Proposal and Evaluation of Pin Access Algorithms for Detailed Routing

Marcelo Danigno, Paulo Butzen
Centro de Ciências Computacionais
Universidade Federal do Rio Grande
Rio Grande, Brazil
{marcelo, paulobutzen}@furg.br

Jorge Ferreira, André Oliveira, Eder Monteiro, Mateus Fogaça, Ricardo Reis
Instituto de Informática
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil
{jaferreira, andre.oliveira, emrmonteiro, mpfogaca, reis}@inf.ufrgs.br

***Abstract*—Routing is a crucial step in the physical design of integrated circuits (ICs). Modern electronic design automation tools delve deep into detailed routing algorithms and pin access methods. The need for fast and efficient algorithms is an emerging challenge for physical design as design rules become more complex. The use of a pre-processing step to find pin access points helps in reducing the problem of detailed routing. This work proposes four different methods for access point generation and presents possible pitfalls when approaching the problem.**

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

I. Introduction

With the steady increase in IC design complexity and the creation of new devices, components, and techniques, modern ICs are becoming even more complex. Factors such as process scaling, design for testability, variability, and manufacturability have raised the speed and reliability of ICs while decreasing their cost. These factors have drastically increased the complexity of circuits, raising the need for fast and efficient EDA (electronic design automation) tools. The usage of EDA tools is so prominent that an improvement in EDA software directly correlates in an overall better IC and the possibility of creating even more intricate circuits [1].

In the physical design, there are several steps to design an IC. One of those stages is the place and route. Placement is the step in which the locations of cells and blocks are defined. Blocks can be, for example, intellectual property (IP), while cells are pre-characterized logic functions implemented in a given technology. For technologies higher than 20nm, a standard cell library consists of several different logic gates with a fixed height. These cells are then placed on the floorplan, with performance and wire-length in mind. The standard cell methodology reduces a lot of effort when using EDA tools, resulting in shorter design times and an easier assembly of the layout [2]. After placement, the routing stage is the one that connects cells using metal wires and vias. These connections are made while considering the resulting wire-length and possible design violations. To reduce the complexity of the calculations, it is possible to treat wire routing and pin access as two different things. This reduces potential violations and decreases the stress in the routing algorithm. Wire routing is connecting two or more pins with metal in a specific layer. Pin access consists of defining possible areas where a pin could be accessed by either via or by contact. Those areas (pin access areas) are called access points (AP).

This paper proposes four different methods of finding violation-free APs. The objective is to compare the presented procedures in performance, memory, and computational complexity while emphasizing possible pitfalls in the creation of a via placement algorithm.

This work is organized as follow: Section II presents some essential details when computing APs while also showing similar works that treat this problem. Section III explains the four procedures and their respective pseudo-codes and characteristics. The results obtained when running the four different methods are shown in Section IV. The conclusion is presented in Section V.

II. Background

In this section, we present the theoretical foundation that this paper is based on. For this work, we generate AP's to guide the detailed routing. The detailed routing step is when it is determined the exact route that a wire takes when connecting two or more pins. 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. We also only generate access points inside a pin geometry. These APs could then be used to connect pins or metal segments from different layers. This definition is not the same as the theoretical approach presented in Section 1, where an AP is just the pin access area (the whole pin geometry).

There is an extensive literature in AP generations strategies [3] [4] [5], and each of them treats APs as different elements in the routing flow. This paper uses a greedy implementation to find the best via that could be placed in a particular pin. For each AP, the algorithm tests the via with the biggest effective area. This causes the algorithm to test multi-cut vias first (a group of redundant vias), potentially increasing reliability and yield. If no violation occurs, that via would be placed. If a violation happens, the algorithm tests the next via in the list (ordered by area). This results in a worst-case complexity of

$\mathcal{O}(n \times k \times \mathcal{O}vc)$, where n is the number of AP in a pin, k is the number of vias for a specific layer (defined in the technology file), and $\mathcal{O}vc$ is the worst-case complexity of the violation check function.

III. Methodology

In this Section, we present the four proposed methods created to find a valid and violation-free via for a given AP. 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    ▷ Only for APs inside pin bounds
3:          for Via in via_library do
4:              Compute Method  ▷ Runs 1 of the 4 methods
5:              if Via is valid then
6:                  Add Feasible Via
7:              end if
8:          end for
9:      end for
10: end function
```

It is important to note that a via is considered valid if there's no violation with 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.

A. Procedure 1 (P1)

P1 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 (AABB) collision detection algorithm that checks whether or not a violation happens. An example of the process is shown in Figure 1.

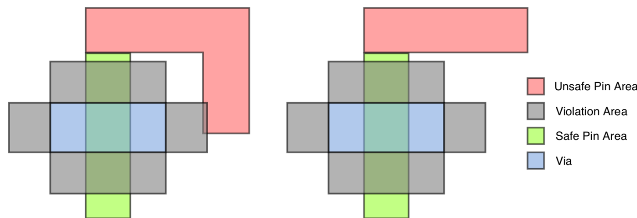

Fig. 1: Example of a simple violation check. The figure on the left shows a violation while the right figure shows a valid via placement.

The green area presented in Figure 1 represents the area where a violation cannot occur. The red area, on the other hand, generates a violation when intersecting with the grey rectangles. The violation area in this image represents the metal to metal spacing (also called min. spacing) rules. Since the violation could occur both vertically and horizontally, this work defines them as horizontal spacing and vertical spacing. For P1, the red area is defined by all pin geometries that are not from the same net or do not intersect with the via.

The procedure consists of creating the four rectangles based on the design rules of the technology files. These include horizontal spacing, vertical spacing and, if needed, End-Of-Line (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. If a collision occurs, and the geometry responsible for it doesn't collide with the via itself, the via is considered invalid. The algorithm flow is described in Algorithm 2.

Algorithm 2 Procedure 1

```
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 cell 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 $\mathcal{O}(4 \times np)$, being np 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 2.

Additionally, this implementation only does 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. Procedure 2 (P2)

P2 was designed to solve the false violations that happen in P1. P1 treated the via as a whole, including parts that don't cause a violation. P2, on the other hand, only treats the edges of the via that are outside the pin bounds. These edges are the

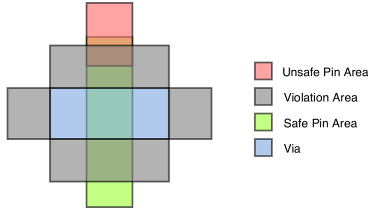

Fig. 2: Example of a false violation. In this case the pin geometry is composed of two rectangles, causing the algorithm to fail.

only ones that can generate a violation. This process requires a new grid to be created, which consists of the x and y points of each polygon that intersects with the via. With this new grid, it is possible to create edges using the points that are only within the via or within both the via and the pin. These edges are the ones that can cause a violation with other geometries. With the created edges (excluding those completely inside the pin geometries), 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 and another point also require an EOL violation check. An example of the violation check is presented in Figure 3.

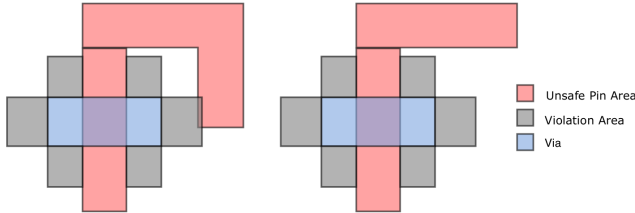

Fig. 3: Violation check used in P2. The figure on the left shows a violation while the right figure shows a valid placement.

Figure 3 is similar to Figure 1, 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 also uses RTrees to get the geometries to check and to find out the ownership of a point (if it is part of an via, pin, both or neither). This also solves the problem of neighboring cells not being accounted for in violation checks. The algorithm flow is described in Algorithm 3.

From Algorithm 3 we can see that this method requires the creation of multiple supporting structures, increasing the overall complexity and memory footprint of the algorithm. Creating the grid requires multiples queries of the RTree. Generating the lines to be checked 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. This method, however, does not cause false violations, resulting in a higher amount of vias placed.

Algorithm 3 Procedure 2

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

C. Procedure 3 (P3)

P3 is a simplified P2. 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 4.

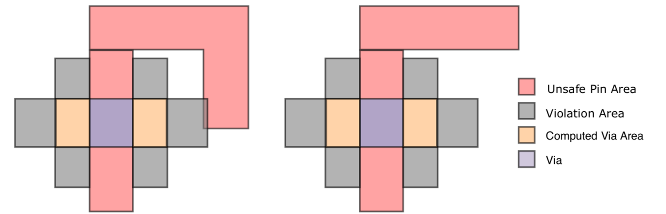

Fig. 4: Violation check used in P3. The figure on the left shows a violation while the right figure shows a valid placement.

Figure 4 has the same concepts as Figure 3, however instead of dealing with edges of the via itself, it treats the orange area. The orange 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 P2. The worst-case complexity is $\mathcal{O}(R)$, being R the difference between A and B; however, it still

TABLE I: Access Point Generation, Runtime and Memory for the four methods.

Benchmark	Pin	P1			P2			P3			P4		
		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%	0.0002	0	100.00%	0.0005	0	100.00%	0.0010	0	100.00%	0.0089	0
ispd18_sample2	32	100.00%	0.0002	0	100.00%	0.0006	0	100.00%	0.0012	0	100.00%	0.0098	0
ispd18_sample3	12	75.00%	0.0001	0	75.00%	0.0001	0	75.00%	0.0002	0	75.00%	0.0007	0
ispd18_test1	17203	96.79%	57.5868	10	99.21%	79.7906	18	99.21%	78.7555	18	96.14%	3.8561	2
ispd18_test2	157990	91.21%	2737.4500	46	99.06%	5086.0700	126	99.06%	5143.2700	127	91.37%	26.5994	9
ispd18_test3	158492	91.23%	2736.4400	45	99.07%	5243.8800	128	99.07%	5195.0300	128	91.35%	26.6865	9
ispd18_test4	317034	96.32%	11662.5000	92	96.19%	21971.7000	250	96.30%	21544.1000	249	73.22%	84.6997	14
ispd18_test5	316984	96.36%	11715.5000	92	96.24%	21996.5000	249	96.34%	21521.3000	250	73.84%	78.0911	14
ispd18_test6	474330	96.35%	26586.8000	137	96.22%	49041.3000	371	96.33%	48620.9000	372	73.54%	117.1900	22

Algorithm 4 Procedure 3

```
1: function ViolationCheck(Via)
2:     B = RTree(via.location)
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 n B
9:     if C n A != C then
10:        Via is not Valid
11:    end if
12: end function
```

requires multiple support structures, increasing the overall computation time and memory footprint of the algorithm.

D. Procedure 4 (P4)

P4 removes the RTree and is based around the calculations presented in P3. 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 P3. The algorithm flow is the same as presented in Algorithm 4.

While the worst-case complexity is the same as P3, 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.

IV. Results

To compare the four methods, the results have been computed for ISPD 2018 [6] benchmarks. These results are presented in Table I. 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 four procedures, shown in Table I found valid APs (violation-free) for more than 70% of the pins in the designs. When analyzing the resulting design files, most of the pins where no via is placed had no valid AP inside the pin geometries, requiring a supporting off-grid algorithm to compute. 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, P1 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. P2 and P3 display similar results. By removing some uses of the RTree and simplifying the logic, P3 yields a slightly lower run-time, while increasing the overall readability and maintainability of the code. Both methods don't have false violations presented in P1. However, their drawback is in the computational complexity, resulting in an increased run-time when comparing to P1. P4, on the other hand, removes the need for a RTree and simplifies the code even further. This results in a significant lower run-time and memory footprint. However, using a list of polygons generates some false positives, since a polygon is treated as multiple rectangles instead of only one object.

V. Conclusion

Given the results presented, it can be seen that computational complexity and care for design rules must be taken in though when designing any algorithm revolving around routing 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, the only valid procedure would be P4. It is important to note that the procedures presented in this work could also be improved by using multithreading since the computation behind each AP is specific for each pin.

References

[1] D. Jansen, *The electronic design automation handbook*. Springer Science & Business Media, 2010.

[2] A. E. Dunlop and B. W. Kernighan, "A procedure for placement of standard cell vlsi circuits," *IEEE Transactions on Computer-Aided Design*, vol. 4, no. 1, pp. 92–98, 1985.

[3] X. Sun and F. Lombardi, "Design for testability of sequential circuits," *Proc. IEEE - Computers and Digital Techniques*, vol. 141, no. 3, pp. 153–160, 1994.

[4] A. B. Kahng, L. Wang, and B. Xu, "Tritonroute: an initial detailed router for advanced vlsi technologies," in *2018 IEEE/ACM ICCAD*, 2018, pp. 1–8.

[5] 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 TODAES*, vol. 21, no. 3, pp. 42–63, 2016.

[6] S. Mantik, G. Posser, W.-K. Chow, Y. Ding, and W.-H. Liu, "Ispd 2018 initial detailed routing contest and benchmarks," in *Proc. ISPD*, 2018, pp. 140–143.