CUSTOM-QUALITY WIRE ROUTING USING MODERN DESIGN RULES

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Christopher Charles LaFrieda

August 2005

ABSTRACT

This thesis presents a wire routing methodology that produces custom-quality results. We use a gridless tile-based approach that extends previous works in four main ways. First, it captures all the intricacies of modern design rules, e.g. the difference between contact-to-contact spacing and contact-to-wire spacing. Second, it implements a robust cost model that includes: i) horizontal wire costs, ii) vertical wire costs, iii) via costs, and iv) jog costs. Third, a design-rule correct route is always guaranteed even if the search for the least-cost path is terminated early. Fourth, route ordering is dynamically updated based upon the routability of nodes. The resulting router is shown to route 1.5-11x faster than the Cadence Chip Assembly Router while consuming 6-8x less memory with 5-15% less wiring overhead.

BIOGRAPHICAL SKETCH

Christopher Charles LaFrieda attended Midwood High School in Brooklyn, NY. He graduated in 1997 from their Medical Sciences Program. Christopher then began his undergraduate studies at the State University of New York at Buffalo. In May of 2001, Christopher graduated from SUNY at Buffalo with a B.S. in both Computer Science and Computer Engineering.

Currently, Christopher is in the M.S./Ph.D. program of the School of Electrical and Computer Engineering at Cornell University. He is a member of the Asynchronous VLSI design group, under the tutorage of Professor Rajit Manohar.

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor, Prof. Rajit Manohar, for taking a wayward graduate student under his wing. For all our discussions and debates, and for his continued support in all my research endeavors, I thank him. I would also like to thank the other members of my committee, Sally McKee and Martin Burtscher, for our collaborations, their critiques of my work, and helping to make the Computer Systems Lab (CSL) here at Cornell the success it is today.

I would also like to thank the other members of the AVLSI research group: Filipp Akopyan, David Biermann, Virantha Ekanayake, David Fang, Clint Kelly, Song Peng, and John Teifel. Their hardwork, dedication and eagerness to help has created an environment conducive to the unbridled exchange of ideas and concepts. I have no doubt that they will all go on to do great things.

On a more personal note, I would like to thank my parents, Patrick LaFrieda and Barbara Galvagni, my brothers and sister, Patrick, Joseph and Michele LaFrieda, for supporting me in all ways possible.

TABLE OF CONTENTS

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Related Work . . . 2
- 1.3 Core Contributions . . . 3
- 1.4 Organization of Thesis . . . 4

2 Preliminaries **5**
- 2.1 Corner-Stitched Data Structure . . . 6
  - 2.1.1 The Corner-Stitched Tile . . . 6
  - 2.1.2 Tile Planes . . . 7
- 2.2 The CONTOUR Router . . . 8
  - 2.2.1 Preprocessing Geometry . . . 8
  - 2.2.2 Finding Initial Paths . . . 9
  - 2.2.3 Propagating Paths . . . 10
- 2.3 Limitations of CONTOUR . . . 12
  - 2.3.1 Excessive Jogs . . . 13
  - 2.3.2 Design Rule Violations . . . 14
  - 2.3.3 Notches . . . 14
  - 2.3.4 Loops . . . 15

3 The Router Framework **17**
- 3.1 Data Structures . . . 17
  - 3.1.1 3D Tiles . . . 17
  - 3.1.2 Explicit Oxide Layers . . . 19
  - 3.1.3 Extensions to Path Structure . . . 19
- 3.2 New Tile Types . . . 20
  - 3.2.1 Terminal Tiles . . . 20
  - 3.2.2 Boundaries . . . 21
- 3.3 Framework Configuration . . . 21
  - 3.3.1 Layers . . . 21
  - 3.3.2 Materials . . . 22

4 Proposed Routing Methodology **24**
- 4.1 Node-to-Node Routing . . . 24
  - 4.1.1 Preprocessing Geometry . . . 24
  - 4.1.2 Finding Initial Paths . . . 27
  - 4.1.3 Path Propagation . . . 29
  - 4.1.4 Cost Model . . . 31
  - 4.1.5 Route Drawing . . . 32
- 4.2 Route Management . . . 33
  - 4.2.1 Route Ordering . . . 34
  - 4.2.2 Sorting Algorithm . . . 35

5 Results **37**
5.1 Benchmark Circuits 37
5.2 Comparison With Cadence Chip Assembly Router 38
5.2.1 Run Time 39
5.2.2 Memory Usage 40
5.2.3 Total Wire Length 41
5.3 Quality of Layout 41

6 Conclusions **44**
6.1 Conclusions 44

A The Technology File **45**
A.1 Parameters 45
A.2 Layers Block 46
A.3 Materials Block 46
A.4 Contacts Block 47
A.5 Translations Block 47

Bibliography **48**

LIST OF TABLES

3.1 Framework layers. 22
3.2 Framework materials. 23

5.1 Benchmark circuits. 38

LIST OF FIGURES

1.1 The design cycle for the SNAP microprocessor. 2

2.1 A corner stitched tile. 6
2.2 An example of a horizontal tile plane. 7
2.3 An example of a vertical tile plane. 8
2.4 Adding contours to material tiles. 9
2.5 An example of initial wire paths. 10
2.6 Path propagation between two nodes. 12
2.7 Two possible paths with the same cost. Path A has four turns, but Path B has only one. 13
2.8 A wire spacing violation. 14
2.9 A contact spacing violation. 14
2.10 A contact-to-contact notch. 15
2.11 A contact-to-elbow notch. 15
2.12 An example of a loop between closely placed nodes. 15

3.1 The 3D corner-stitched tile. 18
3.2 An example of explicitly representing oxide layers as tile planes. . . 19
3.3 Path propagation through a horizontal terminal tile. 21

4.1 The major stages of the routing algorithm. 25
4.2 The minimum spacing of a wire to itself is $2 * spacing + width$ units. 26
4.3 Geometry of a terminal after preprocessing. 27
4.4 Finding initial paths. 28
4.5 Centerline segments created by a new path are bloated to check for spacing violations with centerline segments created by previous paths. 30
4.6 Two examples of new paths that may create jogs. The new path in the top image will create 1 or 2 jogs. The new path in the bottom image will create no jogs or 1 jog. 32
4.7 An example of metal fill for contact-to-wire notches. 33
4.8 Route Ordering. A: All possible routes added to list and sorted. B: $a0 <=> a1$ is unroutable so the next pair, $a0 <=> a2$ is attempted. C: $a0 <=> a2$ routes, nodes $a0$ and $a2$ are removed and replace with $a3$. 35
4.9 Routes are ordered so that harder to route nodes at attempted first. 36
4.10 Routability is determined by searching the area just outside of a node's contour for space tiles. 36

5.1 Run time for our router versus Cadence. 40
5.2 Memory usage for our router versus Cadence. 41
5.3 Wire overhead for our router versus Cadence. 42

5.4 Metal-One geometry for a benchmark circuit with node density of .018. 43
5.5 Metal-Two geometry for a benchmark circuit with node density of .018. 43

Chapter 1

Introduction

1.1 Motivation

Designing VLSI systems is an increasingly daunting task. Current technologies can accommodate hundreds of millions of transistors in an IC. Design automation is often used to manage this complexity. Unfortunately, synthesized designs usually perform a few times worse than custom designs across all metrics. However, some recent research has shown that mixing both custom design styles and automated design styles yields promising results[3, 14]. Consequently, we believe that layout automation tools should be versatile enough to allow designer interaction.

Recently the AVLSI group at Cornell University designed and fabricated an asynchronous sensor network microprocessor called SNAP[5]. A breakdown of the time spent in each stage of the design cycle is shown in Figure 1.1. Approximately 60% of the total time to design and fabricate this processor was spent doing physical layout. It has been estimated that routing non-critical wires accounted for nearly 90% of time to complete the physical layout. Obviously, minimizing the time spent routing non-critical wires will result in the largest reduction in our design cycle. Hence we aim to automate wire routing with this work.

Traditionally, there are two main approaches to routing[13, 6]: uniform-grid and gridless. The uniform-grid approach is simpler, but usually makes some assumptions about the placement of the geometry it routes. These assumptions can be minimized by using a fine grid, however this leads to an increased memory overhead and longer routing times. On the other hand, the gridless approach is compact and makes no assumptions about placement of geometry. In order for a

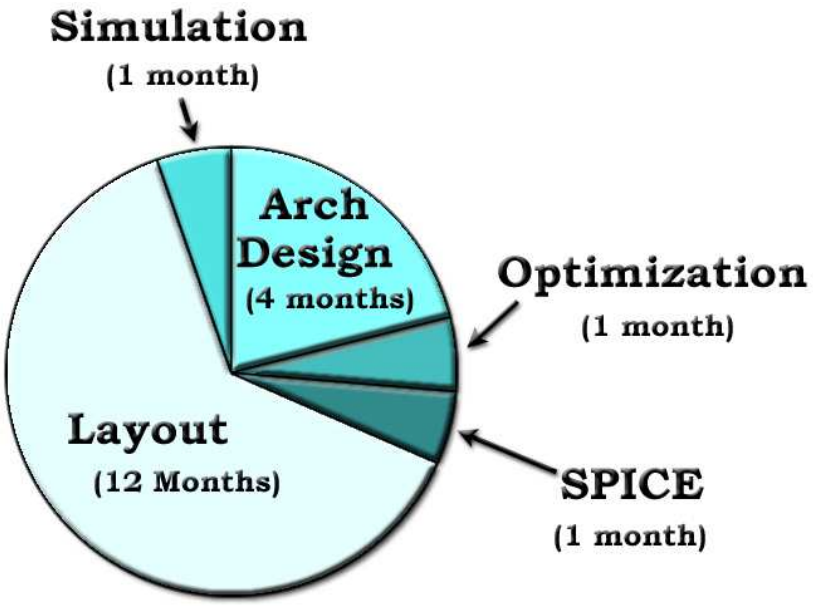

Figure 1.1: The design cycle for the SNAP microprocessor.

router to allow interaction it must be able to operate on any arbitrary, design-rule correct layout. This requirement suggests using a gridless-style router. However, capturing all the intricacies of modern design rules makes using a gridless router challenging. We will present a methodology to meet these challenges.

1.2 Related Work

Gridless routers have been extensively studied in the past. The Magic VLSI editor [12] has an interactive router based on its corner-stitched tile structure [11, 2]. Hamachi improved on the Magic router by adding the notion of preferred directions for routing and avoiding hazard areas [7]. Lunow later presented a Magic router that assigns a preferred direction per metal layer and uses protection frames to aid in avoiding obstacles [9]. Arnold *et al*[1] developed the IRoute router which uses contours (bloated tiles) to prevent design-rule errors.

The main influence for this work is the CONTOUR router developed by Dion *et al*[4]. CONTOUR incorporates many of the ideas of the previous works and provides a novel path searching mechanism for finding possible routes. CONTOUR uses space tiles in the layout to store possible paths between the two nodes being routed. Non-minimal cost routes are pruned off based on a cost model that assigns costs for horizontal and vertical movements. An exhaustive search is time consuming, so the search may be terminated when a route is found that falls within a user-defined threshold of the minimal cost route (an approximation based upon the distance between the two nodes).

1.3 Core Contributions

Although CONTOUR is quite robust it has some limitations. First, design rule correctness within the current route is only guaranteed when choosing a minimum cost path. Non-minimum cost paths, possibly containing design rule violations, are permitted since an exhaustive search is very time consuming. Second, there is no facility to account for the difference between: i) wire-to-wire spacing, ii) wire-to-contact spacing, and iii) contact-to-contact spacing. Third, routing wires and contacts with different widths will result in spacing violations, known as notches, within the route itself (regardless of choosing the least cost path). Fourth, the current cost model doesn't factor in jog or via costs, which have a profound effect on the resulting layout. Fifth, no method for choosing the order in which to route nodes is provided.

Our routing methodology guarantees that routes will always be design rule correct even if non-minimum cost paths are allowed. This is achieved through a clever technique that enforces spacing rules across different parts of the route

found during the path search. In addition, since enforcing spacing rules limits the total number of possible paths, waiting for the least cost path is more feasible. We account for spacing rules related to contacts by explictly modeling oxide layers, which provides a convenient place to store contours related to these rules. Additionally, we provide a method to account for jog and via costs in our improved cost model. We also show how to postprocess geometry so that we can eliminate notches when routing wires and contacts with different widths. Finally, we present a method to route the harder to route nodes first, thus increasing the routability of the circuit.

1.4 Organization of Thesis

In the next chapter, we will discuss the corner stitching data structure, the CONTOUR router, and the limitations of the CONTOUR router. Chapter Three presents the data structures that provide the framework for our router. Chapter Four covers our novel node-to-node routing methodology as well as our method to maximize the routability of a circuit by routing the more difficult nodes first. Chapter Five compares this router with the Cadence Chip Area Router. Chapter Six concludes this thesis.

Chapter 2

Preliminaries

The final phase of integrated circuit design involves creating a schematic with the exact physical placement of transistors and their connecting wires. IC Manufacturers extract masks from this schematic that are used to transfer base materials onto semiconductor wafers. There are many different types of fabrication technologies and each has its own set of rules for the placement of materials in the schematic. These rules, known as design rules, ensure that devices created during the fabrication process function correctly. This schematic of circuit geometry is referred to as the *layout* of a circuit.

Circuit designers employ the use of layout editors to create layout and check that it is absent of design rule violations. Creating layout is composed of three main steps: i) create transistor stacks, ii) place transistor stacks, and iii) create connections (wires) between stacks. Out of these three steps, creating connections between stacks is the most complex and time consuming. While the materials used in transistors stacks reside on one layer (in most models), there are upwards of six metal layers used for wires in modern fabrication technologies. The process of connecting two nodes entails running wires, through one or metals layers, between them. Contacts (vias) are metal squares used to connect wires on different metal layers. The process of creating wires between nodes is known as *routing*.

To aid in the discussion of our work, we provide a review of the corner-stitching data structure[11] and the CONTOUR router [4].

2.1 Corner-Stitched Data Structure

Corner-stitching is an efficient data structure for representing layout geometry and is the backbone of the Magic VLSI design tool[12]. Planes of corner-stitched tiles are used to store geometry for the various materials of an IC. The two major advantages of this data structure are: i) its compact representation, and ii) its ability to find neighboring tiles quickly.

2.1.1 The Corner-Stitched Tile

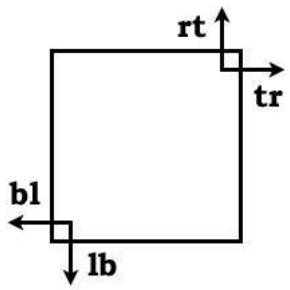

Figure 2.1: A corner stitched tile.

The corner-stitched tile, shown in Figure 2.1, contains four corner pointers to neighboring tiles, the x,y coordinates of its bottom-left corner, and a material type. The upper bounds of the tile can be obtained by requesting the lower bounds of the north and east neighbors. The power of this structure lies in its ability to find its neighbors efficiently. For example, we can find the neighbors bordering the left side of the tile by following the *tr* pointer, then following the subsequent *lb* pointers.

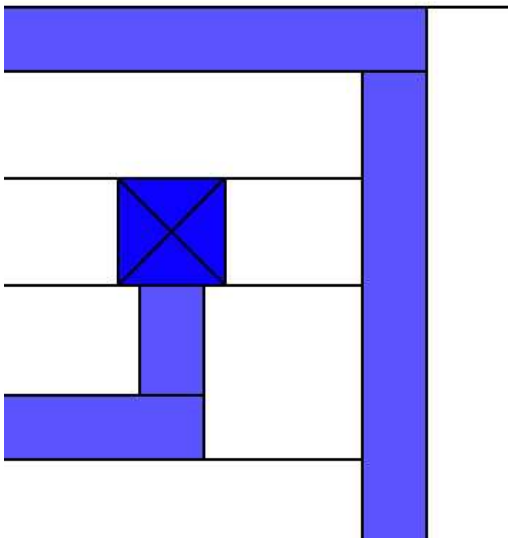

Figure 2.2: An example of a horizontal tile plane.

2.1.2 Tile Planes

A minimal tile structure is enforced by using the minimal number of horizontal tiles to represent the geometry, which is shown in Figure 2.2. Alternatively, the minimal number of vertical tiles can be used, shown in Figure 2.3. The aforementioned tile plane figures each contain eight space tiles, four metal tiles and one contact tile. Tiles are added and removed thorough a series of splits and merges that update its corner points and those of its neighbors. The last recently used tile is stored for each layer and is used as a starting point for various algorithms. This tile is referred to as the layer's hint tile.

Contact tiles are an abstraction that represents the metal materials that it connects and the via between these metals. The size of the contacts is a factor of the via size, spacing, and metal border. Usually, contact tiles reside on the bottom of the two layers that it connects. Magic uses similar abstractions to save the designer from explicitly drawing wells and selects.

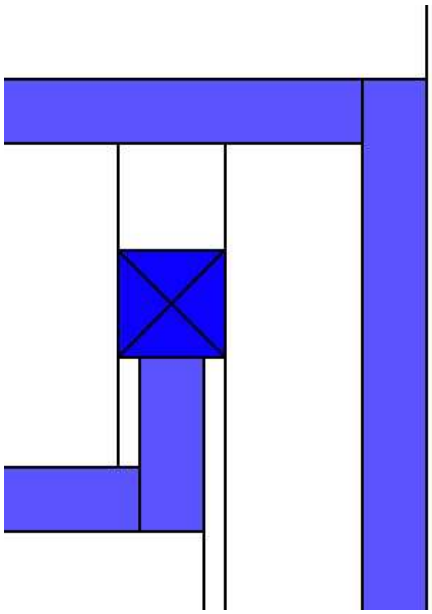

Figure 2.3: An example of a vertical tile plane.

2.2 The CONTOUR Router

CONTOUR uses a centerline routing algorithm based on a method that is a hybrid of maze routing[10] and line searching[8]. The solution for the centerline of the route is found and bloated to meet width requirements. The CONTOUR routing algorithm is composed of five basic steps: i) preprocess geometry, ii) find initial paths, iii) propagate paths, iv) postprocess geometry, and v) draw resulting route. We will omit discussion on postprocessing geometry and route drawing for the sake of brevity.

2.2.1 Preprocessing Geometry

CONTOUR preprocesses geometry to create contour tiles that surround existing material tiles. The purpose of these contour tiles is to designate areas of the geometry that cannot be used for routing. The contour tiles reserve enough area

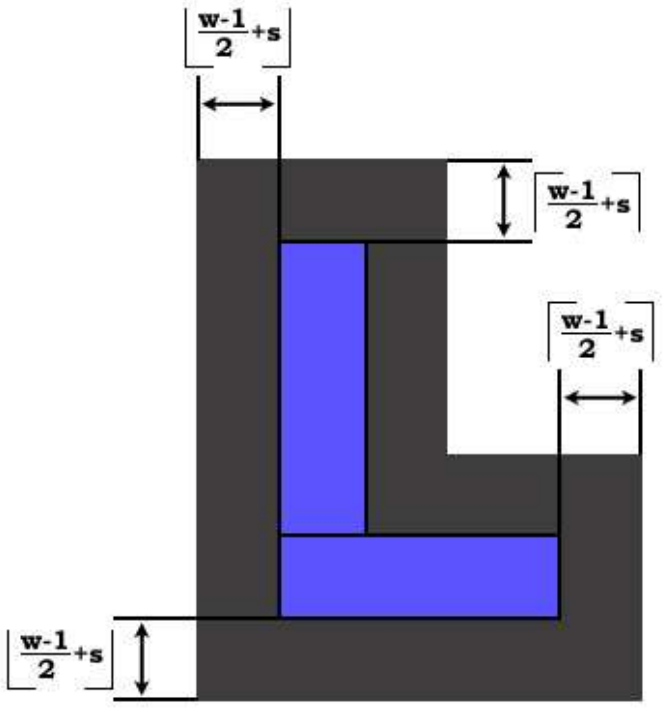

Figure 2.4: Adding contours to material tiles.

so that spacing rules will not be violated after the centerline path is bloated to meet width requirements. The centerline is bloated by $\lfloor \frac{width-1}{2} \rfloor$ on its north and east sides and by $\lceil \frac{width-1}{2} \rceil$ on its south and west sides, where $width$ is the width of the metal being routed. In order to reserve enough space, the material tiles need contours that exceed their north and east walls by $\lceil \frac{width-1}{2} + spacing \rceil$ and their south and west walls by $\lfloor \frac{width-1}{2} + spacing \rfloor$, as shown in Figure 2.4.

2.2.2 Finding Initial Paths

Before the search for routes can begin, the initial paths of each terminal must be located. These paths can either be a wire connecting to the terminal or a contact connecting to the terminal. Initial wire paths are found by following method: i)

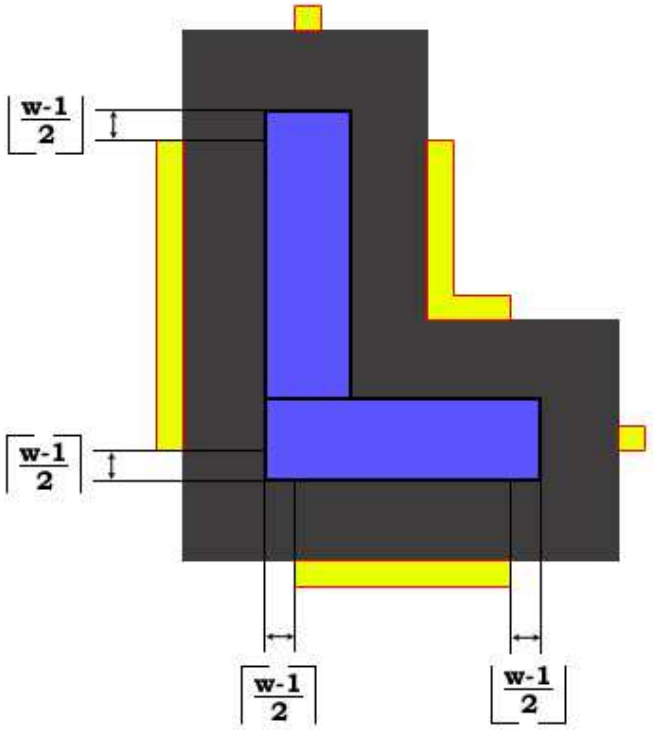

Figure 2.5: An example of initial wire paths.

copy terminal geometry to a temporary tile plane and shrink it by $\frac{width}{2}$, ii) explore the area within $width + spacing$ in the original plane, and iii) if this area is a space tile and overlaps the shrunken geometry in one dimension it is an initial wire path. An example of initial wire paths is shown in Figure 2.5. We will avoid a discussion of initial contact paths and just note that CONTOUR requires contacts to be smaller than or equal in size to the terminal geometry.

2.2.3 Propagating Paths

To implement the CONTOUR routing algorithm some extensions to the corner-stitching data structures are needed, as well as some additional data structures. First, a new material type representing a contour is needed. Tiles must be extended

to contain the head of two different linked lists. These two lists are: i) the paths leading from one terminal to each space tile, and ii) the paths leading from the other terminal to each space tile. Two priority queues, one for each terminal, are needed to keep track of the propagation paths. Paths in these queues are sorted so that paths that have the potential to form the least-cost route have higher priorities.

Finally, we have the key to this algorithm, the path data structure. The path data structure is composed of the following six items: i) a tile pointer, ii) minimal cost rectangle, iii) source cost, iv) destination cost, v) back pointer, and vi) next pointer. The tile pointer is merely a pointer to the tile in which the path resides. The minimal-cost rectangle is essentially the bounds of the path. This rectangle is composed of a group of unit squares that share the same cost and often it will be a single unit square. The source cost is the cost to get to the path's rectangle from its starting terminal based on horizontal and vertical movement costs. The destination cost is the Manhatten distance from the path's rectangle to the opposite terminal. The back pointer is a pointer to the ancestor path that spawned the current path. The next pointer is a pointer to the next path in the tile's linked list of paths.

Now that the data structures are explained, we can detail the path propagation algorithm. After initial paths are found, they are inserted into the tiles' linked lists of paths and also inserted into the priority queue cooresponding to their terminal. The path with least possible cost route, simply the sum of source and destination costs, is removed from each queue and propagated. Propagation entails finding all neighboring space tiles, creating least costs paths to these tiles, and inserting these new paths into the priority queues. A possible solution is formed when the linked list of paths leading to a space tile from the other terminal is non-empty.

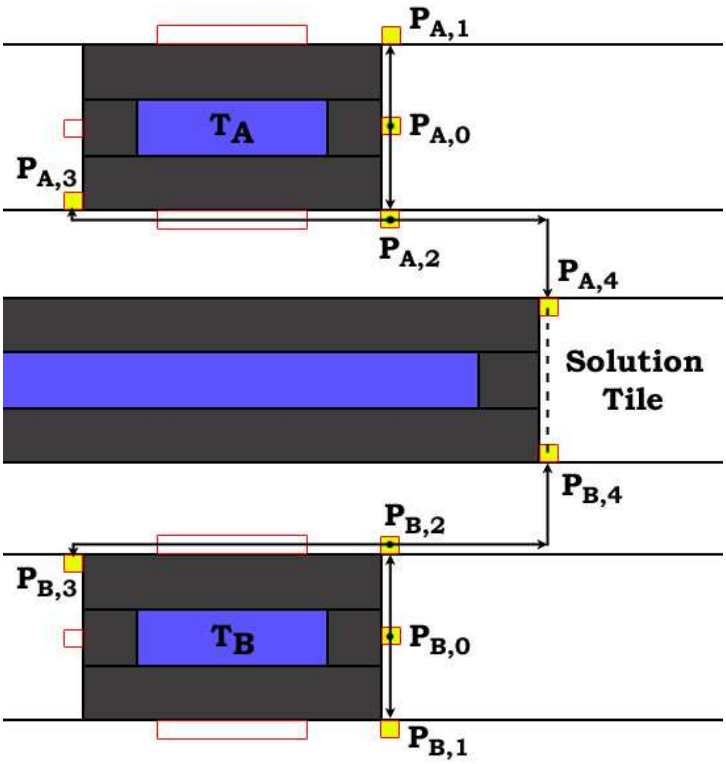

Figure 2.6: Path propagation between two nodes.

The least-cost solution is recorded and path propagation is terminate when either: i) the priority queues are empty, ii) the current best solution is cheaper than the least-cost routes of the head of each queue, or iii) the current best solution is within a user-defined threshold. An example of path propagation is shown in Figure 2.6.

2.3 Limitations of CONTOUR

The basic route-finding algorithms of CONTOUR are sound. However in current technologies the results are poor. These poor quality results are outlined in this section.

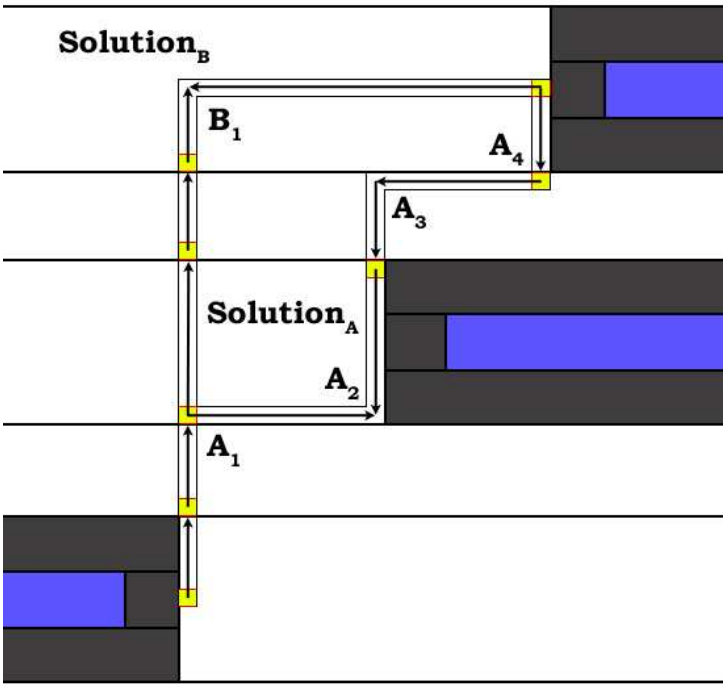

Figure 2.7: Two possible paths with the same cost. Path A has four turns, but Path B has only one.

2.3.1 Excessive Jogs

The cost model used in CONTOUR only includes a cost for horizontal and vertical movements. This will often results in excessive jogs since paths with the same cost can have a different number of jogs. Figure 2.7 shows an example of two paths with the same cost, however one path has four turns and the other has only one. In this example, the path with the larger number of turns would be found first and chosen as the solution.

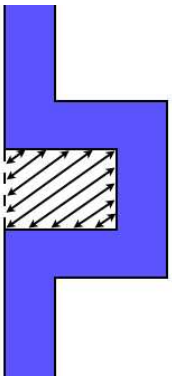

Figure 2.8: A wire spacing violation.

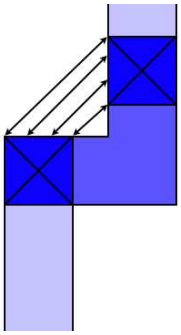

Figure 2.9: A contact spacing violation.

2.3.2 Design Rule Violations

Routes created by CONTOUR may have spacing violations with other parts of the same route. Wire spacing violations are possible when the path search is terminated early. There is no mechanism to prevent CONTOUR from choosing paths that create in U-shaped structures similar to the one in Figure 2.8. Contact spacing violations are always possible and are often a minimum cost path depending on the horizontal and vertical costs of the two layers involved. An example of a contact spacing violation is depicted in Figure 2.9.

2.3.3 Notches

Using CONTOUR to route wires and contacts with different widths will result in notches. Notches are spacing violations within a wire that can be corrected by filling them with metal. There are two types of notches: i) contact-to-contact as in Figure 2.10, and ii) contact-to-elbow as in Figure 2.11.

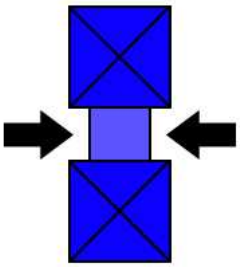

Figure 2.10: A contact-to-contact notch.

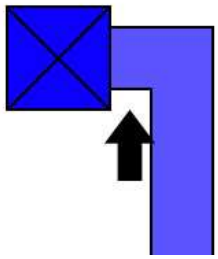

Figure 2.11: A contact-to-elbow notch.

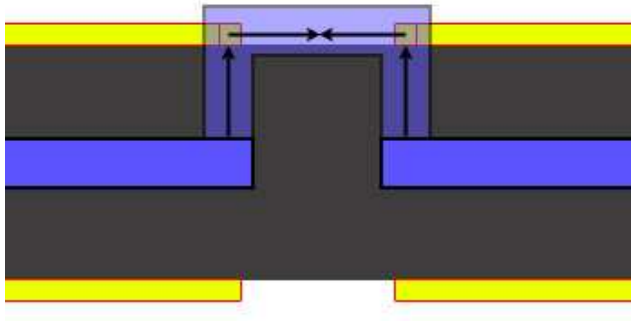

Figure 2.12: An example of a loop between closely placed nodes.

2.3.4 Loops

A side-effect of creating contours around terminal tiles are loops. Nodes of a net may be placed closely together prior to routing, as in Figure 2.12. If the nodes are $width + 2 * spacing$ distance or less apart their contours will merge into a larger contour. This larger contour will prevent the router from creating the direct connection between the two nodes. In the best case, the result is a wire loop which is obviously suboptimal. However, this limitation frequently causes routing between such two nodes to fail completely. This occurs because a good transistor stack placement will often result in transistors of the same type with the same inputs at minimum distances from one another. Since transistors in a stack

are placed as closely together as possible to minimize leakage current, the direct connection between such nodes is often the only possible connection.

Chapter 3

The Router Framework

This chapter outlines the underlying framework of the router. Some changes to the original data structures are needed to resolve the limitations of CONTOUR. In addition, the router must be configurable so that it can easily be applied to future technologies.

3.1 Data Structures

The data structures used in the original CONTOUR router need to be modified to resolve some of its limitations. Changes are made primarily to the corner-stitched tile, the tile planes, and the path structure.

3.1.1 3D Tiles

The corner-stitched tile data structure is extended by adding the following: i) a pointer to the tile above the northwest corner, in the tile plane above, ii) a pointer to the tile below the northwest corner, in the tile plane below, iii) an integer tag representing the route that the tile belongs to, and iv) a byte of status bits.

The two new pointers allow the tile to access its neighbors in the plane above and below quickly. In CONTOUR, the hint tiles in the adjacent planes are used for this purpose. The hint tile is used as an access point into the adjacent tile plane and then the plane is traversed until neighboring tiles are found. The performance of this algorithm greatly depends on the placement of the hint tile. Since the hint tile is the last recently used tile of a plane, its placement is usually optimal. However, since paths from each terminal are processed sequentially during path

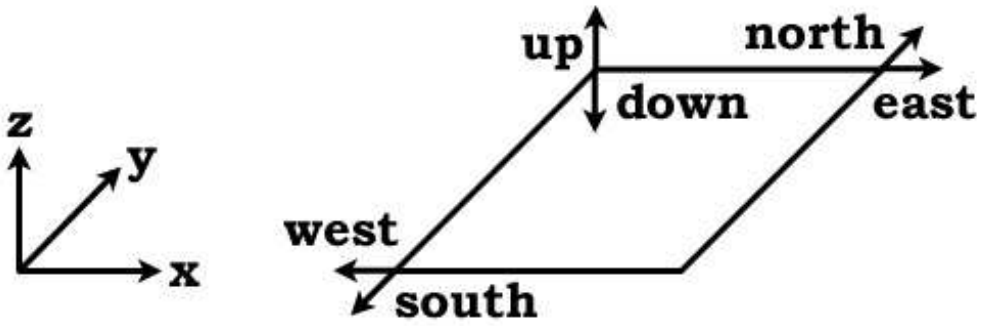

Figure 3.1: The 3D corner-stitched tile.

propagation the hint tile will often be poorly placed if the terminals are a significant distance apart. The addition of the new tile pointers allow for more direct access to neighboring tiles in adjacent planes without relying on the placement of the hint tiles.

The route tag is an integral part of the overall route management. Routing a circuit often entails undoing, ripping-up, previous routes and trying them again in a different order. Normally the entire electrically connected node needs to be removed since there isn't any way to distinguish between the different routes in the node. This can be a rather large step back if the node contains many individual connections. By tagging the tiles with the ID of their route we have a way to remove a single route without having to remove the whole node.

The tile's status bits provide an efficient way to mark the tiles during different types of processing. The status bits contain five generic tags and three specific tags: i) start terminal tag, ii) end terminal tag, and iii) connected tag. The start and end terminal tags are necessary for the new geometry preprocessing stage described in the next chapter. The connected tag is used to signify that a tile is already part of the set connected tiles (useful for finding a unique set).

3.1.2 Explicit Oxide Layers

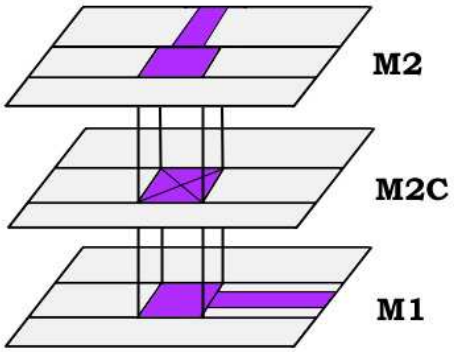

Figure 3.2: An example of explicitly representing oxide layers as tile planes.

The tile planes traditionally represent the active and metal layers in a given technology. In addition, we will explicitly represent the oxide layers as tile planes as shown in Figure 3.2. Contacts can now be stored on oxide layers rather than on one of its composite metal layers. More importantly, this allows us to use the oxide layer to make contours that enforce contact spacing rules. Also, if metal tiles and contact tiles are stored on the same layer it would be difficult to manage the route tags. For example, a contact tile of a new route may overwrite a metal tile of a previous route and erase the metal tile's route tag.

3.1.3 Extensions to Path Structure

The path structure is extended in two ways: i) a path now keeps track of its current heading, and ii) a path is now a node in two doubly-linked lists. A path's heading is simply its neighbor relation to the previous path. For instance, if a path propagates to a east-neighboring space tile then this new path has an east heading. The possible headings are north, east, south, west, up, and down. The up and down headings specify movement between tile planes. Headings will play a key role in

calculating jog and via costs, implementing spacing rules during propagation, and route drawing.

Paths are now nodes in two doubly-linked lists: i) the tile's linked list of paths, and ii) the propagation priority queue of one of the two terminals (now implemented as a linked list). The doubly-linked lists allow redundant paths, paths that have cheaper alternative paths, to be deleted during propagation. The child paths of these redundant paths are recursively deleted since they will have cheaper solutions and since their back pointers are invalid.

3.2 New Tile Types

New tile types are needed to allow for more complex behaviors such as applying different propagation rules or altering the cost model. These new tile types are described in this section.

3.2.1 Terminal Tiles

Terminal tiles can be one of two types: i) horizontal, and ii) vertical. Terminal tiles restrict path propagation so that propagation may only occur in the axis cooresponding to terminal tile's nomen. An example of path propagation in a horizontal terminal tile is shown in Figure 3.3. Terminal tiles are used to enforce spacing rules with respect to each terminal of a particular route. This will be explained further in the following chapter.

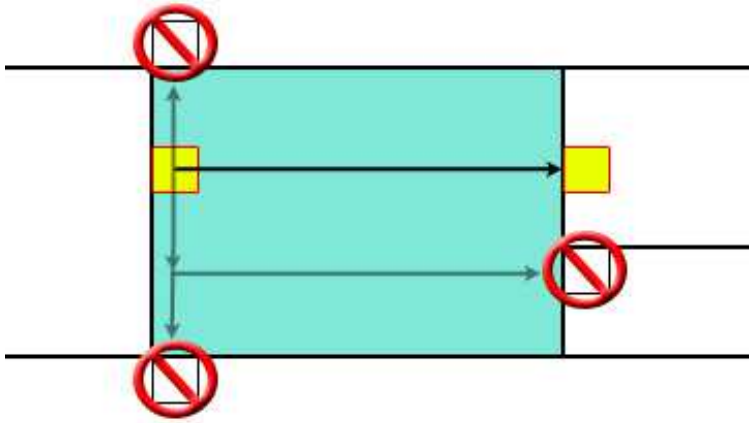

Figure 3.3: Path propagation through a horizontal terminal tile.

3.2.2 Boundaries

Soft boundary tiles and hard boundary tiles are useful in creating artificial bounds in layout. Soft boundary tiles contain a cost multiplier that modify the cost when propagating through them. Hard boundary tiles disallow path propagation completely. The combination of these two tiles provides a straight forward way for the router to avoid certain areas of the design or minimize the use of certain area of the design. These boundaries are specified via the directives discussed in Appendix B.

3.3 Framework Configuration

Layout geometries are read in from Magic[12] design files. The layers and materials are configured through the router's technology file, described in Appendix A.

3.3.1 Layers

The router generates metal and oxide layers depending on the number of metal layers specified in the technology file. This is shown in Table 3.1. The first two

layers are the active and active oxide layer. The following layers are metal layers and their oxide layers. The only exception is the top metal layer which doesn't have an oxide layer since it doesn't require contacts.

Table 3.1: Framework layers.

Layer	**Default Material**
Active	Polysilicon
Active Oxide	Poly Contact
Metal_i	Metal_i
Metal_i Oxide	Metal_{i+1} Contact

3.3.2 Materials

Active and active oxide layers required more than one material type. The active layer needs both diffusion and polysilicon types. The active oxide requires diffusion contacts, substrate contacts, and the polysilicon contacts. Metal layers and metal oxide layers need only a single material, metal and contact respectively. These materials are listed in Table 3.2.

Table 3.2: Framework materials.

Material	**Layer**
P-Diffusion	Active
N-Diffusion	Active
Polysilicon	Active
P-Diffusion Contact	Active Oxide
N-Diffusion Contact	Active Oxide
P-Substrate Contact	Active Oxide
N-Substrate Contact	Active Oxide
Polysilicon Contact	Active Oxide
Metal_i	Metal_i
Metal_{i+1} Contact	Metal_i Oxide

Chapter 4

Proposed Routing Methodology

This chapter presents our routing methodology. There are two main components to successfully routing a circuit: i) efficient and robust node-to-node routing, and ii) smart ordering of the node pairs to be routed.

4.1 Node-to-Node Routing

In this section we present a methodology that overcomes the shortcomings mentioned in the previous section. Specifically, we extend the CONTOUR path-finding algorithm to incorporate a robust cost model and to always generate design-rule correct routes. In addition, we provide some techniques to help make waiting for the absolute least-cost route more feasible.

The major stages of the routing algorithm are shown in Figure 4.1. The algorithm begins by first choosing a pair of nodes to route. The geometry is then preprocessed to ensure that contours are drawn around the nodes that aren't being routed. Initial paths for each node of the route are located and inserted into the path heaps. Paths are propagated from each priority queue until a minimum cost path is found or both priority queues are empty. If a valid path is found, it is extracted from the layout. The geometry is returned to it's initial state and the route is drawn if one was found.

4.1.1 Preprocessing Geometry

All of the initial layout geometry is contoured when it is first read from a file. These contours are created in a similar fashion as the original CONTOUR algorithm.

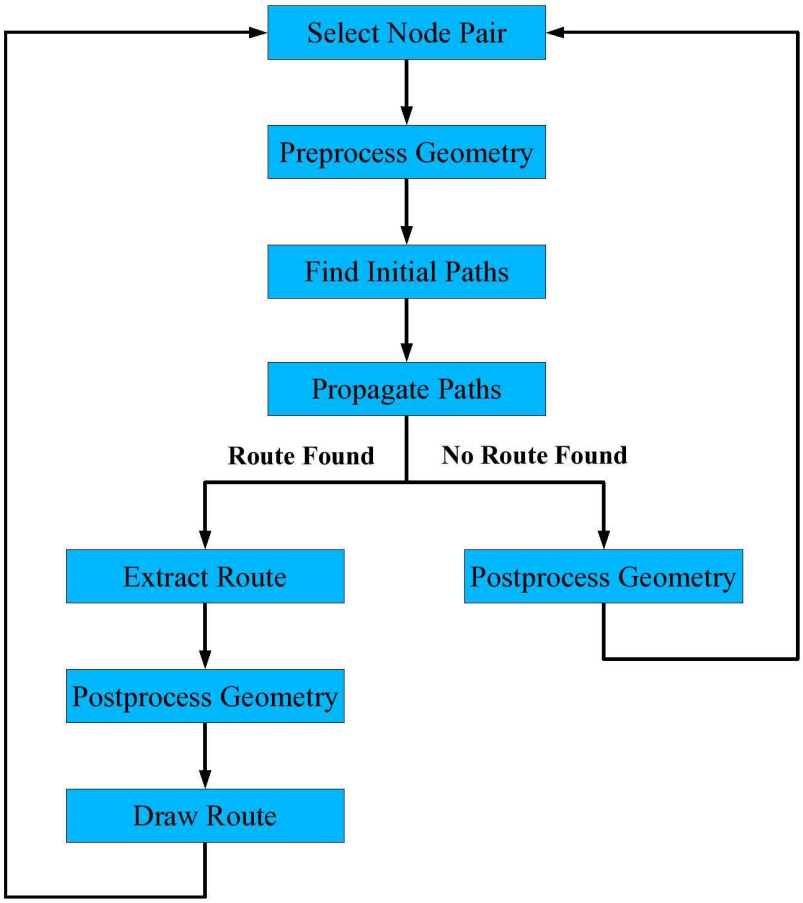

Figure 4.1: The major stages of the routing algorithm.

However, in our method we account for contact-to-contact spacing on the oxide layers, wire-to-wire spacing on the metal layers, and wire-to-contact spacing on both layers. Therefore, each material tile will receive three contours each: i) a contour on the current layer, ii) a contour on the layer above, and iii) a contour on the layer below. These contours are all $\frac{width-1}{2} + spacing$ units large, however the *width* and *spacing* for each material on each layer may be different.

If there are contours around the two nodes being routed there will be resulting

wire loops as outlined in Section 2.3.4. Therefore, the first step in preprocessing the geometry is to remove contours from the pair of nodes. This can be achieved in a two stage process. First, delete all contours within $\frac{width-1}{2} + spacing$ units of each node's tiles. Note that this will likely also delete contours that belong to some of the nodes not being routed. Second, recreate contours for nodes that lie within $\frac{width-1}{2} + spacing + 1$ of the routing nodes. This will restore contours which have been wrongfully deleted from the nodes that aren't being routed.

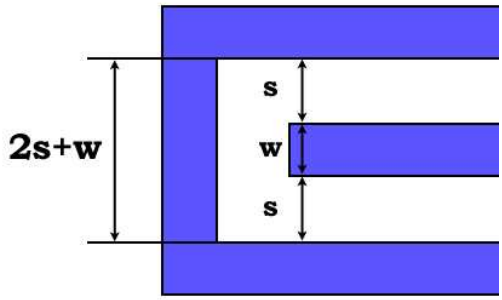

Figure 4.2: The minimum spacing of a wire to itself is $2 * spacing + width$ units.

As justification for the next step of the preprocessing stage, we will examine Figure 4.2. An important observation we can make about a wire is that tiles around a bend must be at least $2 * spacing + width$ units apart from one another. Anything less than a distance of $2 * spacing + width$ units would imply that the wire contains an unnecessary bend, since nothing else could fit between the two segments of the wire. We can enforce this spacing using horizontal and vertical terminal tiles, first mentioned in Section 3.2.1.

Horizontal terminal tiles are found by bloating metal tiles horizontally by $2 * spacing + width + \frac{width-1}{2}$ (the distance to the centerline of the next valid metal) and copying them to a temporary tile plane. Now we shrink these tiles vertically

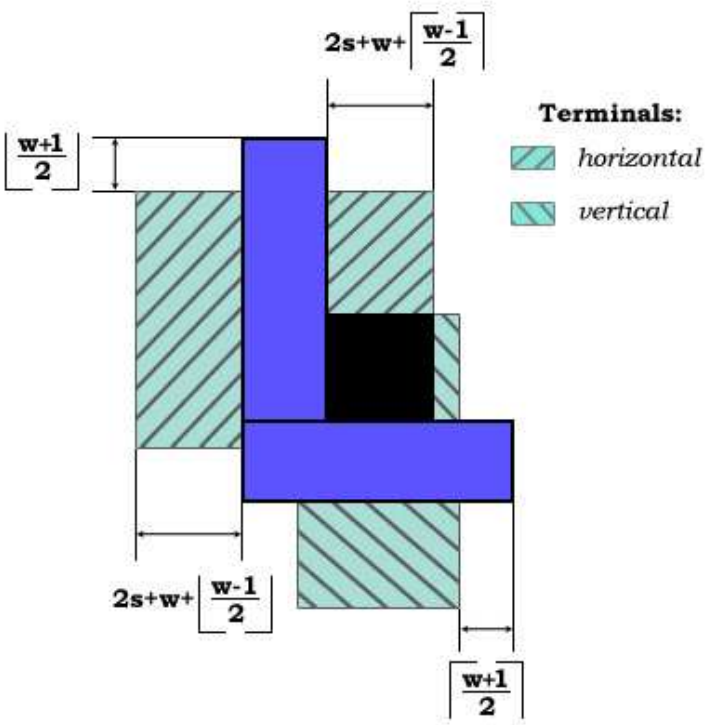

Figure 4.3: Geometry of a terminal after preprocessing.

by $\frac{width+1}{2}$, which allows turns to occur at the edges where they are legal. Finally the temporary tiles are copied back to the original layout as horizontal terminal tiles. Vertical terminal tiles are formed in a symmetric fashion, with the additional caveat that regions where both types of terminal tiles overlap become contours. An example is shown in Figure 4.3.

4.1.2 Finding Initial Paths

Intra-planar candidate initial paths can be found by locating edges on metal planes with metal on one side and space on the other. These paths start at the edge and extend one unit into the space tile. Their direction is simply the direction of the space tile from the edge. These paths must be be shrunk by $\frac{width-1}{2}$ in the

plane perpendicular to its heading. This will ensure that later bloating of these paths will result in lines that do not extend past the initial geometry. Finding inter-planar candidate initial paths requires using a temporary tile plane. This procedure entails copy metal tiles from one layer to a temporary tile plane, and shrink (or bloat if negative) these tiles by $width - (1 + bloat)$, where $bloat$ is the amount to bloat the centerline to form a contact, and $width$ is the width of the contact. This will ensure that a contact overlaps the metal by at least a $width$ amount. The entire process needs to be repeated for both contacts heading upward and contacts heading downward.

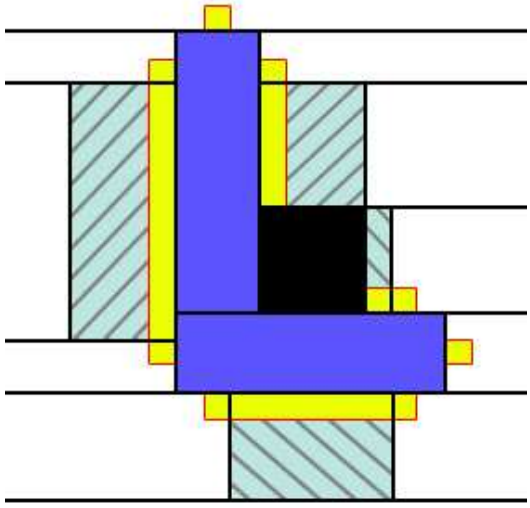

Figure 4.4: Finding initial paths.

We now apply the candidate initial paths and only keep them if: i) they overlap space tiles, or ii) they overlap terminal tiles and are heading in the same direction as the terminal tile. The result is shown in Figure 4.4. Note, up and down initial paths are not stored on the oxide layer, but are moved directly to the metal tile planes. This choice was made to restrict paths from ever being on the oxide layers,

whether initially or through propagation, with the exception of special case paths that may connect the two terminals of a route through a single contact.

4.1.3 Path Propagation

Path propagation upward and downward now has to cross check space tiles in the adjacent oxides planes with space tiles in metal planes two layers away. This is a simple extension to the existing algorithm and is illustrated in the following example of upward path propagation. First, find all neighboring space tiles in the oxide layer above the current tile (we may assume that we are in a metal plane since paths are prohibited from the oxide layers). For each neighboring space tile in oxide layer, we find the neighboring space tiles above, in the next metal layer, and compose a unique list of space tiles in this metal layer (a space tile in the metal plane may be an upper neighbor to more than one tile in the oxide plane). We propagate only a single path to each of the space tiles in the metal plane by choosing the shortest propagation through the oxide space tiles to the metal space tiles. Choosing a single path to each unique metal space tile prevents a blow-up of unnecessary paths.

Path propagation also needs to ensure that new paths won't create spacing violations with other parts of the route if it is used in the solution. To enforce these spacing rules we need to determine what line segments would be added to the centerline by the new path. There may be up to two line segments added to the centerline depending on if a jog is needed or not. Figure 4.5 depicts an example of a path that requires two line segments.

The new centerline segments are bloated in all directions except the direction from which it comes from. These segments are bloated by $2*width-1+2*spacing$

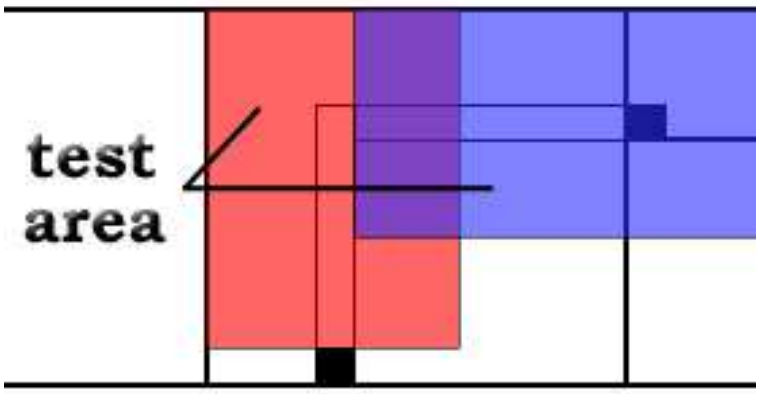

Figure 4.5: Centerline segments created by a new path are bloated to check for spacing violations with centerline segments created by previous paths.

using the same reasoning as in the terminal tile sizing except this time we're checking the spacing between two centerlines(which haven't been bloated to their full width). After bloating the centerlines, we follow the path's back pointer and check these bloated centerlines against unbloated centerlines formed by ancestor paths. If any overlap occurs then the new path is illegal. Note that even though the two bloated centerlines of this example overlap, the design is still fine. A similar, yet simpler, process is used to determine if a path that creates a contact is legal. When a possible solution is formed, each centerline of both path sets must be check against one another.

In the original CONTOUR, path propagation terminates early only if the cost of the current best route is within some user-defined threshold of the minimum possible cost route (the cost of the route if there aren't any obstacles). Unfortunately, the least-cost path may be a few times greater than the estimated minimum-cost route (especially with the new cost model introduced in the next section). Instead, we will use the length of the lull between solutions as a termination point. Typi-

cally, when a solution is found a series of slightly different, but cheaper, solutions will replace it. By waiting for this activity to wind down we may get a solution that is minimal, at least locally, to other recent solutions.

4.1.4 Cost Model

Our cost model calculates source costs and destination costs differently from the previous approach. The source cost now includes via and jog costs. Via costs are relatively straight forward to implement, however, jog costs can be a bit tricky. A single path propagation may have up to two jogs as shown in Figure 4.6. If the current and next path are aligned, then a single jog occurs as long as the current path's heading is different than the next path's heading, as shown in the bottom image of Figure 4.6. If the current and next path are unaligned, as in the top image of Figure 4.6, then one jog occurs if the two paths' headings are different, and two jogs occur if they are the same. This algorithm needs to be modified slightly when considering the jogs in the solution tile.

In the original CONTOUR algorithm, the destination cost is simply the Manhattan distance to the bounding box of the opposite terminal. This measure may be grossly inaccurate if the terminal is many layers away, or if the terminal forms a large L-shaped region. Instead, we make more accurate approximations of the destination cost by finding the minimum cost from the current path to each of the initial paths of the opposite terminal. This yields a much more accurate estimate of destination cost, and it rules out blocked portions of the opposite terminal since they don't have initial paths. Overall, this allows for a wiser order in which the paths propagate.

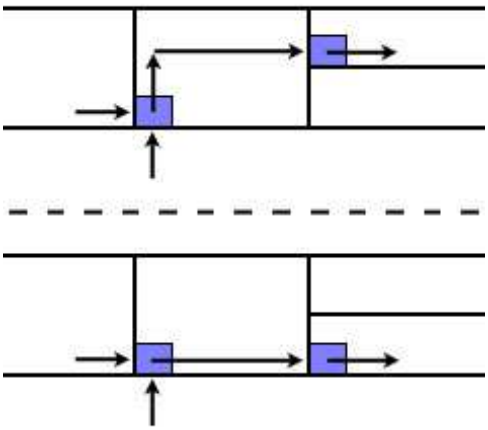

Figure 4.6: Two examples of new paths that may create jogs. The new path in the top image will create 1 or 2 jogs. The new path in the bottom image will create no jogs or 1 jog.

4.1.5 Route Drawing

An unwanted side-effect of routing contacts and wires at different widths are notches. As shown in Figure 2.10 and Figure 2.11, notches may form between two different contacts or a contact and an elbow. The two spacing rules that are violated here is wire-to-wire spacing and wire-to-contact spacing. In our implementation, this is accurately depicted since a contact is decomposed into the wires on the two layers it connects and a via on the oxide layer between them (all the same size as the original contact). The wire-to-wire spacing can be easily fixed by bloating the metal tiles by $\frac{spacing-1}{2}$ and then shrinking them by the same amount. This essentially fills notches as large as $spacing - 1$ (technically anything larger isn't a notch). The wire-to-contact spacing violation requires a bit more processing to remedy.

To illustrate this, consider the case where poly-to-poly spacing is three and poly-to-contact spacing is four. A bloat by $\frac{4-1}{2}$ shorts any two adjacent poly

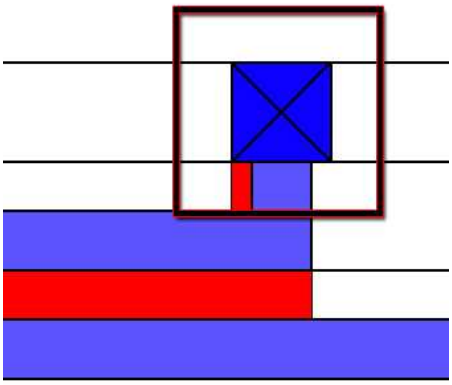

Figure 4.7: An example of metal fill for contact-to-wire notches.

lines with minimum spacing. Instead, we need to perform the following series of operations on a temporary tile plane. First, bloat the metal tiles by $\frac{contact\ spacing-1}{2}$ using a different material (the contour material will work fine), and then shrink by the same amount. Now search an area extending out from the contact bounds by *contact spacing*. Finally, any contour material found in this area can be safely copied into layout as metal.

4.2 Route Management

When no solution for a route can be found, the only course is to rip up previous routes and try again. If the route ordering is poor then long cycles of ripping-up and rerouting may ensue. This may greatly delay finding a solution for a set of routes, or even worse, prevent a solution from ever being found.

As previously mentioned, the tile structure has been augmented to allow a route tag for each tile. This tag allows us to do something unique, rip up partial nets. Usually one must rip up an entire net, since there is no way to distinguish one part of a net from another. This can be a rather large step backwards, especially if the

net is a global signal. It should be noted that pre-existing geometry has a route tag of zero and can never be removed.

4.2.1 Route Ordering

Route ordering is maintained through the use of the following: i) a sorted list of possible routes, ii) an index into the list of possible routes, and iii) a stack of finished routes. All the node-to-node pairs of each net inserted into the list of possible routes are sorted (the sorting algorithm is discussed in the next subsection), as shown in Figure 4.8A. The index is initially set to zero and we try to find a route between the first node pair.

If a route between two nodes is found, the following occurs: i) the route is drawn using the current route tag, ii) the node pair is pushed onto the finished route stack along with the current index and route tag, iii) all node pairs in the possible route list containing the routed nodes are removed, iv) a new node is created and all new node pairs are inserted into the possible route list, v) the index is set to zero and the route tag is incremented, and vi) the possible route list is sorted. An example of this is depicted in Figure 4.8B and 4.8C.

If a node pair fails to route, and both nodes occur further down in the list, then the index increments and the next node pair is attempted. If a node pair fails, and it is the last occurrence of at least one node, then the following occurs: i) pop the last finished route, ii) delete the tiles marked with the route tag of the last finished route, iii) remove the node and node pairs created by the last finished route, iv) reintroduce the old nodes and their node pairs, v) set the index to the index of the last route and increment, vi) decrement the route tag, and vii) sort the list of possible routes.

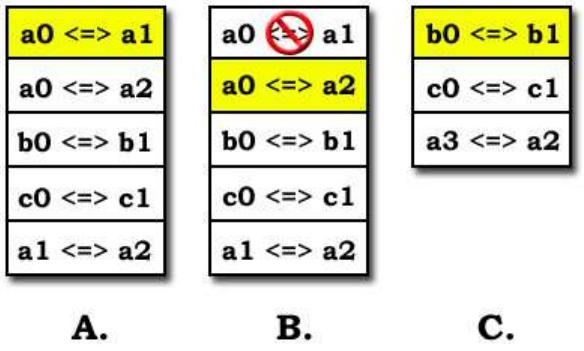

Figure 4.8: Route Ordering. A: All possible routes added to list and sorted. B: $a0 <=> a1$ is unroutable so the next pair, $a0 <=> a2$ is attempted. C: $a0 <=> a2$ routes, nodes $a0$ and $a2$ are removed and replace with $a3$.

The algorithm terminates when either all nodes are routed or the list of finished routes is empty and the last occurrence of a node in a node pair fails to route. This algorithm walks through all possible route orderings in the order set forth by the sorting algorithm, as shown in Figure 4.9.

4.2.2 Sorting Algorithm

Creating a route may completely block off a node and make it unroutable. Due to this fact it is desirable to first route those nodes that are harder to route, since they have a higher probability of being blocked off. The challenge here is to accurately determine the routability of a node. We can make the following observation: node routability is closely related to the combined area of a node's initial paths.

Finding the initial paths of each node after completing a route can be expensive since there may be many nodes and finding initial paths requires removing contours. However, we can exploit two facts to make this reasonably cheap. One, adding or removing a route will only affect the routability of surrounding nodes.

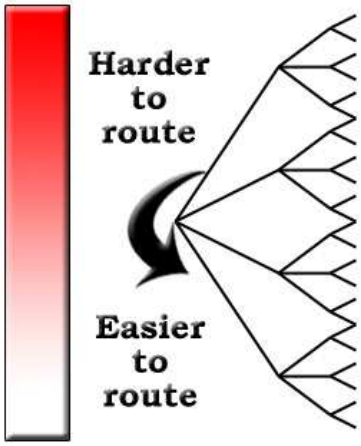

Figure 4.9: Routes are ordered so that harder to route nodes at attempted first.

More precisely, only nodes within $width + 2 * spacing$ units (one more than the size of two contours) of the added/removed route may experience a change in their routability. Two, we can approximate the total area of the initial paths by exploring the region just outside of a node's contours, specifically the area within $\frac{width-1}{2} + spacing + 1$ units of the nodes. This is shown in Figure 4.10.

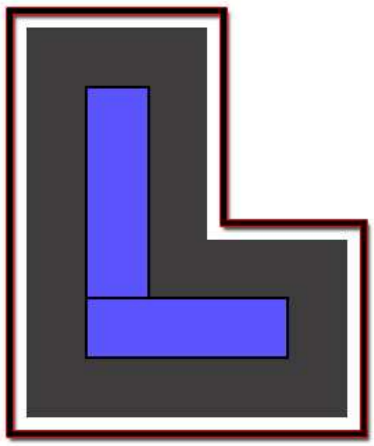

Figure 4.10: Routability is determined by searching the area just outside of a node's contour for space tiles.

Chapter 5

Results

We have implemented this tile-based gridless router in the C++ programming language using just over 10,000 lines of code. A technology file is used to specify the number of layers, the various costs associated with each layer, and the design rules. The router reads and writes geometry using the same format as the Magic VLSI layout editor[12]. Nets can be specified by placing labels on the geometry. This is quite convenient since Magic contains libraries to generate transistor stacks with labeled nodes. Additional layout directives, such as a constraining box for the generated routes or high cost areas, may be specified using labels with the Magic attribute tag.

5.1 Benchmark Circuits

Thirty benchmark circuits were randomly generated for the purpose of comparison. These circuits are composed of labeled squares of metal randomly placed throughout chip area across all metal layers. These bits of metal are minimum size and guarantee at least minimum spacing with respect to the other bits of metal. A good figure of merit for the complexity of these benchmark circuits is *Node Density*:

$$Node\ Density = \frac{Nets * Nodes\ per\ Net}{Metal\ Layers * Area\ per\ Layer}$$

The benchmark circuits have fixed area, metal layers and nodes per net. The area for each circuit is $1,000\lambda$ x $1,000\lambda$. This area is chosen since it is large enough to justify using a router, while small enough to still be consider detail

routing and not global routing. Three metal layers are used for the benchmark circuits, which is a typical number of routing layers for a circuit (higher metal layers are often reserved for global signals). Each circuit has five nodes per net, which is approximately average for non-power signals in a given circuit. The *Node Density* is varied from .011 to .020 by varying the total number of nets, which is shown in Table 5.1. Three of each type of circuit and their results are averaged for comparison. Note that varying the other parameters doesn't significantly change the following results.

Table 5.1: Benchmark circuits.

Node Density	**# of Nets**	**Total Wires**
.011	66	264
.012	72	288
.013	78	312
.014	84	336
.015	90	360
.016	96	384
.017	102	408
.018	108	432
.019	114	456
.020	120	480

5.2 Comparison With Cadence Chip Assembly Router

Cadence is a suite of IC development tools, considered to be the industry standard. It encompasses many tools for design synthesis, including a custom design router

which is the counterpart to the router proposed in this paper. The Cadence Chip Assembly Router V11.2.41 is used for comparison. Specifically, the detail router with "same net checking" turned on (this prevents notches) and 25 pass limit is used (which results in thousands of rip-ups). Our router is configured to stop trying to route after $1,000$ rip-ups. All comparisons were done on a Intel Pentium 4 CPU 3.20GHz with 1GB of RAM.

Both routers are configured to only route on the first three metal layers using same rules as the TSMC (Taiwan Semiconductor Manufacturing Company) 180nm logic process. The following preferred directions for metal layers were used: i) metal-one is horizontal, ii) metal-two is vertical, and iii) metal-three is horizontal. Our router was able to route all but one circuit, while the Cadence router failed to route two circuits. All of the circuits that failed to route were from the set with a node density of .02. These were treated as outliers and not averaged in with their set.

5.2.1 Run Time

The graph in Figure 5.1 shows the average run times for each set of circuits for our router and the Cadence router. The run times for the Cadence router seems to rise exponentially with node density, while run times of our router rise only linearly. As wires become more dense, the Cadence router goes through many rip-up and reroute cycles, while our router performs few rip-ups. Our router is $1.7 - 5$ times faster than the Cadence router for a node density of $.011 - .019$. At a node density of .02 our router is nearly 11 times faster, however, these circuits are particularly dense and hard to route.

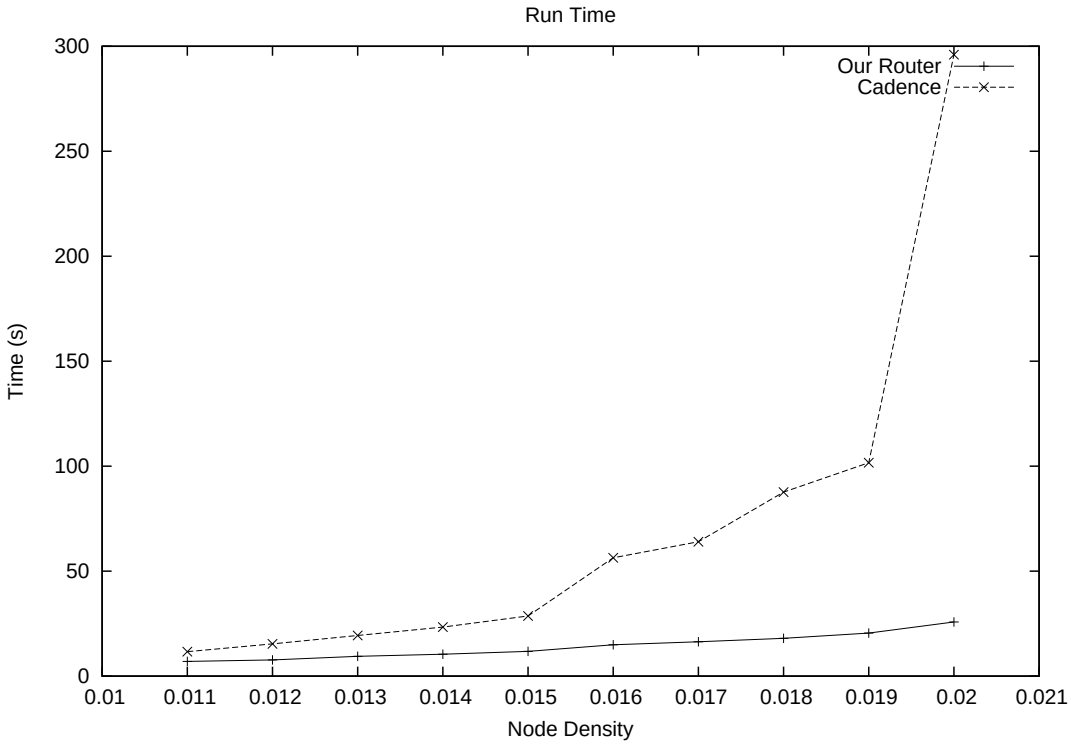

Figure 5.1: Run time for our router versus Cadence.

5.2.2 Memory Usage

The graph of Figure 5.2 depicts the average memory usage for each set for both routers. The Cadence router consumes approximately $6 - 8$ times more memory than our router. This seems to suggest that the Cadence router uses a uniform-grid representation for layout geometries. As theorized by the author, using a uniform-grid representation to apply intricate rule sets incurs a large memory overhead, since a fine grid is required. The gridless representation is clearly superior in minimizing memory usage.

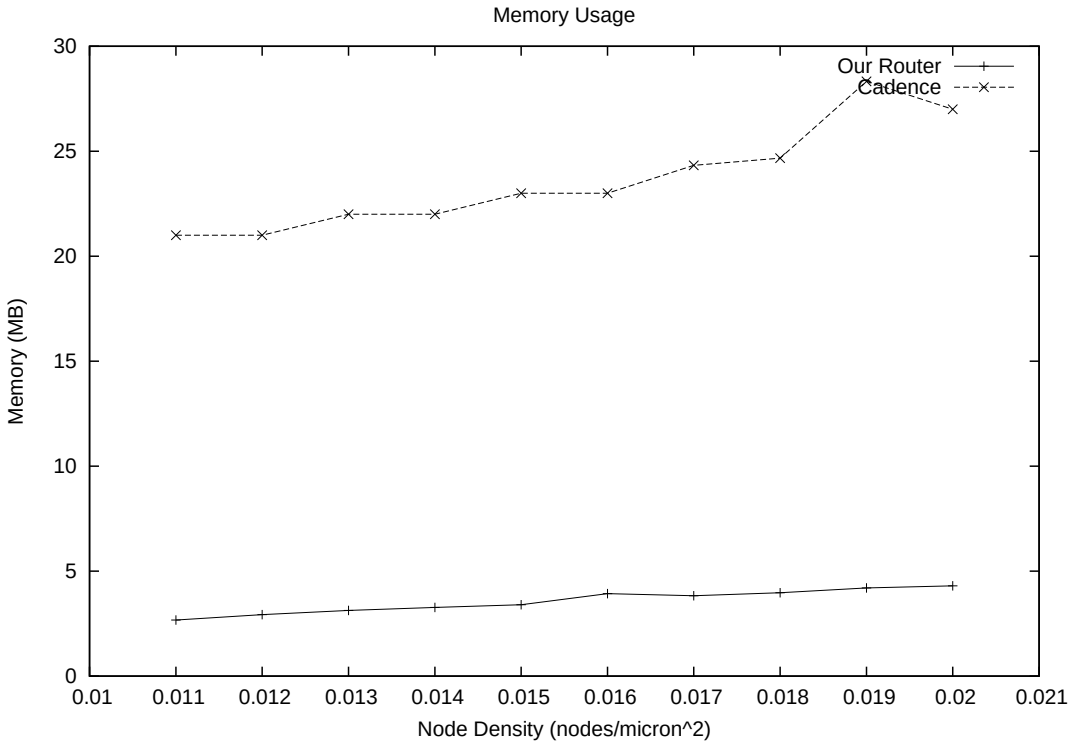

Figure 5.2: Memory usage for our router versus Cadence.

5.2.3 Total Wire Length

Figure 5.3 shows a graph of the total wire length for each set of circuits for both routers. Our router uses roughly $5 - 20\%$ less total wire length than Cadence. As node density increases the percentage of wire length saved by our router decreases. This makes sense since higher node densities reduce the total possible solutions for routing a given circuit. Fewer higher cost solutions will be valid, and thus the remaining valid solutions will have similar costs.

5.3 Quality of Layout

Figure 5.4 shows the metal-one layer of a benchmark circuit with a node density of .018, while Figure 5.5 shows the cooresponding metal-two layer. The wires on the

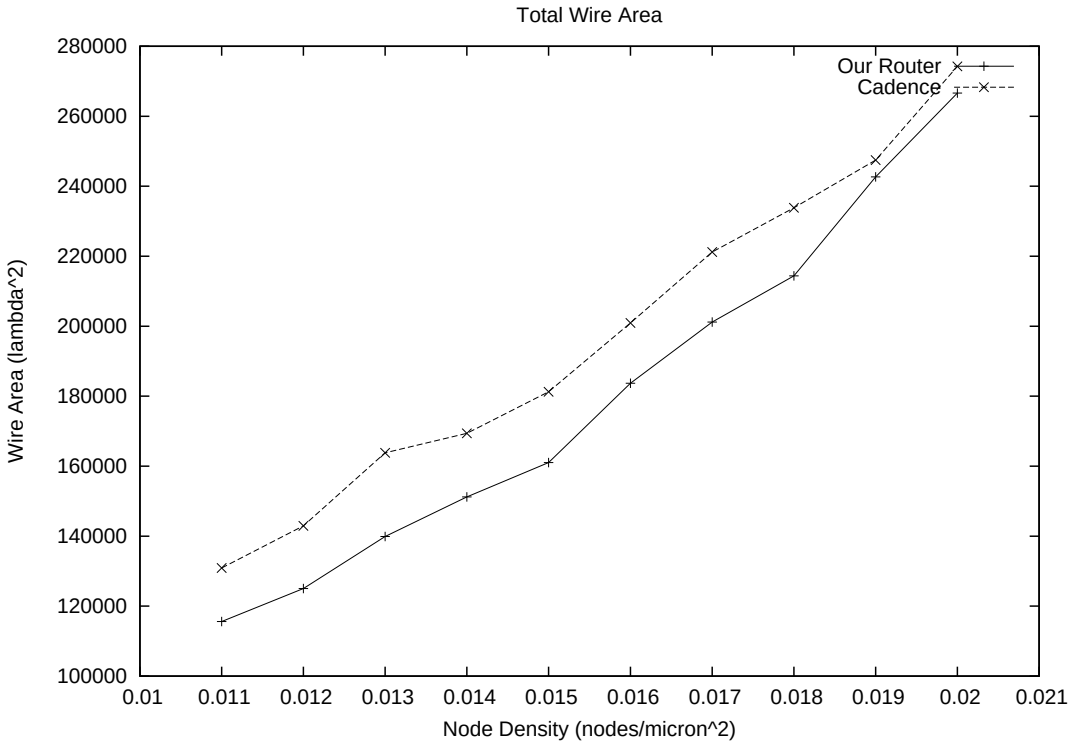

Figure 5.3: Wire overhead for our router versus Cadence.

metal-one layer are mostly horizontal, which was the preferred direction. Similarly, the wires on the metal-two layer are mostly vertical. Vertical wires on metal-one mostly occur in areas where wires on metal-two are closely packed, which shows that unpreferred directions are only used when necessary. Both layers exhibit a minimal number of jogs and are quite dense.

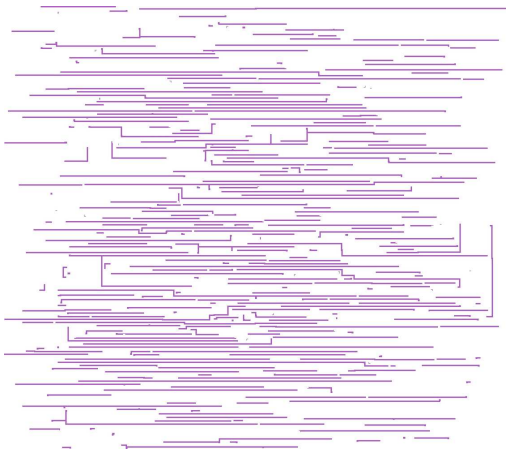

Figure 5.4: Metal-One geometry for a benchmark circuit with node density of .018.

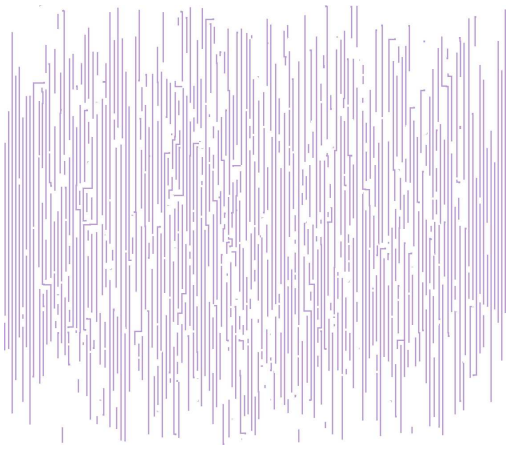

Figure 5.5: Metal-Two geometry for a benchmark circuit with node density of .018.

Chapter 6

Conclusions

6.1 Conclusions

We have presented a tile-based gridless router that produces custom-quality results. Unlike previous gridless routers, this router can exploit modern design rules, operate on arbitrary geometry, and always produce design-rule correct layout. We have shown that early termination of the path finding algorithm will always produce a design-rule correct route. We also described a technique to manage routes so that: i) a single node-to-node connection in a net can be ripped up, and ii) the hardest nodes to route are always routed first. The resulting router can outperform the Cadence Chip Assembly Router. Comparisons with 30 different benchmark circuits show our router to be 1.5-11x faster while consuming 6-8x less memory with 5-15% less wiring overhead.

Appendix A

The Technology File

As discussed in Section 3.3, the router builds layers and materials based upon the settings specified in the technology file. To reduce the complexity of the technology file format, the router assumes that the process has a single active layer and an arbitrary number of metal layers. The materials and contacts associated with these layers are generated as well. The user only needs to define the number of metal layers and the *width* and *spacing* of the materials and contacts. The *spacing* that is defined is the minimum spacing between the current material and default material for its layer.

The technology file is composed of an implicit parameter section followed by sections defined with blocks. The blocked sections are: i) *layers* block, ii) *materials* block, iii)*contacts* block, and iv) *translations* block. Each of the blocked sections begins with the name of the block (all lowercase) and ends with *end*.

A.1 Parameters

There are currently only two required parameters that must appear first in the technology file. They are *tech* and *metals*. The *tech* parameter is just the name of the technology and should match the technology specified in the cooresponding Magic files. The *metals* parameter specifies the number of metal layers in the technology.

tech *technology_name*

metals *number_of_metal_layers*

A.2 Layers Block

The *layers* block assigns a cost to the layer, a preferred direction to the layer, and turns the layer on/off. Costs can either be cheap, normal, or expensive. Preferred layers can be horizontal, vertical, or neither.

layers

layer_name [cheap—normal—expensive] [horizontal—vertical—neither] [on—off]

end

A.3 Materials Block

The *materials* block contains a set of blocks cooresponding to each generated material type. These nested blocks begin with the material's name and end with *end*. Inside each material block one must specify each of the following: i) width, ii)spacing, iii) upspacing, and iv) downspacing. The width is the minimum width of the material. The three types of spacing refer to the material's minimum spacing to the routing material on the current layer, the layer directly above, and the layer directly below. Since metal layers are sandwiched between oxide layers, upspacing and downspacing refer to spacing to the upward contact and downward contact respectively.

materials

material_name

width *minimum width*

spacing *spacing to routing material*

upspacing *spacing to routing material above*

downspacing *spacing to routing material below*

end

end

A.4 Contacts Block

The *contacts* block is similar to the *materials* block except that this block defines materials on the oxide layers. The upspacing and downspacing keywords refer to spacing to the routing metal on adjacent layers.

contacts

material_name

width *minimum width*

spacing *spacing to routing material*

upspacing *spacing to routing material above*

downspacing *spacing to routing material below*

end

end

A.5 Translations Block

Materials that appear in the Magic file may often need to be translated into zero or more materials that the router understands. The *trans* block defines this mapping.

trans

material_name to_material_1 to_material_2 ...

end

BIBLIOGRAPHY

[1] Michael H. Arnold and Walter S. Scott. An interactive maze router with hints. In *DAC '88: Proceedings of the 25th ACM/IEEE conference on Design automation*, pages 672–676. IEEE Computer Society Press, 1988.

[2] Myong Heon Cynn. *Incremental algorithms for general purpose layout system.* PhD thesis, Champaign, IL, USA, 1994.

[3] Willaim J. Dally and Andrew Chang. The role of custom design in asic chips. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 643–647. ACM Press, 2000.

[4] Jeremy Dion and Monier Louis M. Contour: A tile-based gridless router. Technical report, Western Research Laboratory Research Report 95/3, 1995.

[5] Virantha Ekanayake, Clinton Kelly IV, and Rajit Manohar. Bitsnap: Dynamic significance compression for a low power sensor network asynchronous processor. In *Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems*. IEEE Computer Society, 2005.

[6] Sabih H. Gerez. *Algorithms for VLSI Design Automation.* John Wiley & Sons, Inc., 1999.

[7] Gordon T Hamachi. An obstacle-avoiding router for custom vlsi. Technical report, University of California at Berkeley, 1986.

[8] David W. Hightower. A solution to line-routing problems on the continuous plane. In *DAC '69: Proceedings of the 6th annual conference on Design Automation*, pages 1–24. ACM Press, 1969.

[9] R. Eric Lunow. A channelless, multilayer router. In *DAC '88: Proceedings of the 25th ACM/IEEE conference on Design automation*, pages 667–671. IEEE Computer Society Press, 1988.

[10] Edward F. Moore. The shortest path through a maze. In *International Symposium on Theory of Switching*, pages 285–292. Harvard University, 1957.

[11] John K. Ousterhout. Corner stitching: a data structuring technique for vlsi layout tools. Technical report, University of California at Berkeley, 1983.

[12] John K. Ousterhout, Gordon T. Hamachi, Robert N. Mayo, Walter S. Scott, and George S. Taylor. Magic: A vlsi layout system. In *DAC '84: Proceedings of the 21st conference on Design automation*, pages 152–159. IEEE Press, 1984.

[13] Naveed Sherwani. *Algorithms for VLSI Physical Design Automation.* Kluwer Academic Publishers, 1999.

[14] Victor Zyuban, Sameh W. Asaad, Thomas W. Fox, Anne-Marie Haen, Daniel Littrell, and Jaime H. Moreno. Design methodology for semi custom processor cores. In *GLSVLSI '04: Proceedings of the 14th ACM Great Lakes symposium on VLSI*, pages 448–452. ACM Press, 2004.