. *ISPD 2018 Initial Detailed Routing Contest and Benchmarks*. Accessed: Dec. 4, 2019. [Online]. Available: http://www.ispd.cc/slides/2018/s7_3.pdf

[39] B. Schäling, *The Boost C++ Libraries, 2nd ed.*. Laguna Hills, CA, USA: XML Press, 2014.

[40] *LEF/DEF Reference 5.7*. Accessed: Dec. 4, 2019. [Online]. Available: http://www.si2.org/openeda.si2.org/projects/lefdefnew

[41] *Si2 OpenAccess*. Accessed: Dec. 4, 2019. [Online]. Available: http://projects.si2.org/?page=69

Andrew B. Kahng (Fellow, IEEE) received the Ph.D. degree in computer science from the University of California at San Diego, La Jolla, CA, USA, in 1989.

He is a Professor with the Department of Computer Science Engineering and the Department of Electrical and Computer Engineering, University of California at San Diego. His interests include IC physical design, the design–manufacturing interface, combinatorial optimization, and technology roadmapping.

Lutong Wang (Graduate Student Member, IEEE) received the B.S. degree in microelectronics from Tsinghua University, Beijing, China, in 2014, and the M.S. degree in electrical and computer engineering from the University of California at San Diego, La Jolla, CA, USA, in 2016, where he is currently pursuing the Ph.D. degree.

His research interests include physical design implementation and DFM methodologies.

Bangqi Xu (Graduate Student Member, IEEE) received the B.S. degree in electrical engineering from the University of Michigan at Ann Arbor, Ann Arbor, MI, USA, in 2015, and the M.S. degree in electrical and computer engineering from the University of California at San Diego, La Jolla, CA, USA, in 2017, where he is currently pursuing the Ph.D. degree.

His current research interests include detailed placement, routing methodology, and optimization.