Algorithms for Access Point Selection at Pre-Routing Stage

Marcelo Danigno[1], Mateus Fogaça[2,4], Eder Monteiro[4], Jorge Ferreira[2,4], André Oliveira[2,4], Ricardo Reis[2,3,4] and Paulo Butzen[1,2,5]
[1]Centro de Ciências Computacionais, Universidade Federal do Rio Grande
[2]PGMicro/[3]PPGC, [4]Inst. de Informática, [5]Dep. de Elétrica, Universidade Federal do Rio Grande do Sul
marcelo@furg.br, {mpfogaca, emrmonteiro, jaferreira, asdeoliveira, reis}@inf.ufrg.br, paulobuzen@ufrgs.br

***Abstract*—Pin accessibility has arisen as a major issue for detailed routing due to the increased number of design rules and higher pin density of modern technology nodes. The existing literature tackles this problem across many stages of the design flow: standard-cell layout, placement optimization and routing. In our present work, we cope with the pin accessibility challenge as a pre-processing step for detailed routing. We devise five techniques to find on-track locations in which the router can access I/O pins of standard cell instances without causing design rule violations (DRVs). We provide the algorithm complexity analyses for each of our techniques. Some of our methods tackle the complexity of checking the design rules using pessimism (i.e., they are subject to false positives). We also study the cases in which our methods are overly pessimistic and ways to overcome pessimism. Finally, we validate our techniques using the 45 and 32 nanometer designs from the ISPD 2018 contest on initial detailed routing and show that our most robust technique can find DRV-free access points for more than 99% of the pins of the test cases in less than 5 minutes.**

***Index Terms*— Electronic Design Automation; Physical Design; Detailed Routing; Pin Access.**

I. Introduction

Detailed routing is the most timing-consuming step in the physical implementation flow. Yet, detailed routers have to tackle with an ever-increasing set of design rules in modern technologies [1]. Failing to route a design successfully results in a solution with many design rule violations (DRVs). The DRVs are solved by either (i) being manually fixed by physical design engineers; or (ii) running an aggressive routabillity-driven placement; or even (iii) rethinking the floorplan. These solutions have a high and sometimes prohibitive turn-around-time.

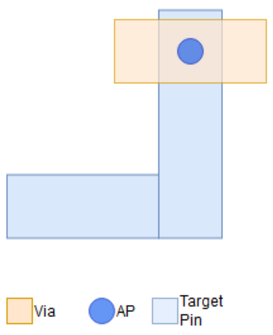
In modern technologies, the shrink of feature sizes allows an increased density of pins in the placement region and inside a standard cell. Additionally, the number of routing tracks in a standard cell has been significantly reduced. If we also take into consideration the complex technology rules, this raises a challenge called *pin accessibility*. The pin accessibility challenge consists of finding an on-track location in the routing grid in which the router can access the pins of an instance. This location is called an *access point* (AP). The connection from the access point to a pin is implemented using a via or a local wire. One common issue is that multiple pins may share the same access point. Figure 1(a) illustrates an access point of an instance using a via, Figure 1(b) depicts an access point of an instance using a local wire connection and Figure 1(c) shows two pins sharing the same access point.

Existing literature. Many works in the literature devise criteria to predict pin accessibility issues in placement [12, 18] and standard cell layout [6] so they can be avoided in previous stages of the design. Other research focus on how to optimize pin accessibility [3,17,21,22]. Our present work falls into yet another category that tackles the problem of *finding access point before detailed routing*.

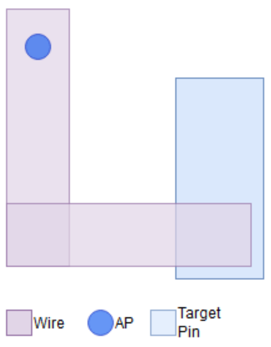
Our present work proposes five procedures to find violation-free access points. A preliminary version of our work appeared in [8]. We extend our paper of [8] by providing (i) an extended literature review, (ii) more detailed information about the design rules that our techniques support, (iii) adding an enhanced technique to find access points.

Our contributions are summarized as follows.

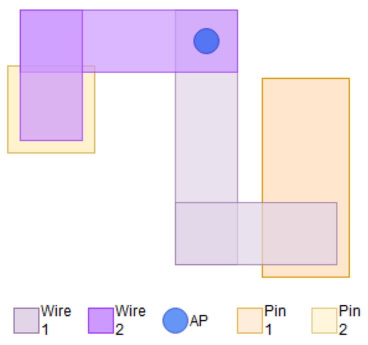
1. We provide a study and taxonomy about the literature on the pin accessibility challenge. Our studies comprehend techniques for *pin accessibility optimization* and *pin accessibility on a detailed router*.



(a) Connecting a pin to a via.



(b) Connecting a pin to an wire segment.



(c) Two pins sharing an AP.

Fig. 1: Possible connections from an AP.

Digital Object Identifier 10.29292/jics.v16i2.128

2. We devise five algorithms for finding DRV-free access points as a pre-processing step for detailed routing. We provide a study on the algorithms code complexity and pitfalls.

3. We validate our techniques the 45 and 32 nanometer designs from the ISPD18 testcase suite [14] and how that we can find access points for more than 99% of the pins in most of the testcases in less than 250 seconds with our more robust technique.

This work is organized as follow: Section II presents some essential details when computing APs and similar works that treat this problem. Section III explains the five procedures and their respective pseudo-codes and characteristics. The results obtained when running the five different procedures are shown in Section IV. Finally, we present our conclusions in Section V and in Section VI we present our future works.

II. Preliminaries

In this Section, we present some details on how we treat APs, an overview of the literature on pin accessibility, and some definitions on design rules. This is the groundwork for the procedures which will be shown on Section III.

A. *Problem Definition*

In this work, we generate APs to guide the detailed routing step. This is usually done with the support of a routing grid, which is a group of lines (creating a grid) that assists in the routing of metal layers. The routing grid specify discrete positions for wires to avoid design rules violations. To obtain an AP, first we need to determine what an AP is. In this work, an AP is a Cartesian point in the routing grid. These APs could then be used to connect pins or metal segments from different layers.

B. *Related Works*

We divide the literature into two categories: The first category comprehends optimization methods that aim to improve pin accessibility. The second category comprehend studies on how to perform pin access under high density of pins and complex design rules.

Optimization of pin accessibility.

Xu et al. [21] improves the pin accessibility of a 10-nm standard cell library modeling pin access as a via assignment problem. Seo et al. [17] redesigns standard cells with a high density of pins to improve pin accessibility and propose a measure of *inaccessibility of pins* to guide placement. Xu et al. [22] optimizes the pin accessibility of standard cells for unidirectional routing. Ding et al. [3] propose dynamic programming-based and linear programming-based detailed placement techniques to improve pin accessibility. Finally, Yu et al. [4] proposes a deep learning-based approach, that aims to avoid pin patterns with bad pin accessibility.

Pin accessibility in detailed routing.

Ozdal [16] proposes a *escape routing* technique to improve routability in regions with a high density of pins using a multi-commodity flow model and a Lagrangian relaxation algorithm. Nieberg [15] performs off-grid routing to find the nearest path from pin geometries to on-track locations while considering design rules. Xu et al. [23] address the pin accessibility problem under self-aligned double patterning constraints. More recently, Sun et al. [19], Kahng et al. [10], Bai et al. [2] and Li et al. [11] present alternative solutions to address pin accessibility during initial detailed routing in advanced technology nodes. Chan et al. [5] proposes a machine learning alternative that helps routability for Sub-14nm Process Node technologies, which includes techniques that deal with pin density and proximity.

Our present work falls into the second category and addresses pin accessibility for initial detailed routing algorithms, similarly to [19], [10], [2] and [11].

C. *Design Rules*

To explain each procedure, first we need to know how they interact with the following objects : pins, vias, design rules and access points. Pins and vias are treated as simple polygons, which are placed in the design and have to follow certain rules. These would be the design rules. For this experiment, we consider:

- Minimum Area : a routing segment has to have an area higher than a pre-defined threshold.

- Short : a routing segment has to be connected to a specific net depending on the logic of the circuit.

- Float : a routing segment needs to be connected to another one through an access point, since it is lined with the routing grid, or the beginning or end point of the geometry, as specified in the design file.

- Minimum Spacing (MS): a routing segment needs to be a certain distance away in one axis from others when not connected.

- End-Of-Line Spacing (EOL): The beginning and end of a routing segment needs to be a certain distance away in two axis from others when not connected.

All this rules are obtained from a technology file. Finally, the access points, as explained before, are a Cartesian point in the routing grid. They are not a physical object — i.e. not placed in the design. APs are used solely for support on the procedures, since they can be easily categorized (in valid or invalid) and are always inside the routing grid. An example of all these objects is presented in Figure 2.

From Figure 2, we can see a simple example where we only consider one type of via. When placed on an invalid AP (the red dots), it would cause a minimum spacing violation. Thus, for this example, the only valid placement would be on the bottom right AP. For the current work however, all types of vias in the technology file are considered. The library used also has different via definitions for each orientation, thus testing all possible combinations of via and AP.

For the procedures, we will use two different example topologies with one valid AP. These are simple but present some characteristics that are important to take in mind when validating a via. They can be found on Figure 3. The first topology is a simple two-rectangle pin, while the second one uses three rectangles.

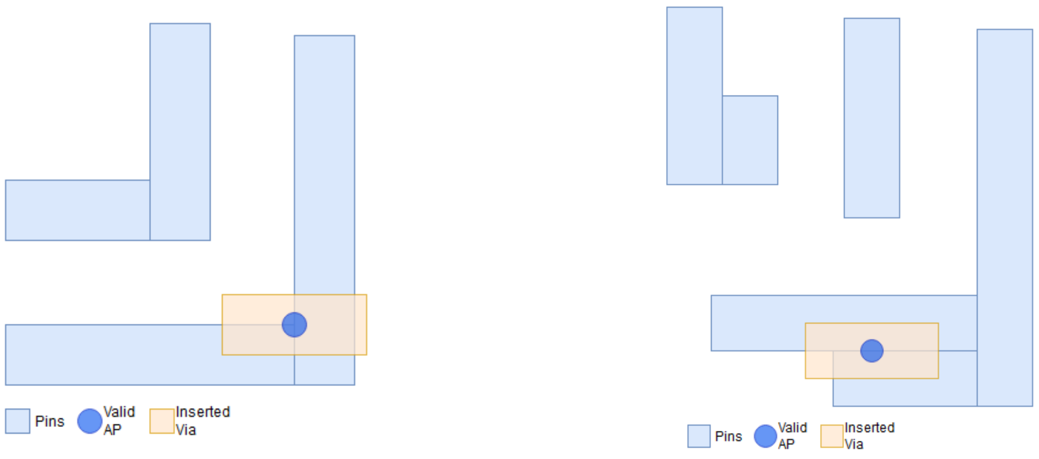


Fig. 3: Example topologies.

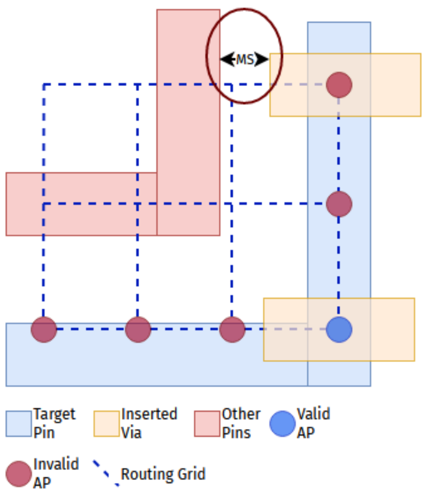


Fig. 2: Example of two via placements, one in a invalid AP and another in a valid one.

III. Methodology

In this Section, we present the five proposed procedures created to find a valid and violation-free via for a given AP. Four of these methods are called by another function which is applied for each pin of the circuit. The inputs are the possible vias for the layer and a pin. The function in question is described in Algorithm 1.

Algorithm 1 Via Placement in AP

```
1: function VIAPLACEMENT(via_library,Pin)
2:     for AP in Pin do
3:         for Via in via_library do
4:             Compute Method                ▷ Runs a procedure
5:             if Via is valid then
6:                 Add Feasible Via
7:             end if
8:         end for
9:     end for
10: end function
```

This function iterates through all APs inside a pin geometry, then computes a violation check procedure for each via of the technology file. It is important to note that a via is considered valid if there is no violation of other geometries or other vias. This test includes the different layers that compose the via. If the via is valid, it is added to a list of feasible vias. The via could also be placed on to the layout as soon as it is established that it is valid. This, however, could cause possible drawbacks, such as being a potential obstacle for future vias or raising the computational effort when computing other APs. For the current work, vias are placed as soon as they are defined as valid and then the algorithm runs for another pin. This new pin will consider the placed via as a new obstacle (defined in the internal structures for each procedure).

This algorithm results in a worst-case complexity of $\mathcal{O}(n \times k \times \mathcal{O}v)$, where n is the number of AP in a pin, k is the number of vias for a specific layer (defined in the technology), and $\mathcal{O}(v)$ is the worst-case complexity of the violation check function (any VIOLATIONCHECK function of the procedures in this section). For example, k could be the number of vias that connect layer 2 (metal) to layer 3 (metal) and n could be the number of AP in the output of a cell.

A. *Axis-Aligned Bounding Boxes Collision Test (AABB)*

The first procedure consists in the creation of four hypothetical test rectangles. These rectangles are created as an extension of the via, and they define the area where a violation could occur, based on the technology rules. They are then used on a simple axis aligned bounding box collision detection algorithm that checks whether or not a violation happens. An example of the process is shown in Figure 4.

Figure 4 consist of manipulations between rectangles of four colors: red, green, yellow and white. The yellow rectangle is the via. The red area generates a violation when intersecting with the white rectangles, which represents the MS (also called metal to metal spacing) and EOL rules. The green area represents the area where a violation cannot occur. Since the violation could occur both vertically and horizontally, this work defines them as horizontal spacing and vertical spacing. For AABB, the red area is defined by all pin geometries that are not from the same net or do not intersect with the via.

The algorithm flow for AABB is described in Algorithm 2.

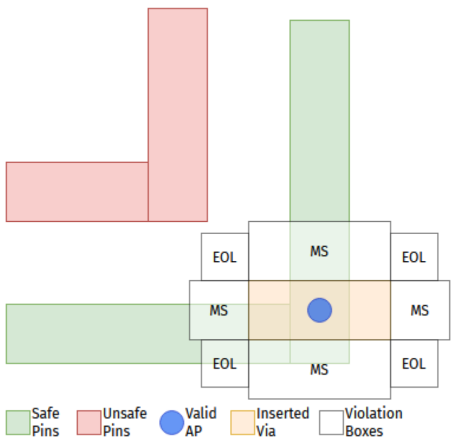


Fig. 4: Example of a simple violation check.

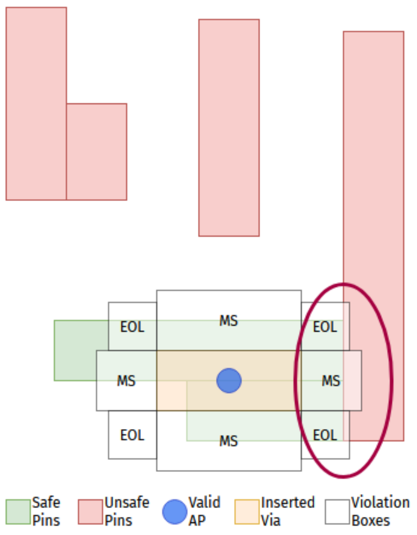


Fig. 5: Example of a false violation. In this case the pin geometry is composed of multiple rectangles, causing the algorithm to fail.

The procedure consists of creating the four rectangles based on the design rules of the technology files (Lines 2-5). These include horizontal spacing, vertical spacing and, if needed, EOL spacing (either horizontally, vertically or both; applied or not based on the minimum EOL width). After obtaining the rectangles, the algorithm checks for a collision between the rectangles and the cell geometries (including possible power lines) using an AABB collision check (Lines 6-12). If a collision occurs, and the geometry responsible for it doesn't collide with the via itself, the via is considered invalid.

Algorithm 2 AABB

```
1:  function VIOLATIONCHECK(Via,Cell)
2:      CollisionBox[0] = via + HorizontalRules
3:      CollisionBox[1] = via + VerticalRules
4:      CollisionBox[2] = via + HorizontalEOLRules
5:      CollisionBox[3] = via + VerticalEOLRules
6:      for Pin in Cell do
7:          for Box in CollisionBox do
8:              if Box collides with Pin then
9:                  Via is not Valid
10:             end if
11:         end for
12:     end for
13: end function
```

From Algorithm 2, we can see that the complexity of this method is high due to having to iterate over all the pin geometries, even those that are more than one metal spacing (minimum distance from metal to metal of the same layer) away. The worst case complexity is $\mathcal{O}(4 \times p)$, being p the number of pins in the specific cell. This method also results in possible false violations (a via that is entirely valid causing a false positive in the code), as a consequence of collision checks of parts of the geometries that are completely inside the pin bounds. These geometries that are also part of the pin itself cannot cause a violation. An example of this problem is presented in Figure 5.

Additionally, this implementation only performs violation checks for pins of the current cell. This may cause a via to be valid when, in reality, a pin from an adjacent cell could cause a violation with it.

B. *Edge Extension Collision Test (EE)*

EE has been designed to solve the false violations that happens in AABB. AABB treats the via as a whole, including parts that do not cause a violation. EE, on the other hand, only treats the edges of the via that are outside the pin bounds. These edges are the only ones that can generate a violation.

This process requires a grid to be created, which consists of the x and y points of each vertex of the via and the polygons that could intersect with it. To be able to do that, each point of the grid can have three values: "1" means a pin polygon is present, "2" means a via polygon is present and "3" means an intersection between the pin and the via occurs in the point. With this new grid, it is possible to create edges using the points that are only within the via ("2") or within both the via and the pin ("3"). These edges are the ones that can cause a violation with other geometries. The only thing that can not be overlooked is that an edge should not be composed of only "3" nodes, meaning the edge is completely inside the pin geometries and the same false violations will occur.

With the created edges, we can check violations for specific directions. This direction depends on where the edge is located on the via. If the edge is from the top part of the via, only upper violations are checked. The same happens for all four directions. Edges that consist of a via vertex ("2") also require an EOL violation check. An example of the violation check is presented in Figure 6.

Figure 6 is similar to Figure 4, however, there are two main differences. The unsafe pin area is now all pin geometries from the design and the violation area extends only from the edges of the via that are not part of a pin.

The procedure uses a new structure: RTrees. Which is used to find geometries (from a bounding box) and to find out the ownership of a point (if it is part of a via, pin or both). This also solves the problem of neighboring cells not

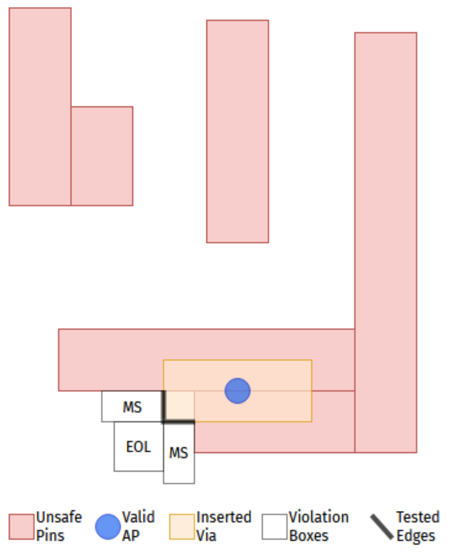


Fig. 6: Violation check used in EE.

being accounted for in violation checks. The algorithm flow is described in Algorithm 3.

Algorithm 3 EE

```
 1: function VIOLATIONCHECK(Via,Cell)
 2:     Geometries = RTree(via.neighborhood)
 3:     for X in Geometries do
 4:         for Y in Geometries do
 5:             Grid[X][Y] = RTree(X,Y).type
 6:         end for
 7:     end for
 8:     for line in Grid do
 9:         if line $\not\subseteq$ Geometries then
10:             Edges.append(line)
11:         end if
12:     end for
13:     for Edge e in Edges do
14:         e = e + Extension            ▷ Depends on direction
15:         if e collides with Geometries then
16:             Via is not Valid
17:         end if
18:     end for
19: end function
```

This algorithm bases all its calculations on a property called ”neighborhood”. This via.neighborhood is simply the via bounding box extended (both in X and Y) by the biggest (i.e. the most strict) rule in the design. With this, we can obtain only the geometries that can cause a violation.

It starts by iterating over all the combinations of X and Y points of the polygons found by the RTree. This includes points from different polygons. After that each point is checked by the RTree to see if an intersection is found. If it is found, we find the type of this intersection by analyzing which polygon caused it (being the types 1,2 or 3 that were mentioned before). This computations represents line 5 or the algorithm. After that step is done, it is easy to define lines by the rules presented before (edges are created using points that have type 2 or 3) and test them by a simple collision test (the same as in AABB).

From Algorithm 3 we can see that this method requires the creation of multiple supporting structures (Lines 2,5 and 10), increasing the overall complexity and memory footprint of the algorithm. Creating the grid requires multiples queries of the RTree (Lines 3-7) , which has complexity $\mathcal{O}(log(p))$, where p is the number of pins in the whole design. Generating the lines to be checked (Lines 8-12) also has a high complexity, with a worst case of $\mathcal{O}(c \times l)$, being c the number of columns in the grid and l the number of lines in the grid. Thus, the complexity of the algorithm becomes $\mathcal{O}(c \times l \times log(p))$. This method, however, does not cause false violations, resulting in a higher amount of vias placed.

C. *Polygon Subtraction Collision Test (PS)*

PS is a simplified EE. It aims to increase readability and reduce the number of queries to the RTree using polygons. A polygon is one or more geometries that describes a particular shape in the design. If A is the via polygon and B a polygon that consists of every pin of the cell, A-B results in the polygons that are part of the via but not a pin. These are the polygons which can cause a violation. By extending these polygons with the design rules we can check for an intersection with B for possible collisions. If the result of said calculation is not entirely within A, a violation was caused. An example of the violation check is presented in Figure 7.

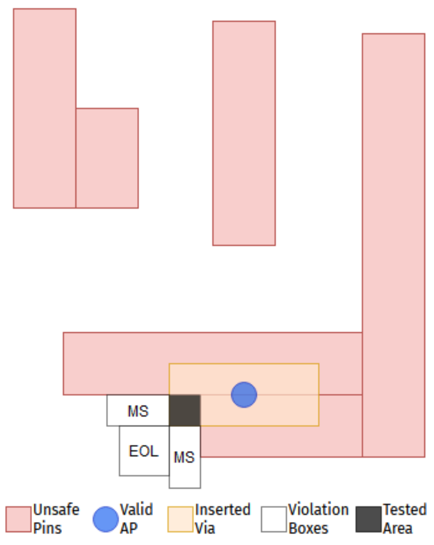
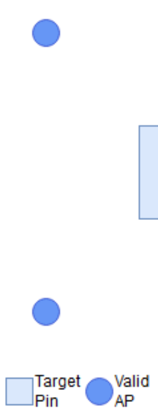


Fig. 7: Violation check used in PS.

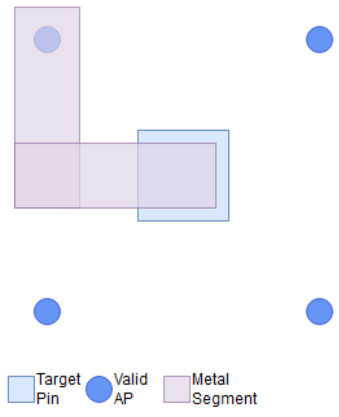
Figure 7 has the same concepts as Figure 6, however instead of dealing with edges of the via itself, it treats the black area. The black area is the difference between the via polygon and the pin polygon.

This method still uses the RTree to get the geometries to check and to find out the ownership of a point. However, it creates polygons with the resulting geometries to reduce the number of queries to the RTree. The algorithm flow is described in Algorithm 4.

From Algorithm 4, we can see that this procedure is more straightforward than EE. The worst-case complexity is $\mathcal{O}(log(p))$, being the complexity of the RTree query. However, it still requires multiple support structures, increasing



(a) Pin Polygon with no valid APs.



(b) Connecting metal segments from a neighboring AP.

Fig. 8: Extension present in WCE: Connecting an "L" shape to the pin geometry.

the overall computation time and memory footprint of the algorithm.

Algorithm 4 PS

```
 1: function VIOLATIONCHECK(Via)
 2:     B = RTree(via.neighborhood)
 3:     A = Via
 4:     R = A - B
 5:     for Poly in R do
 6:         Poly = Poly + Extension  ▷ Depends on direction and ownership
 7:     end for
 8:     C = R ∩ B    ▷ R was changed due to the extensions added
 9:     if C ∩ A != C then
10:         Via is not Valid
11:     end if
12: end function
```

D. *Enhanced Polygon Subtraction Collision Test (EPS)*

EPS removes the RTree and is based around the calculations presented in PS. Instead of creating a Rtree, this method creates a map of polygons for each layer and for each cell. In other words, this method has multiple different polygons, one for each cell, and each one of them consists of all the geometries of said cell. The overall procedure is the same as in PS. The algorithm flow is the same as presented in Algorithm 4.

The worst-case complexity is $\mathcal{O}(R)$, being R the number of polygons of the difference between A and B. R depends on the via.neighborhood, thus, a pin with multiple geometries around it is harder to compute than one that is alone (so cells like buffers are faster than other combinational cells). This method does not use a RTree to find the surrounding geometries and for ownership tests. It uses polygons and collision tests between them. Since the pin polygon is a list that consists of multiple rectangles, we can use a simple AABB collision test for each item of said list. This also removes the need for iterating through each query to create the polygons to check. The result is a reduction in both complexity and memory. Additionally, this implementation only performs violation checks for pins of the current cell, just like AABB. However, since it is faster to test, it is possible to limit the number of vias found by ignoring those too close to the margin of the cell. This is done by using the same extension (but this time to decrease a bounding box) as via.neighborhood.

E. *Wire Connection Extension (WCE)*

WCE is an extension of EPS. While EPS is fast, it still struggles to place a via on congested benchmarks. For some, it may be impossible to place one on the pin geometries without causing a violation. To fix this issue, a supporting algorithm was created to work alongside EPS, by using APs outside the pin geometries. This is done by changing the VIAPLACEMENT function from Algorithm 1 to run another loop if no valid via was found. This is done only for APs not in Pin (instead of those in the pin, which are used in line 2). Since this change is in the VIAPLACEMENT and not the VIOLATIONCHECK functions, any procedures could theoretically be used, however, for this experiment, we only applied WCE when using EPS.

The algorithm works as follows: first we select an AP outside the pin geometries. Then, we pick the pin polygon closest to that AP. After that we try to connect metal segments to that polygon in an "L" shape. If these shapes cause a violation, we test another AP. If they do not cause a violation, we can test placing a via on the AP. An example of this method is shown on Figure 8.

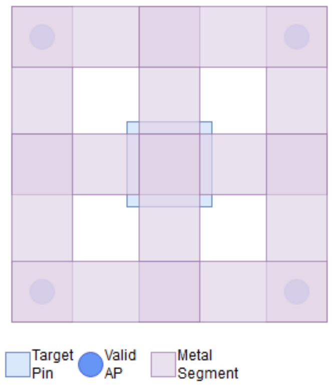


Fig. 9: All the possible "L" shape topologies.

Table I.: Access Point Generation, Runtime and Memory for the five procedures.

Benchmark	Pin	AABB			EE			PS			EPS			WCE		
		AP	Runtime (s)	Memory (MB)	AP	Runtime (s)	Memory (MB)	AP	Runtime (s)	Memory (MB)	AP	Runtime (s)	Memory (MB)	AP	Runtime (s)	Memory (MB)
ispd18_sample1	22	100.00%	<1	<1	100.00%	<1	<1	100.00%	<1	<1	100.00%	<1	<1	100.00%	<1	1
ispd18_sample2	27	100.00%	<1	<1	100.00%	<1	<1	100.00%	<1	<1	100.00%	<1	1	100.00%	<1	1
ispd18_sample3	8	62.50%	<1	<1	62.50%	<1	<1	62.50%	<1	<1	62.50%	<1	<1	100.00%	<1	<1
ispd18_test1	17203	96.79%	58	10	99.21%	80	18	99.21%	79	18	99.21%	1	2	100.00%	14	5
ispd18_test2	157990	91.21%	2737	46	99.06%	5086	126	99.06%	5143	127	99.06%	13	10	99.99%	158	38
ispd18_test3	158110	91.23%	2736	45	99.07%	5244	128	99.31%	5195	128	99.07%	13	9	99.99%	159	38
ispd18_test4	316652	96.32%	11663	92	96.19%	21972	250	96.41%	21544	249	95.22%	29	19	100.00%	339	188
ispd18_test5	316220	96.36%	11716	92	96.24%	21997	249	96.58%	21521	250	95.29%	33	20	100.00%	240	188

From Figure 8, we can see an example of a pin that has no AP within it. Thus, this extension tests AP outside the pin geometry by connecting the AP to the pin with an ”L” shape. This shape is created by two metal segments. If the metal segments fail to be placed (a violation is found), the algorithm tests another topology of the ”L” shape or tries another AP. For the example of Figure 8, we can see all the possible metal connections on Figure 9.

This changes the complexity from the base function from $\mathcal{O}(n \times k \times \mathcal{O}v)$ to $\mathcal{O}((n \times k \times \mathcal{O}v) + (m \times k \times 4 \times \mathcal{O}v))$, being m the number of APs not within any pin and 4 the total number of different rectangles that can be tested for a specific AP (as seen on Figure 9).

This extension greatly increases the number of vias placed. However, as we can see from the complexity increase, it has to call the violation test function multiple times (one time for each polygon of an ”L” shape). Thus, this increase has a big performance hit. It is important to know, though, that this procedure only uses this extension when no possible vias are found inside the pin geometries. This means the effect of this extension is only found when computing a small part of the circuit (around 5%, from what we will see in Section V).

IV. Results

To compare the five procedures, the results have been computed for ISPD 2018 [14] benchmarks. These results are presented in Table I. They were extracted using the Rsyn framework [7]. Table I contains the name of the benchmark, the number of pins of the benchmarks, and, for each procedure: the percentage of pins that had at least one valid AP (for each method), the runtime and the memory.

The five procedures, shown in Table I found valid APs (violation-free) for more than 60% of the pins in the designs. /blueWhen analyzing the resulting design files, most of the pins where no via is placed had no valid AP inside the pin geometries (the case for ispd18_sample3), which is why WCE shows the better results. This is mainly due to small buffer cells. The other occurrences are either as a result of the method used or because the pin and geometries around it were congested.

When it comes to individual analysis, AABB results in a reduced number of vias placed because of its logic. While it is simple, it causes false violations, which lowers the number of vias placed, especially for the bigger benchmarks. EE, PS and EPS display similar results. While PS has less calls to the RTree than EE, both end up having similar runtimes because of the complex polygon calculations done in PS (while EE only has simple rectangle tests). By removing the RTree and simplifying the logic, EPS yields a significant lower runtime and memory footprint than its predecessor (PS), while increasing the overall readability and maintainability of the code (when compared to EE). Both methods don’t have false violations presented in AABB.

WCE, in the other hand, is a major increase in the number of vias placed. For the bigger benchmarks, this increase can be as high as 15000+ APs. However, that addition has a medium impact on the run-time. The main problem of the 0.01% of pins that WCE couldn’t compute is congestion. If a pin has a small number of APs, and every one of them cause a violation with surrounding geometries (most of times with the pin itself), possible fixes would be a rectangular metal path to complete the area where the violation happens (if the logic allows it) or off-grid calculations.

V. Conclusions

Given the presented results, it can be seen that computational complexity and care for design rules must be taken in though when designing any algorithm revolving around routing [9] and access point generation. One simple false positive case could fail to compute multiple pins in a design of considerable size. When considering the complexity of routing algorithms and the results presented, a trade off between run-time and via placement is important. If a cell is too congested to be routed to, WCE could easily solve that problem in a small time frame. However, using it for the whole circuit could have a big impact in run-time. Which is why procedures such as EPS, which are simple and fast, can have a big advantage in re-routing computations.

VI. Future Works

For this work, two topics can be focused on to improve both the quality of the results and the runtime. First, the structure used to defined the pins or cells considered for each method can be changed to better treat problems like neighboring cells (which happens when using a map) and runtime (which is hit when using RTrees). This would be a structure that considers hierarchical statements such as rows and cells and has fast query times when in-memory.

Second, the algorithm to define the start and end point for WCE can be changed to both have a smaller wirelenght and an easier time to find valid APs.

It is also important to note that the procedures presented in this work could also be improved by using multi-threading since the computation behind each AP is specific for each pin.

Finally, a study on the effect of the procedures (or a combination of them) can be done when using the violation-free vias they have found during detailed routing.

Acknowledgements

This study was financed in part by Brazilian federal government agencies: Fundação de Amparo a Pesquisa do Rio Grande do Sul (FAPERGS), Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), and Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) - Finance Code 001.

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*, 2019, pp. 7–12.

[2] X. Bai, D. Xiao, W. Zhu and J. Chen, “An Initial Detailed Routing Algorithm Considering Advanced Technology Nodes”, *Proc. CSTIC*, 2019, pp. 1–3.

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

[4] T. C. Yu, S. Y. Fang, H. S. Chiu, K. S. Hu, P. H. Y. Tai, C, C. F. Shen, and H. Sheng. ”Pin accessibility prediction and optimization with deep learning-based pin pattern recognition.” *Proc. DAC*, 2019, pp. 1–6.

[5] W. T. J. Chan, P. H. Ho, A. B. Kahng, and P. Saxena. ”Routability optimization for industrial designs at sub-14nm process nodes using machine learning.”, *Proc. ISPD*, 2017, pp. 15–21.

[6] S. Fang, C. Tai and R. Lin, “On Benchmarking Pin Access for Nanotechnology Standard Cells”, *Proc. ISVLSI*, pp. 237–242.

[7] G. Flach, M. Fogaça, J. Monteiro, M. Johann and R. Reis, “Rsyn: An Extensible Physical Synthesis Framework”, *Proc. ISPD*, 2017, pp. 33–40.

[8] M. Danigno, P. Butzen, J. Ferreira, A. Oliveira, E. Monteiro, M. Fogaça and R. Reis, “Proposal and Evaluation of Pin Access Algorithms for Detailed Routing”, *Proc. ICECS*, 2019.

[9] A. B. Kahng, “Research directions for coevolution of rules and routers’, *Proc. ISPD*, 2003, pp. 122—125.

[10] A. B. Kahng, L. Wang and B. Xu, “TritonRoute: An Initial Detailed Router for Advanced VLSI Technologies”, *Proc. ICCAD*, 2018, pp. 1–8.

[11] H. Li, G. Chen, B. Jiang, J. Chen and E. F. Y. Young, “Dr. CU 2.0: A Scalable Detailed Routing Framework with Correct-by-Construction Design Rule Satisfaction”, *Proc. ICCAD*, 2019, pp. 1–7.

[12] R. Lin and Y. Chiang, “Impact of Double-Row Height Standard Cells on Placement and Routing”, *Proc. ISQED*, 2019, pp. 317–322.

[13] C. Lin, M. Tsai, K. Lee, T. Chen, T. Wang and Y. Chang, “Recent Research and Emerging Challenges in Physical Design for Manufacturability/Reliability”, *Proc. ASPDAC*, 2007, pp. 238–243.

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

[15] T. Nieberg, “Gridless Pin Access in Detailed Routing”, *Proc. DAC*, 2011, pp. 170–175.

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

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

[18] H. Su, S. Nishizawa, Y. Wu, J. Shiomi, Y. Li and H. Onodera, “Pin Accessibility Evaluating Model for Improving Routability of VLSI Designs”, *Proc. SOCC*, 2017, pp. 56–61.

[19] F.-K. Sun, H. Chen, C.-Y. Chen, C.-H. Hsu and Y.-W. Chang, “A Multithreaded Initial Detailed Routing Algorithm Considering Global Routing Guides”, *Proc. ICCAD*, 2018, pp. 82:1—82:7.

[20] X. Sun and F. Lombardi, “Design for testability of sequential circuits”, *IEE Proc. E* 141(3) (1994), pp. 153–160.

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

[22] 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.

[23] 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. DAES* 21(3) (2016), pp. 42:1–42:21.