The Tao of PAO: Anatomy of a Pin Access Oracle for Detailed Routing

Andrew B. Kahng[†‡], Lutong Wang[‡] and Bangqi Xu[‡]
[†]CSE and [‡]ECE Departments, UC San Diego, La Jolla, CA, USA
{abk, luw002, bangqixu}@ucsd.edu

***Abstract*—Pin accessibility has been widely studied, particularly in recent works that span detailed placement optimization, standard cell layout optimization and new design rule-aware access model. However, to our knowledge, no previous work has described a full solution for pin access analysis, with validations on real detailed routing benchmarks. This paper presents a complete, robust, scalable and design rule-aware dynamic programming-based pin access analysis framework that is capable of both standard cell-based and instance-based pin access analysis. Integration into the open-source TritonRoute router results in superior solution quality compared to previous best-known results for the official ISPD-2018 benchmark suite.**

I. Introduction

Pin accessibility has been one of the most crucial issues in advanced node enablement [1][9]. Various related topics have been widely studied in recent works; these include detailed placement optimization, standard cell layout optimization, and a new design rule-aware access model.

The works of [6][15] perform detailed placement optimization using a global routing solution as guidance, with pin accessibility modeled only in the form of pin density. Ding [2] develops a dynamic programming and linear programming-based detailed placement optimization considering pin access per instance pin. Ye [14] proposes an integer linear programming formulation to solve the unidirectional cell layout optimization under middle-of-line structure. However, the above models are over-simplified with assumptions of 1D gridded design and distance-based cost function, with no precise awareness of design rules. Ozdal [8] proposes a multicommodity flow-based method with Lagrangian relaxation to solve the escape routing problem for dense pin clusters. However, the proposed method has limited scalability due to its per-net analysis nature. Recently, works of Xu et al. [11][12][13] have developed a series of pin access planning and regular routing techniques for self-aligned double patterning. These works, still under the assumption of 1D gridded design, are the first in the open literature that try to address both cell-level and instance-level pin accessibility. However, the methodology has drawbacks: (i) there is no robust flow to generate "hit points" given any 1D/2D, gridded/non-gridded design, with or without specific (e.g., self-aligned double patterning) design rules; (ii) the flow is unrealistic in that the number of "hit point combinations" is very large, resulting in a complex lookup table that is impractical to use; and (iii) the benchmark suite is not public and includes testcases only up to 12K cells. These small testcases nevertheless consume as much as 800 seconds of wall time in multithreaded mode, which is a prohibitive runtime cost for real industry testcases and use contexts.

To our knowledge, no works present a complete, fully-defined pin access analysis flow, or demonstrate robustness with a real detailed routing contest benchmark suite. In this work, we present a real, robust, scalable and design rule-aware dynamic programming-based pin access analysis framework that performs both standard cell-based and instance-based pin access analysis. Through integration to the open-source TritonRoute [4][20], we demonstrate superior solution quality over the best known results [5] using the official ISPD-2018 benchmark suite [7]. Our main contributions are summarized as follows.

- We propose a multi-level, standard cell-based and instance-based pin access analysis framework with intra-cell and inter-cell pin accessibility awareness.
- We propose a robust and design rule-aware pin access point generation methodology for unique instances, supporting both planar and via access, and both on-track and off-track access.
- To achieve **intra-cell** pin compatibility, we propose a dynamic programming-based, design rule and boundary conflict-aware access pattern generation methodology for unique instances.
- We propose a dynamic programming-based access pattern selection methodology for standard cell instance clusters, which minimizes **inter-cell** pin access conflicts. To the best of our knowledge, this proposed framework is the only scalable solution in the open literature.
- We improve the pin access over the open-source TritonRoute v0.0.6.0 [19] (the latest release as this paper is finalized), achieving design rule check (DRC)-clean via access for all of ISPD-2018 benchmark suite testcases. With the integration to TritonRoute, we demonstrate superior solution quality over the best known results using the official ISPD-2018 benchmark suite.

The remainder of this paper is organized as follows. Section II provides background information for pin access. Section III describes our pin access methodology. Section IV presents our experimental setup and results. Section V gives conclusions and directions for future work.

II. Preliminaries

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

A. Unique Instance

A **unique instance** is defined by a *signature*, which consists of (i) the cell master of the instance (e.g., NANDX1, NORX4, etc.); (ii) the orientation of the instance (e.g., R0, R180, MX, MY); and (iii) offsets to all track patterns that exist in the design DEF. Two instances having different signatures require separate intra-cell pin access analysis flows. Figures 1(a) and (b) illustrate two different unique instances. Although the two instances share the same cell master and orientation, they are considered as different unique instances because they have different offsets to routing track patterns, resulting in different on-track, off-track conditions for the same pin access location (relative to the origin of the cell master). Thus, these instances require separate intra-cell pin access analyses. By contrast, two instances having the same signature would have exactly the same intra-cell pin access

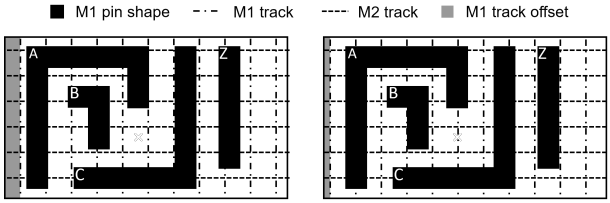

Fig. 1. Illustration of two different unique instances that have the same cell master and orientation, but different offsets to track patterns.

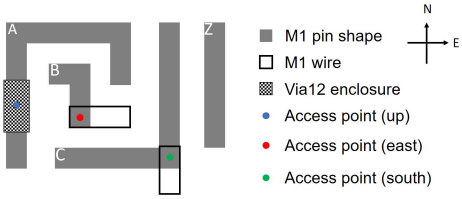

Fig. 2. Illustration of access points.

analysis result. Thus, we only need to perform intra-cell pin access analysis once for each unique instance.

B. Access Point and Access Pattern

1) Access Point: For each pin, an access point is an x-y coordinate on a metal layer where the detailed router ends routing. Each access point stores one or more directions from which the router can access the pin.[1] For example, in Figure 2, pin A has an access point indicating the up direction. We use a via12 enclosure to show that an up-via (i.e., a via connecting the pin to the upper metal layer) is valid to escape from this access point. Similarly, pin B (resp. C) has an access point indicating that routing to the east (resp. south) is valid. In our implementation, each access point may store, i.e., 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.

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

C. Coordinate Type

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

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

[1]We only consider access points that have valid up (via) direction access for standard cells, since via access is preferred as compared to planar access in advanced technology nodes.

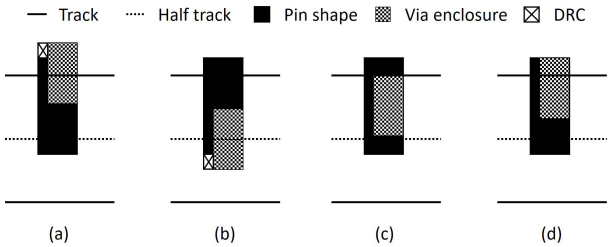

Fig. 3. Illustration of four y-coordinate types, overlaid with same-layer up-via enclosure at the access point: (a) on-track; (b) half-track; (c) shape-center; and (d) enclosure boundary. Only (c) and (d) are DRC-clean.

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

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

III. Methodology

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

A. Step 1: Pin-Based Access Point Generation

Although we could enumerate all coordinate types to generate every access point per pin, in a reasonable detailed routing-driven pin access analysis framework the number of generated access points per pin should be neither too small nor too large. Too small a number of access points will overly restrict the solution space in detailed routing, resulting in degraded solution quality. On the other hand, given the heuristic, cost-based nature of modern detailed routing [5][20], too large a number of access points will provide excessive options (e.g., many off-track access points) for the detailed router, again resulting in degraded solution quality. Thus, the access point generation flow must be robustly designed to generate a proper amount of access points. In our flow, for example, to generate an access point at (x, y) on Metal1, where the preferred routing direction is horizontal, we consider all four coordinate types for the y coordinate (corresponding to the preferred direction), but only consider the first three coordinate types for the x coordinate (corresponding to the non-preferred direction), so as to reduce unique, off-track coordinates. We explain below the determination of "proper amount" after the description of Algorithm 1.

Algorithm 1 Pin-based access point generation

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

Algorithm 1 describes the pin-based access point generation. In Lines 3 – 4, we loop through different combinations of x and y coordinates sequentially according to their cost. For example, we first generate all (on-track, on-track) points, then (off-track, on-track) points, etc. In Line 5, for each type of coordinates, we first generate all access points. Then in Lines 6 – 10, we add all valid access points to the output. An access point is valid if a via can be dropped DRC-free to access the pin. We use an accurate DRC engine similar to the one used in [20] to perform the design rule check, considering all design rules existing in the specific design. Next, in Lines 11 – 13, we check whether we have generated enough access points for a pin, and early-terminate the procedure once the number of generated access points is equal to or greater than our required number k. Given the above, all access points of given coordinate types are generated, DRC-checked and added before we try to early-terminate the procedure. Therefore, the number of access points generated may be slightly larger than k. This behavior allows more access points to be generated when we are given a large pin shape, while also reducing the occurrence of unique, off-track coordinates. In our implementation, $k = 3$ for both standard-cell and macro-cell pins.

B. Step 2: Unique Instance-Based Access Pattern Generation

For each unique instance, we now describe how to pick a good access point per pin to form an *access pattern* in which the chosen access points are compatible with each other. Figure 4 illustrates our unique instance-based access pattern generation flow. The access pattern generation mainly consists of (i) pin ordering, (ii) graph construction, and (iii) dynamic programming-based pattern generation.

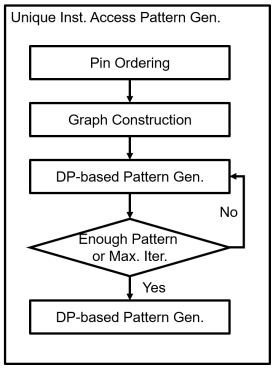

Fig. 4. Iterative access pattern generation flow.

Pin ordering. Pin ordering is a preparation step for graph construction and dynamic programming-based pattern generation. Given a unique instance and an ordering of the pins in the unique instance, we assume only the neighboring ordered two-pin pairs might have conflicting access points (i.e., the two access points cause DRCs). For example, if we have a pin order of <A, B, C, Z>, then our assumption is that only <A, B>, <B, C> and <C, Z> could have conflicting access points, while <A, C>, <A, Z> and <B, Z> should not have conflicting access points. In this way, the access patterns can be generated within a reasonable amount of time, without the need to perform design rule check among all two-pin pairs. For corner cases where non-neighboring two-pin pairs have conflicting access points, we can still avoid such cases by a post-processing method, described at the end of the discussion below of DP-based access pattern generation. As shown in Section IV, this method works well in all ISPD-2018 benchmark suite testcases.

For a pin, if the averaged coordinates of all its access points are (x_{avg}, y_{avg}), then given a unique instance, we sort the pins according to $(x_{avg} + \alpha \cdot y_{avg})$. Figure 5 illustrates an example of a unique instance with four pins. If $\alpha = 0$, then the pin ordering is equivalent to the ordering of x_{avg}. Thus, we obtain a pin order of <A, B, C, Z>. The first and last pins according to the pin order are **boundary pins**, which receive special treatment in access pattern generation as described below. Generally, given a reasonably small α ($\alpha < 1$), the first and last pins are the leftmost and the rightmost pins in the unique instance, respectively. In our implementation, we use $\alpha = 0.3$.

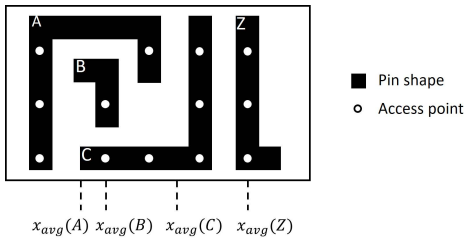

Fig. 5. Pin ordering.

Graph construction. We build a graph for dynamic programming. Figure 6 shows the directed graph corresponding to the unique instance shown in Figure 5, assuming $\alpha = 0$. All edges are directed from left to right in the figure. The leftmost (resp. rightmost) vertex in the graph is the (virtual) starting (resp. ending) vertex, which serves as the starting (resp. ending) point in the dynamic programming that we describe below. Vertices between the starting and ending vertices represent access points; these are grouped by the owner pin of the access point, and ordered sequentially following the aforementioned pin order. We label all access points according to the pin index (m) and access point index (n). For example, the access point with label {3,2} in Figure 6 is the second access point ($n = 2$) of the third pin ($m = 3$). We build complete bipartite graphs over neighboring groups' respective vertex sets. A path from the starting vertex to the ending vertex visits one access point vertex per pin. The visited access points represent an access pattern.

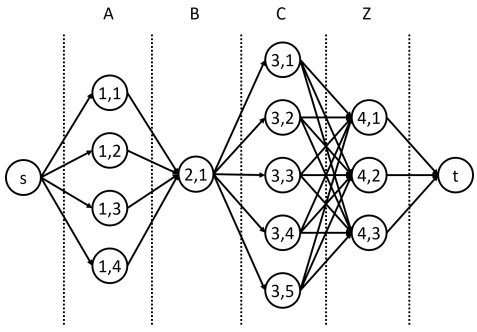

Fig. 6. Graph for dynamic programming-based access pattern generation.

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

Algorithm 2 Access pattern generation

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

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

Finally, for all the access patterns that we generate, we use a DRC engine similar to the one used in [20] to validate whether there exist unseen DRCs, i.e., between non-neighboring groups of access points, or between multiple objects. To accelerate the access pattern generation, only up-vias are included for DRC.

Algorithm 3 Edge cost calculation

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

C. Step 3: Cluster-Based Access Pattern Selection

Given access patterns per unique instance, we select the best access patterns per instance so that the access patterns of neighboring instances are compatible. Our cluster-based access pattern selection is performed on a continuous chunk of instances. We first group all instances according to their rows, and each continuous chunk of instances (no empty site in between) forms a cluster. We only consider the access pattern compatibility within a cluster while assuming that the neighboring clusters within or across rows always allow compatible access patterns. The cluster-based access pattern selection works similarly to the access pattern generation. The pin ordering step, in Algorithm 2, is now replaced with the instance ordering step, which naturally follows the left-to-right instance ordering. The graph construction works the same way except that now each vertex represents an access pattern of an instance. Figure 7 shows the ordered cell instances in a cluster and their corresponding graph. Finally, the dynamic programming-based optimization selects the best access pattern per instance to minimize the total cost. To accelerate the procedure, only up-vias of boundary access points (e.g., for pin A and pin Z of each instance in Figure 7(a)) are included for DRC.

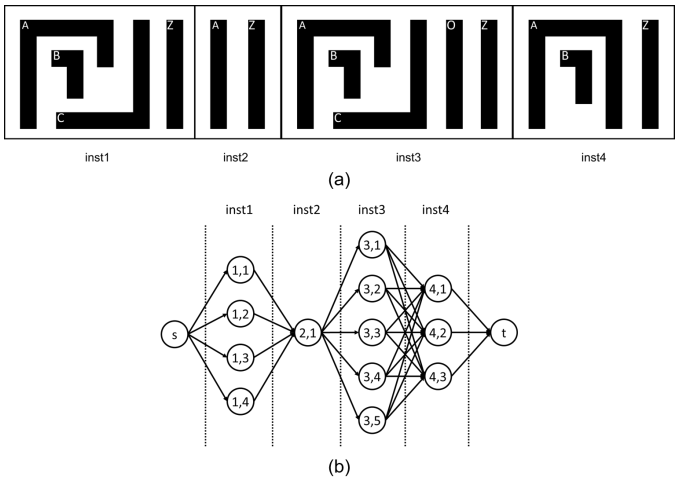

Fig. 7. Illustration of (a) ordered cell instances and (b) corresponding graph.

IV. Experiments

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

A. Experimental Setup

We implement our pin access analysis in C++ and integrate our framework with the open-source TritonRoute [20]. We perform all our experiments using the official ISPD-2018 initial detailed routing contest benchmark suite [7]. Table I summarizes the testcase information.

TABLE I
TESTCASE INFORMATION [7].

Benchmark	#Standard cell	#Macro cell	#Net	#IO pin	#Layer	Die size	Tech. node
ispd18_test1	8879	0	3153	0	9	$0.20\times0.19mm^2$	$45nm$
ispd18_test2	35913	0	36834	1211	9	$0.65\times0.57mm^2$	$45nm$
ispd18_test3	35973	4	36700	1211	9	$0.99\times0.70mm^2$	$45nm$
ispd18_test4	72094	0	72401	1211	9	$0.89\times0.61mm^2$	$32nm$
ispd18_test5	71954	0	72394	1211	9	$0.93\times0.92mm^2$	$32nm$
ispd18_test6	107919	0	107701	1211	9	$0.86\times0.53mm^2$	$32nm$
ispd18_test7	179865	16	179863	1211	9	$1.36\times1.33mm^2$	$32nm$
ispd18_test8	191987	16	179863	1211	9	$1.36\times1.33mm^2$	$32nm$
ispd18_test9	192911	0	178857	1211	9	$0.91\times0.78mm^2$	$32nm$
ispd18_test10	290386	0	182000	1211	9	$0.91\times0.87mm^2$	$32nm$

TABLE II
RESULTS FOR EXPERIMENT 1: COMPARISON BETWEEN THE ORIGINAL TRITONROUTE (TRRTE) AND OUR PIN ACCESS ANALYSIS FRAMEWORK (PAAF) FOR ALL UNIQUE INSTANCE PINS (WITHOUT CONSIDERING INTRA-CELL OR INTER-CELL PIN ACCESS COMPATIBILITY) IN TERMS OF TOTAL #ACCESS POINTS GENERATED (TOTAL #APS), #ACCESS POINTS WITH DRCS (#DIRTY APS), AND RUNTIME.

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

These testcases are real industry designs with up to 290K standard cells in two technology nodes. We note that these testcases use real industry LEF-based design rule syntax, which is much more realistic than the testcases used in previous works [11][13]. Currently, no pin access framework targets the ISPD-2018 benchmark suite. To our best knowledge, no pin access framework has ever demonstrated enough robustness and scalability in publicly accessible, large benchmark testcases. Thus, we compare our work with the pin access framework from the latest release of the open-source TritonRoute v0.0.6.0 [19]. Furthermore, to enable a broader horizontal comparison to other frameworks, we also make necessary improvements to TritonRoute in addition to the integration of pin access analysis. We compare final routed designs to the best known academic detailed router – Dr. CU 2.0 [5]. All our experiments are performed using a Xeon $2.6GHz$ server in single-threaded mode. We perform three experiments.

- **Experiment 1**: We compare the quality of access points for all unique instance pins (without consideration of intra-cell or inter-cell pin access compatibility) from this work with that from TritonRoute v0.0.6.0.
- **Experiment 2**: We compare the quality of access points for all instance pins (with consideration of intra-cell and inter-cell pin compatibility) from this work with that from TritonRoute v0.0.6.0.
- **Experiment 3**: By integrating our framework with the open-source TritonRoute and making additional improvements, we enable a preliminary comparison of pin accesses from the final routed design, and also of the final routed #DRCs, between the original TritonRoute, the best known published result from Dr. CU 2.0 [5][17], and our pin access analysis framework. We further demonstrate the capability to extend our PAAF into $14nm$ and below nodes.

B. Experimental Results

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

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

[2]For the testcases we use in our experiments, three access patterns per unique instance are enough to enable selection of mutually DRC-clean access patterns for all instances.

TABLE III

RESULTS FOR EXPERIMENT 2: COMPARISON BETWEEN THE ORIGINAL TRITONROUTE (TRRTE) AND OUR PIN ACCESS ANALYSIS FRAMEWORK (PAAF) FOR ALL INSTANCE PINS (CONSIDERING INTRA-CELL AND INTER-CELL PIN ACCESS COMPATIBILITY) IN TERMS OF #PINS WITHOUT A DRC-CLEAN ACCESS POINT (#FAILED PINS), AND RUNTIME. TOTAL #PINS MEANS THE TOTAL NUMBER OF ALL INSTANCE PINS (WITH NET ATTACHED).

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

Experiment 3. By integrating our framework with the open-source TritonRoute v0.0.6.0 [19] (the latest release as this paper is finalized) and making additional improvements, we show a preliminary result of pin accesses from the final routed design, and also of the #DRCs for the final routed design, for testcase *ispd18_test5*. Figure 8 compares two pin accesses from the final routed design, between Dr. CU 2.0 and our PAAF. As noted above, PAAF is capable of generating DRC-clean pin access for all instance pins. By using our robust PAAF, we surpass the best known academic detailed routing result in terms of #DRCs. The current best known result comes from Dr. CU 2.0 [5][17], with 755 DRCs. By contrast, we complete detailed routing with only two DRCs, and with no pin access issues remaining.

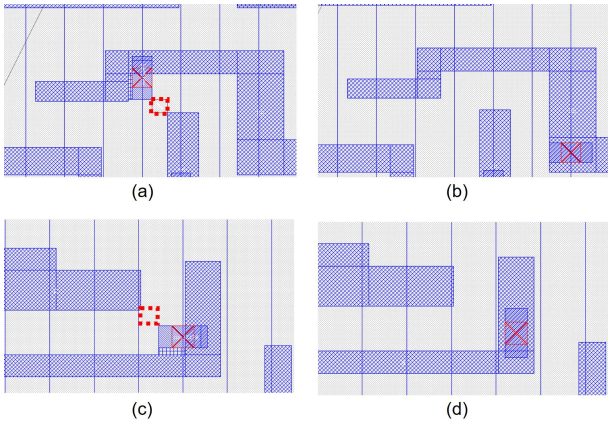

Fig. 8. Comparison of pin access between Dr. CU 2.0 and PAAF: (a) Dr. CU 2.0 (Case 1), (b) PAAF (Case 1), (c) Dr. CU 2.0 (Case 2), and (d) PAAF (Case 2). Dashed red boxes are DRCs. Testcase: *ispd18_test5*.

We also perform a preliminary study on pin accessibility using a commercial $14nm$ library. We perform our experiments using the AES testcase from OpenCores [18] (20K instances, 779 unique instances). Our preliminary study shows that our PAAF successfully generates and selects DRC-clean access points for all 57K instance pins in a runtime of 9 seconds. An example of standard cell pin accesses is shown in Figure 9.

V. CONCLUSIONS

In this work, we present a multi-level, standard cell- and instance-based, complete, robust, scalable and design rule-aware pin access analysis framework. We describe our robust pin-based access point generation, boundary conflict-aware access pattern generation and cluster-based access pattern selection based on dynamic programming. We achieve 100% DRC-clean pin access and demonstrate superior final detailed routing solution compared to the best known results using the ISPD-2018 initial detailed routing benchmark suite. Our ongoing work includes: (i) support of multi-height cells in advanced FinFET technology nodes; and (ii) support of multi-threading to further reduce runtime.

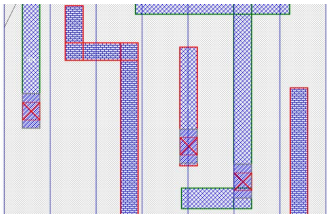

Fig. 9. Illustration of pin accesses in $14nm$. Note that off-track pin access is enabled automatically in PAAF.

REFERENCES

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

[2] Y. Ding, C. Chu and W.-K. Mak, "Pin Accessibility-Driven Detailed Placement Refinement", *Proc. ISPD*, 2017, pp. 133-140.

[3] S. Dolgov, A. Volkov, L. Wang and B. Xu, "2019 CAD Contest: LEF/DEF Based Global Routing" *Proc. ICCAD*, 2019, to appear.

[4] A. B. Kahng, L. Wang and B. Xu, "TritonRoute: An Initial Detailed Router for Advanced VLSI Technologies", *Proc. ICCAD*, 2018, pp. 81:1-81:8.

[5] 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, to appear.

[6] W.-H. Liu, C.-K. Koh and Y.-L. Li, "Optimization of Placement Solutions for Routability", *Proc. DAC*, 2013, pp.153:1-153:9.

[7] S. Mantik, G. Posser, W.-K. Chow, Y. Ding and W.-H. Liu, "ISPD2018 Initial Detailed Routing Contest and Benchmarks", *Proc. ISPD*, 2018, pp. 140-143.

[8] M. M. Ozdal, "Detailed-Routing Algorithms for Dense Pin Clusters in Integrated Circuits", *IEEE Trans. on CAD* 28(3) (2009), pp. 340-349.

[9] X. Qiu and M. Marek-Sadowska, "Can Pin Access Limit the Footprint Scaling?", *Proc. DAC*, 2012, pp. 1100-1106.

[10] J. Seo, J. Jung, S. Kim and Y. Shin, "Pin Accessibility-Driven Cell Layout Redesign and Placement Optimization", *Proc. DAC*, 2017, pp. 54:1-54:6.

[11] X. Xu, B. Cline, G. Yeric, B. Yu and D. Z. Pan, "Self-Aligned Double Pattering Aware Pin Access and Standard Cell Layout Co-Optimization", *IEEE Trans. on CAD* 34(5) (2015), pp. 699-712.

[12] X. Xu, Y. Lin, V. Livramento and D. Z. Pan, "Concurrent Pin Access Optimization for Unidirectional Routing", *Proc. DAC*, 2017, pp. 20:1-20:6.

[13] 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. on DAES* 21(3) (2016), article 42.

[14] W. Ye, B. Yu, D. Z. Pan, Y.-C. Ban and L. Liebmann, "Standard Cell Layout Regularity and Pin Access Optimization Considering Middle-of-Line", *Proc. GLSVLSI*, 2015, pp. 289-294.

[15] Y. Zhang and C. Chu, "CROP: Fast and Effective Congestion Refinement of Placement", *Proc. ICCAD*, 2009, pp. 344-350.

[16] Cadence Innovus User Guide, http://www.cadence.com

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

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

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

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