WRL
Research Report 95/3

Contour:
A Tile-based
Gridless Router

Jeremy Dion
Louis M. Monier

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There are two other research laboratories located in Palo Alto, the Network Systems Lab (NSL) and the Systems Research Center (SRC). Another Digital research group is located in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, WRL-2
250 University Avenue
Palo Alto, California 94301   USA

Reports and technical notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net: `JOVE::WRL-TECHREPORTS`

Internet: `WRL-Techreports@decwrl.pa.dec.com`

UUCP: `decpa!wrl-techreports`

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ``help`` in the Subject line; you will receive detailed instructions.

Reports and technical notes may also be accessed via the World Wide Web: `http://www.research.digital.com/wrl/home.html`.

Contour: A Tile-based Gridless Router

Jeremy Dion
Louis M. Monier

March, 1995

Abstract

We present an automatic maze router based on a corner-stitching data structure. The router is gridless and can deal with arbitrary wire width and spacing rules in any number of layers. It is versatile enough to route dense VLSI chips automatically with results comparable in quality with hand layout.

Table of Contents

List of Figures

1. Introduction

Routing wires in VLSI circuits has been one of the most heavily studied problems in computer-aided design. Most work has been in constrained sub-problems, such as channel and switchbox routing, or routing with a limited number of layers, where a simplified model can yield simpler solutions. In this paper, we describe a router designed for fully automatic connection of wires in dense custom VLSI circuits. It produces hand-quality layout fast enough to have been used to connect all the wires in a custom microprocessor chip without manual intervention [7]. The router is an integral part of a hierarchical VLSI design system [11, 2] in which layout is assembled recursively from hand-crafted circuits and automatically synthesized cells. This design style requires an automatic router which can add wires on top of existing wires and logic on no particular grid, in any wire width, using any number of layers, with results comparable to hand quality over tens of thousands of wires, and without any manual intervention. We knew of no available router which could accomplish this.

Routing a cell is done by breaking each net into a spanning tree of pairwise *connections* between disconnected *terminals*. These connections are then ordered by likely difficulty in order to produce a connection schedule for the cell. The connections are then attempted in order of increasing difficulty. Each connection is completed by finding a design-rule-correct path between its terminals. If a path can be found, the next connection is attempted. In case of blockage, previously made connections are removed and re-routed later.

Routers based on the corner-stitched data structure [15, 8] have been explored previously. Arnold and Scott [1] implemented an interactive router using contours to prevent design-rule errors. Tsai *et al* [16] described a router using both horizontal and vertical *tiling planes* for each layer. Margarino *et al* [9] combined all layers into a single tiling plane. None of the above solved the problem of design-rule correct connections of new wires to arbitrary geometry. None have explicitly represented multiple search paths through a single space tile.

2. Tiles and Contours

Early experiments with gridless maze routers taught us that two facts dominate the problem: the importance of local geometrical operations and the strong constraint of generating only design-rule correct geometry.

2.1. Tiling planes

In the discrete geometrical world of VLSI chips, we need to represent hundreds of millions of rectangles, but the local complexity of any neighborhood is limited. The local nature of geometrical operations is a direct consequence of the way a search operates: explore the neighborhood of a point in order to direct further searches. As a consequence, it is critical to use a data structure which implements local operations in a time related to the local complexity, instead of the total complexity. This excludes quad-trees and other sorted structures, and explains our choice of *corner-stitching* [15] for the following reasons:

- all local operations are performed in near-constant time.

- both geometry and space are explicitly represented; space rectangles are the basis of the search structure described in the next section.

- the memory requirements are linear in the number of rectangles represented.

A tiling (or corner-stitching) plane is a type of planar map restricted to orthogonal geometry; it is a partitioning of the 2D plane into rectangular tiles, with a preferred direction (horizontal or vertical) along which tiles have maximum length. A tile is a simple object described by its *origin*, four corner *stitches* (pointers towards neighboring tiles in the same plane), and a *color*; every point lies in exactly one tile. Once a maximal direction is chosen, the representation of geometry by a tiling plane is canonical. We represent integrated circuits by a collection of tiling planes, one per layer. In an early version of this router, we used two planes with different orientations per layer as in [16], but we later found this to be unnecessary. Each tiling plane contains tiles of two types: solid and clear.

- A *solid* tile describes actual geometry. Its color encodes the type of material, and the connected component of a particular net it belongs to. Simple materials like polysilicon or metals appear only on one layer. Materials for contacts and vias have related tiles on adjacent layers.

- A *clear* tile represents space between solid tiles. Instead of having a single *space* color, we have several clear shades, each representing transparency to wires less than a particular width.

2.2. Contours

A gridless router must generate geometry satisfying the design rules. Any added wire must touch its end points with at least a minimum width, and must stay a minimum spacing away from any other geometry. This problem manifests itself not only when a new wire is drawn, but also during the search, since any valid path must define a design-rule correct wire. For example, it is important to detect quickly that a local search is doomed by the lack of space to accommodate the width and spacing requirements of the current wire.
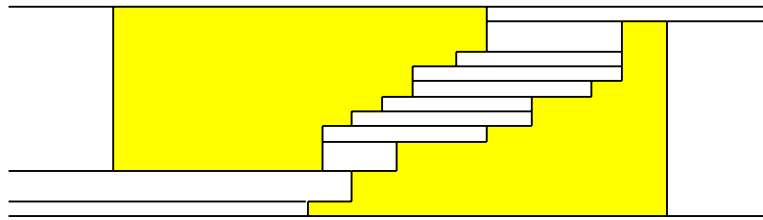
Figure 1: Finding a path by assembling space tiles can be expensive

The brute force approach assembles enough space tiles to check the width and spacing requirements. This operation is critical as it is at the core of the search, and the local geometry can be fragmented enough to make this extremely inefficient, as shown in Figure 1. We experimented with this approach in an early version of the router and found two fundamental flaws: the cost of scanning the space tiles repeatedly was prohibitive, and we were unable to eliminate all design-rule errors.

Figure 2: Two wire geometries and their contours

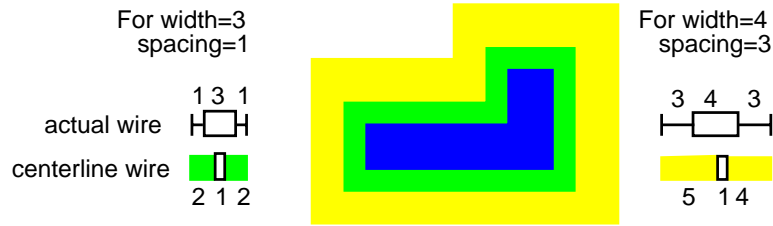Instead, we opted to preprocess the geometry, bloating it by half the space needed for routing a wire (2·*spacing*+*width*−1) and transforming the problem into a unit-width *centerline* routing as in [1]. This greatly simplified the basic search step and the generation of design-rule correct geometry. Figure 2 shows two examples of converting a wire width and space into a centerline wire. The shading around each centerline wire defines a *contour* which is painted around each solid object, and guarantees that any centerline wire outside the contour bloated appropriately satisfies the spacing requirement. Where the total width 2·*spacing*+*width* is even, as in the second example in the figure, the bloat is asymmetric; we choose arbitrarily to bloat the right and top side by one more unit (in this case 5 units) than the bottom and left (4 units).

Figure 3 shows that centerline routing, shown in (b) and (c) on the right column, is strictly equivalent to routing with real widths and spacings, as shown in (a) and (d) on the left.

The general routing problem, however, is not limited to a single layer and a single set of design rules: it usually involves routing wires with different width and spacing requirements on several layers. In order to capture all parameters for one routing problem, one must paint the contours corresponding to all desired wire geometries. When routing a particular wire, contours corresponding to smaller or equal design rules are opaque to the centerline, while others are treated like space.

2.3. Memory requirements

The use of contours cuts down the time spent on the search, at the expense of preprocessing time and storage for the contours. The explicit representation of space tiles in corner stitching requires on average one space tile per solid tile, with a worst case of 3·*n*+1 space tiles for *n* isolated solid tiles. Painting contours further fractures the space tiles, and increases the storage requirements. This cost depends on the actual geometry involved. When the solid geometry is widely spaced, each edge of the solid geometry requires a contour tile, for a worst case of four contour tiles per solid tile. When the solid geometry is packed at the minimum spacing, painting contours simply exchanges space tiles for contour tiles, and the overall cost is negligible. An average routing problem on a VLSI circuit is tightly rather than loosely packed, and becomes tighter as new wires are added.

As a practical but pessimistic example, consider the first metal layer for a 64-bit floating-point divider, consisting of 84K solid rectangles, and a family of wires each two units wider than its predecessor. The storage costs for the contour representation are shown in figure 4. As each contour is added, some space tiles are replaced by tiles of the new contour but overall, a large majority of tiles are contours tiles. For example, with 3 contours, each solid tile is associated with 0.566 space tiles, and 5.678 contour tiles, for a total of 6.244 non-solid tiles.

(a) The original routing problem

(b) Painting the contours

(d) Bloating the path to satisfy the design rules

(c) Finding a unit-width path
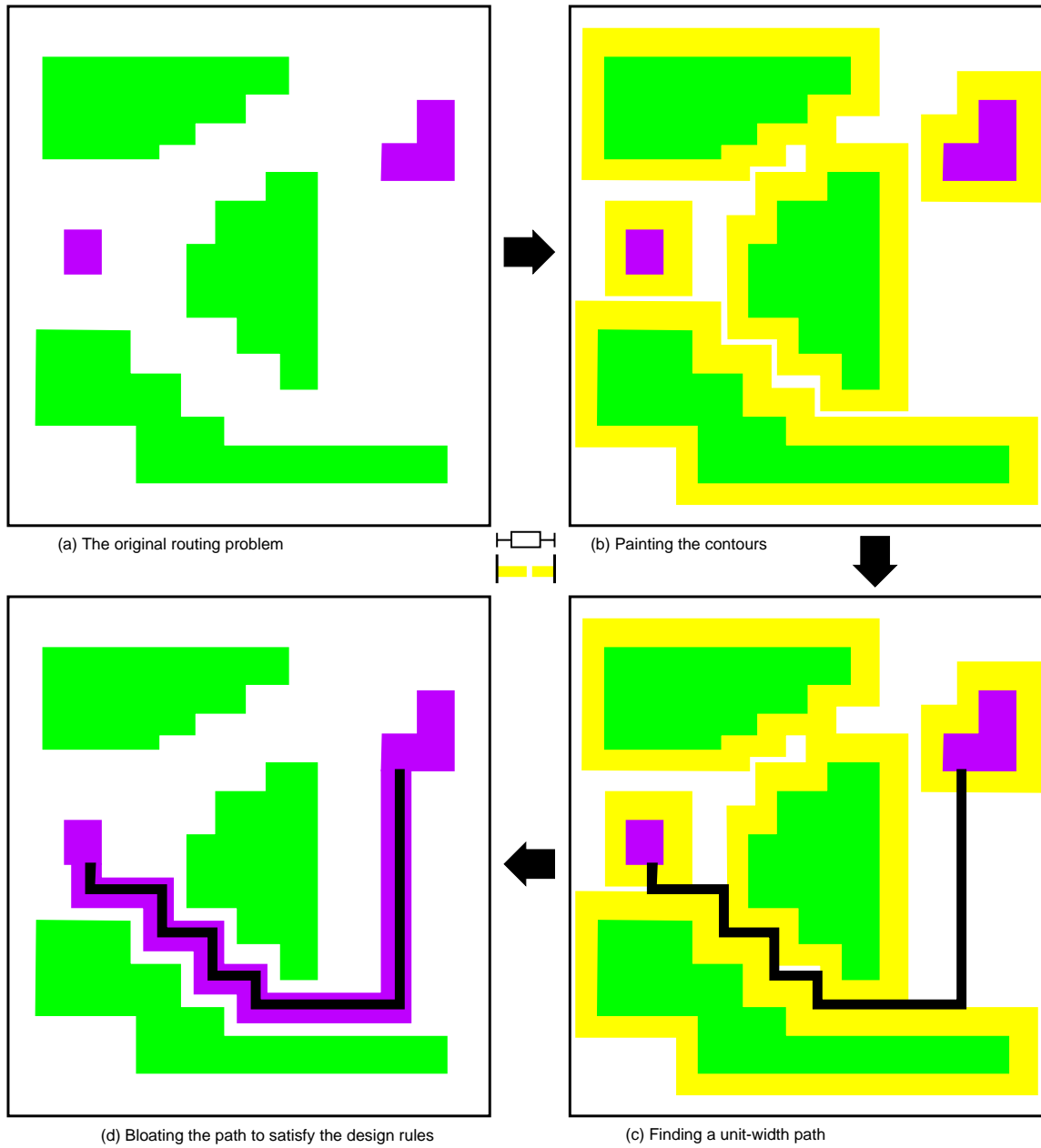
Figure 3: Contour-based routing

# contours	space/solid	contour/solid	non-solid/solid
0	0.734	0	0.734
1	0.967	2.269	3.236
2	0.608	4.151	4.759
3	0.566	5.678	6.244
4	0.535	7.119	7.654
5	0.507	8.503	9.010
6	0.457	9.787	10.244

Figure 4: Space and contour tiles per solid tile for increasing
numbers of contours

3. The Routing Algorithm

CONTOUR uses a routing algorithm which is a hybrid between maze routing [12] and line searching [6]. A single principle underlies this algorithm; ***postpone arbitrary choices***. When such choices arise, such as "should the connection start with a wire or a via?", or "should we turn left or right around this obstacle?", *all* the alternatives are explicitly represented, maintained and propagated until there is enough information to discriminate between them. During early attempts to implement the router, we did not rigorously adhere to this principle, believing it to be too complicated or too costly to implement. The result was always a router that would surprise us by the paths it had chosen for some connections ("Why did it do *that*?"). Only when we finally eliminated all arbitrary choices in the algorithm did the router choose exactly the paths a person would. There are two parts to implementing this principle. First, a path is not a fixed centerline path propagating in free space from a terminal. Instead, it represents a collection of topologically equivalent paths which are resolved to a particular centerline path only when a solution is generated. Second, there may be many paths from a terminal to a particular clear tile. Each path to the tile represents a topologically distinct way of reaching a part of that tile with minimum cost.

In this algorithm, the two terminals are treated symmetrically. The router searches outward from both terminals, and paths starting from each terminal head toward the other. There are three reasons for this. First, as mentioned in [4, 5], it is faster to search from both ends than from one end toward the other due to the smaller area covered by a successful search. Second, blockages are detected quickly, because when a connection fails, it is normally because only one of the terminals is isolated. If a search started only from the unobstructed terminal, there would be no easy way to detect that the obstructed terminal could not be reached. Third, finding legal starting places for a path to leave a terminal is a hard problem. We chose to do this at both ends of the connection rather than solve the symmetrical problem of legal entry to a terminal from paths in free space.

3.1. Starting the Search

Postponing arbitrary choice, we start the search with all legal wires and vias. The terminals need not be simple rectangular pins, and are in general the result of previous routings. Any initial wire or via must touch the terminal geometry with at least a minimum width, and must not create a spacing violation with any other geometry. We begin with the case of starting with a via, as shown in figure 5.

The goal is to place the via inside the terminal geometry on this layer (L1), but in clear space on the second layer (L2). We copy the geometry for the terminal on L1 which can contain a via into a temporary tiling plane (metal1 in figure 5), and shrink it by half the width of a via on this layer (figure 5b). This shrunken geometry defines the centers of vias which lie entirely within the terminal geometry. Its intersection with via-clear tiles on L2 maps the location of legal vias, as shown in figure 5c. For each such intersection, we create a *path* data structure containing the identity of the clear tile and the intersection, and insert it in the path heap. In the figure, paths would be created for both *T1* and *T2*. Because a true via occupies only one unit square in one of these rectangles, each path represents a collection of possible starting vias.
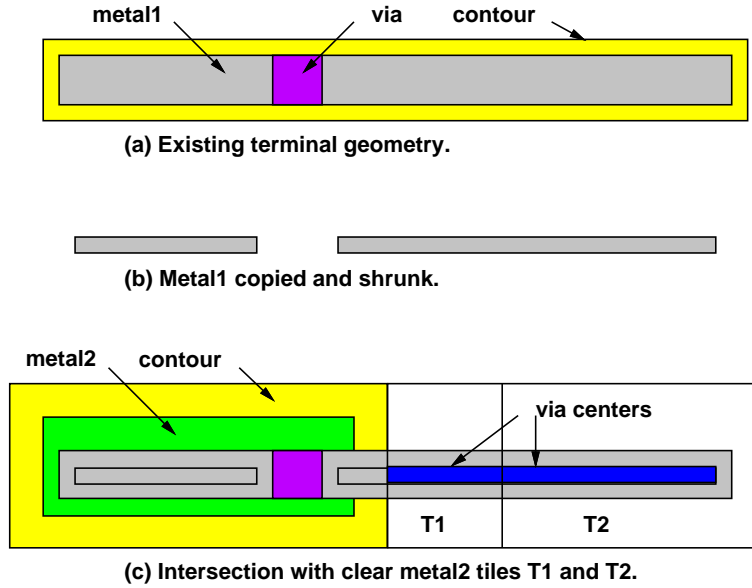
metal1 via contour

(a) Existing terminal geometry.

(b) Metal1 copied and shrunk.

metal2 contour via centers

T1 T2

(c) Intersection with clear metal2 tiles T1 and T2.

Figure 5: Starting With a Via

This algorithm cannot generate vias which are larger than the terminal geometry. This limitation can be overcome with a more general algorithm related to design rule checking. In modern VLSI technologies, however, plug or post vias tend to be as small as wires.
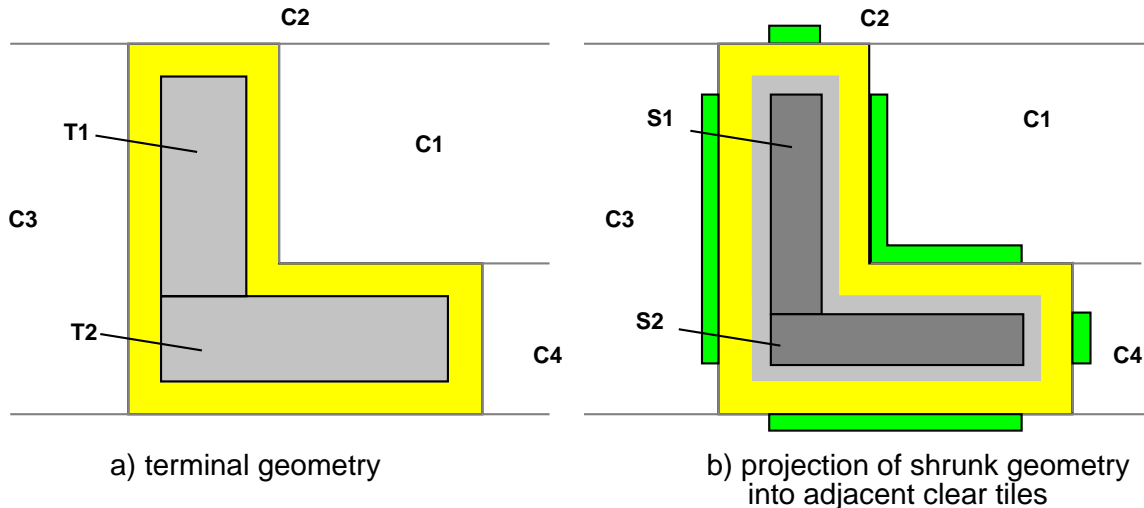
C2 C2

T1 S1 C1

C1

C3 C3

T2 S2

C4 C4

a) terminal geometry b) projection of shrunk geometry into adjacent clear tiles

Figure 6: Starting With a Wire

We next consider the case of generating initial wires, as shown in figure 6. Here, connecting terminal geometry to a clear tile on the same layer requires crossing opaque contour tiles without creating width or spacing errors, in particular notches. The first step is to shrink all terminal geometry on this layer by half the width of wires, using a temporary tiling plane. In figure 6b, tiles *T1* and *T2* produce tiles *S1* and *S2* in the temporary plane, here shown superimposed on the original tiles for clarity. Next, for each of these shrunken tiles we explore the neighborhood within *width+space* in the original plane, looking for clear tiles which overlap the shrunken tile in one dimension. In figure 6, the search around *S1* will find clear tiles *C1*, *C2*, and *C3*. All such

tiles will be within *width*/2+*space* of the terminal geometry. Between each of these tiles *Ci* and the shrunken tile *Sj*, there is an intervening area covered by opaque tiles, and normally forbidden to wires. But any horizontal strip of height 1 starting in the shaded area of *C3* and connecting *C3* to *T1*, for instance, will touch *T1* with a full wire width, and will not create a design rule error with any other part of *T1* or *T2*. These shaded areas in clear tiles abutting opaque tiles around the terminal define where the opaque tiles may be crossed legally. For each such clear tile, we create a *path* data structure containing the identity of the clear tile and the rectangle connecting it to the terminal geometry. Note that in the example, *C1* has two paths created into it, one describing a horizontal connection to *T1*, and one a vertical connection to *T2*.

3.2. Cost Function

It is our belief that the only good cost functions have a physical basis, usually related to the amount of material needed to complete the connection. We use the A* heuristic [3], where the cost of a path is the sum of the exact Manhattan distance from the source to the path (source cost), and a lower bound on the remaining distance to the destination (destination cost). We modify it by assigning a horizontal and vertical *weight* to each layer, which is multiplied by distance travelled on this layer. Weights are used for the different materials and directions in order to control the style of routing. For example, if one specifies that routing on metal1 has a cost of 1 in the horizontal direction, and 10 in the vertical direction, and reverses these coefficient for metal2, routing will have a very strong directionality, similar to the results of a channel router, where even the smallest vertical movement will require a layer change to metal2. If the differences in relative cost is smaller, however, jogs or doglegs will be preferred to layer changes for small distances.

3.3. Paths and Path Heaps

At the start of path propagation, we have two **path heaps**, one for each terminal, seeded with paths for initial vias and wires. The heaps, or priority queues [14], are used to maintain all paths in order of increasing cost. The *paths* stored in the path heaps describe the end points of design-rule correct paths in clear tiles leading from a terminal to a tile. More precisely, a *path* is a data structure containing:

- a *tile* pointer to the clear tile in which the path ends.

- the *source cost* of getting to this tile from the starting terminal.

- the *minimal cost rectangle* in the tile which can be reached for the source cost. This defines a sort of "equipotential" for source cost; by definition any unit square in the rectangle can be reached for the source cost recorded in the path. The rectangle may often be a single unit square, but when it is not as in figure 7, it represents the end points of a collection of topologically equivalent paths with the same cost.

- the *destination cost*, a lower bound on the cost with which the path to the destination can be completed. This is simply the Manhattan distance from the minimal cost rectangle to the bounding box of the other terminal.

- a *back pointer* to the parent path. By following the chain of back pointers in a path to previous paths, we can find the sequence of paths - and therefore of clear tiles - which lead from the current clear tile to the terminal.

● a *next pointer* to the next path to the same tile. Each clear tile contains a path pointer to the head of the list of all paths to this tile, linked by the *next path* pointer. When a new path is generated, it is added to the path heap, and it is also linked to the chain for its clear tile.

3.4. Path Propagation

The basic step during maze searching is *path propagation*, and it continues until either one path heap is empty or an acceptable solution is found. If one path heap becomes empty with no solution found, then all the accessible free space around one of the terminals has been explored, and no solution is possible.

The path heaps are processed alternately during the search, by removing the least cost path from one heap, and then the other. The path removed from the heap is the one whose sum of source cost and destination cost is least. The least cost path defines a clear tile which can be reached from the terminal. To propagate the path, we must find all connected clear tiles on this and adjacent layers, and generate new paths to them. A path to a neighboring tile on the same layer corresponds to extending a wire on the layer. A path to a tile on an adjacent layer corresponds to changing layers with a via. Following the principle of no arbitrary choices, we generate all such paths which reach their tiles with minimum source cost. These become the descendents of the current path, and are inserted into the heap.
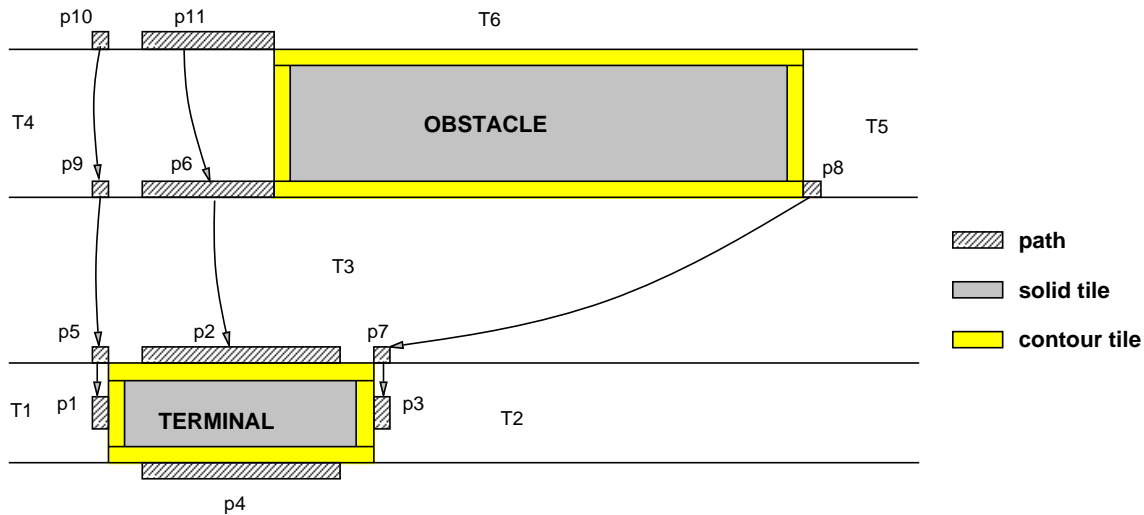

Figure 7: Propagating Wires

The case of propagating a wire is shown in figure 7. Paths are represented by their minimal cost rectangles within their tiles, and the back pointers from paths to the ones that generated them are represented by arrows. The figure illustrates that a tile in general has many paths to it; note that *T3* has three paths, *p2*, *p5*, and *p7*. Suppose that *p6* is the least cost path in its heap. To propagate *p6* from clear tile *T4*, we examine all tiles around the perimeter of *T4*. Any clear tile which shares a border of at least one unit with *T4* is eligible - diagonal abutment at corners is not. Of *T4*'s bordering tiles, only *T6* and *T3* are clear. Of these, *p6* can generate a minimal-cost path only to *T6* at *p11*. In the other clear tiles, paths from *p6* are longer than paths from other

sources, and they are discarded. Note that two paths survive in *T6*: *p10-p9-p5-p1* and *p11-p6-p2*. This is correct; depending on the precise location of the destination geometry, either could be the better path. Note also that *p6* is not a unit square, and so it defines a set of possible paths with equal cost. Again, depending on the precise geometry of a solution through *p6*, any one of the possible unit squares could be the best center for the actual wire.
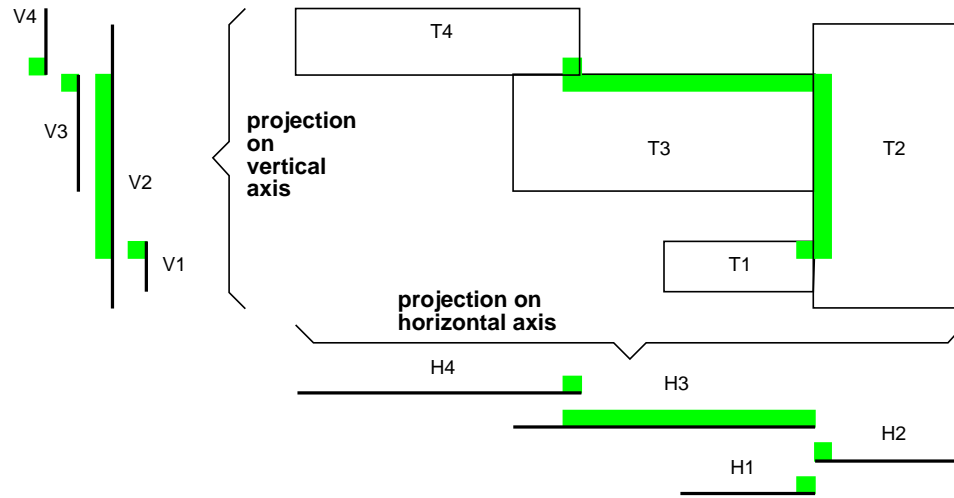

Figure 8: Minimal Cost Path Through a Sequence of Tiles

3.5. Finding the Centerline Route

Once a neighboring tile of a path is found, a new path must be generated for it. Figure 8 shows how the source cost and the minimal-cost rectangle for the new path is computed. The figure shows a sequence of clear tiles *T1* to *T4*, and the minimal-length centerline path which connects them. Since the path lies entirely within the sequence of tiles, it would generate a design-rule correct wire when expanded to full width. The general case of this problem has been explored in [10].

The centerline path is generated by projecting the general two-dimensional problem onto the horizontal and vertical axes, as shown below and on the left of the figure. Each tile becomes a line segment in both projections. For each projection, we solve the one-dimensional problem of moving a unit line segment from the nearest point in the *H4* or *V4* to the nearest point of *H1* or *V1*. This is done by maintaining for each line segment an entry point, and by looking at succeeding segments for the exit point. The exit point is found when a successor segment does not overlap the current segment, as for example, when *H2* does not overlap *H4*. The exit point is then fixed as the point in the current segment nearest the non-overlapping successor. Each intervening segment is then trimmed to have the entry and exit points as ends. The entry point floats for the first segment. The exit point floats for the last segment.

This algorithm tries not to move the entry point into a segment until necessary. When the two one-dimensional solutions are projected back into two dimensions, a centerline path with the minimal number of bends is generated. The algorithm yields the minimal cost rectangle for the path directly, and its source cost is the sum of the horizontal and vertical distances weighted by horizontal and vertical unit costs of this layer.

The case for generating path descendants on adjacent layers involves simple extensions of the case for a wire. Instead of searching the tiles bordering the clear tile on its own layer, we search for clear tiles intersecting it on a neighboring layer. Any areas of intersection of clear tiles on the two layers must be unobstructed on both layers and therefore legal places for the centers of vias. The centerline path generation algorithm for computing the source cost and minimal cost rectangle illustrated in figure 8 works just as well when the sequence of clear tiles lies on several layers; some tiles in the sequence will properly intersect, rather than merely abut. The algorithm must be generalized slightly to move preferentially on the horizontal and vertical line segments having minimal unit cost.

3.6. Pruning Paths

In the previous section, we indicated that a path is added to a tile only if it has minimal cost. We now make this notion precise.
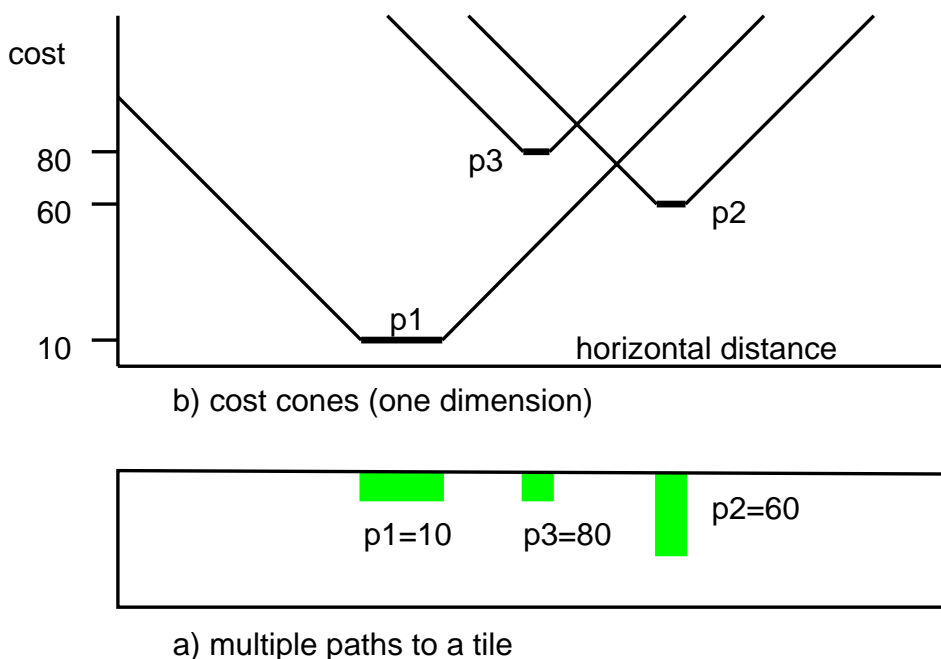

b) cost cones (one dimension)

a) multiple paths to a tile

Figure 9: Redundant Paths to a Tile

Figure 9a shows a single tile with three paths to it from the same terminal. Each path is represented by a rectangle and the cost of reaching any point in that rectangle. Suppose that this wiring layer has unit cost in both dimensions. We can represent the paths by a cost versus distance graph as shown in figure 9b. In this figure, each path defines a cone of the cost for which any point in the tile can be reached. The lowest point in the cone is the source cost of the path; the interior of the minimal-cost rectangle can be reached for this cost. Points outside the minimal-cost rectangle can be reached for the additional cost of a wire in this tile. In the figure, we have drawn only the one-dimensional horizontal projections of these cones for simplicity, though they are in reality two-dimensional cost functions of x and y. The figure illustrates in one dimension the principle that we may eliminate any path whose cost cone lies entirely within the cone of another path. Thus, in the example, *p3* is made redundant by *p1*, but *p1* and *p2* are not mutually redundant, since each can reach points in the tile for minimal cost.

3.7. Selecting a Solution

A solution to the maze search is found when a path is added to a tile which already has a path from the other terminal. When this happens, a connected sequence of clear tiles exists between the source and destination. The algorithm for finding the minimal-cost centerline path described above will generate the cheapest path through these tiles, and its 'source cost' is the actual cost of the solution.
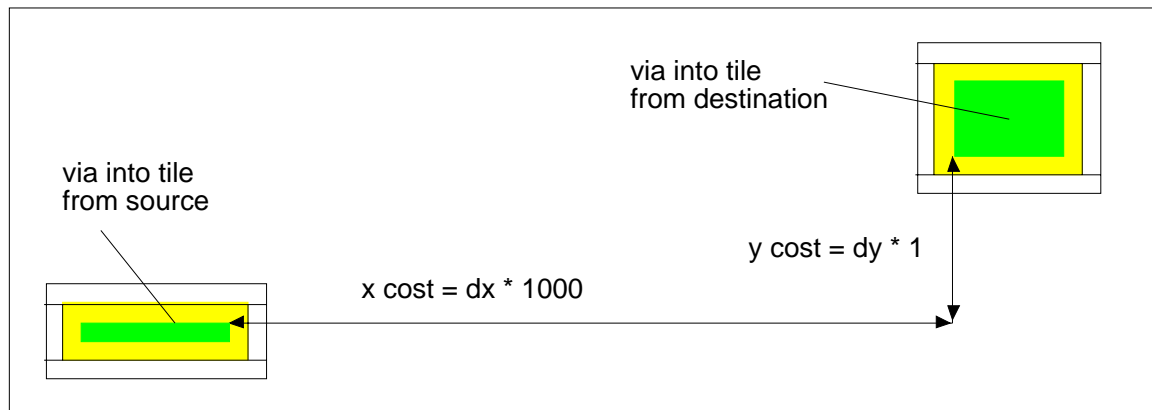

Figure 10: An Unacceptably Expensive Solution

Is the first solution found the best one? Often not, as shown in figure 10 where initial via paths from both terminals are created in the same tile on a layer with expensive horizontal movement. The source cost of each via is small, and the estimated cost of completing the paths through these vias is also small, because the estimate uses best-case costs. But since the direct solution connecting these two vias requires horizontal movement on an unfavored layer, it is very expensive.

Even if the first solution is not the best, it gives useful upper bounds on the cost of any further solutions.

- If all paths in the heaps have a combined source and destination cost larger than the solution cost, we stop the search since any future solution will be more expensive. This is an easy test, since heaps are sorted by this combined cost.

- A new path should be added to a heap only if its combined source and destination cost is less than the current best solution. We may limit the area searched for neighboring tiles by using simple Manhattan distance as a lower bound on the estimated cost: all paths whose Manhattan distance to the destination is larger than the best solution cost minus the path source cost are discarded. This defines a Manhattan ellipse between the path's minimum-cost rectangle and the destination terminal, as shown in figure 11.

As further solutions are generated, some may have lesser cost, and we continually retain the best, further tightening our criterion for stopping the search.

Despite the additional pruning which can be done once the first solution is found, experience showed that most of the search is done to prove that a very good solution found early on is indeed optimal. For each solution, we define a *cost ratio* as the actual cost of the path divided by the cost of a Manhattan path between the same endpoints on the cheapest layers. The *cost ratio*
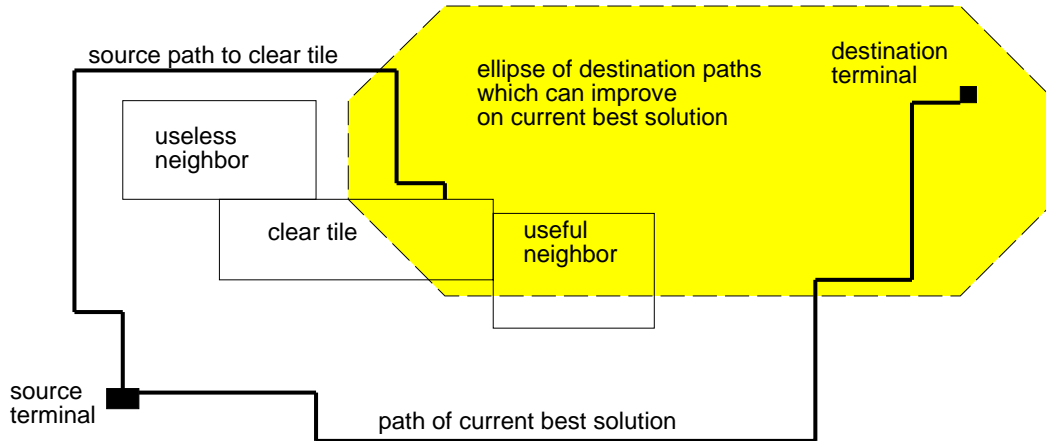
Figure 11: Manhattan Ellipse of Path Completion

can never be less than 1. By stopping the search when the cost ratio of the current best solution is below a user-settable threshold (say 1.05), simple connections can be stopped early while guaranteeing that the solution is within 5% of Manhattan length. This provides a large speed increase for a limited loss in routing quality.

3.8. Design-Rule Errors in the Solution

The contour structure prevents design-rule errors of new geometry with existing geometry. It cannot tell us whether a solution will have design rule errors with itself when converted into real geometry. Width errors are not possible, because a centerline path always bloats into geometry touching the terminals with legal width. Spacing errors on one layer are also not possible. They require two parallel edges of the geometry to be separated by less than a minimum spacing. No legal geometry can lie between these edges, and so they must form part of a U-shaped loop. Therefore there must be a shorter solution which does not contain the loop, and prolonging the search will find it. Vias are the remaining source of potential design-rule errors. A simple example is a solution requiring a change from first-level to third-level metal. It would be quite easy to create an illegal spacing between the second-metal images of the first-to-second and second-to-third vias. These are prevented by special-case code as mentioned in [1].

4. Special Wiring Problems

4.1. Busses

A feature obvious to the eye in VLSI circuits is the regular routing of large data busses consisting of tens or hundreds of wires. Maximum density dictates that when a wide bus is used to connect different functional units on a chip, the individual wires in it should run in parallel and that turns and layer changes should be done in a regular and stylized way. On one layer, this style of routing is known as *river routing* [13].

We make no special provision for routing of many wires in parallel. Instead, we note that regular wiring is by definition easy to describe, and we provide a mechanism to add segments of

wire for particular nets during cell placement. Like all placement operations, this is done by writing code, not by editing layout graphically. By making this relative to edge and corner alignments, wiring can be added in a flexible way which will adapt to future layout changes. An example of such a wiring command might be "add a vertical metal2 bus for dataIn running on the left of the data path and extending for its full height". Because the individual wires of the bus are associated with the netlist, the router sees the bus as just another piece of geometry to be connected on each of the nets of the bus. By judiciously inserting wiring in this way during placement, we have found it extremely easy to direct the router to make the obvious right choices without further assistance.

A second situation requiring regular wiring is busses embedded in data paths, where a track in each repeated bit pitch is used to carry one bit of a bus over the length of the data path. Here, as above, it is also possible to lay down explicitly the wire used for the bus in each bit pitch. We have also used a simple and more flexible strategy with success. Most nets are broken down into pairwise connections by repeatedly adding the shortest necessary connection in the normal spanning tree algorithm. However, busses embedded in data paths are more optimally routed as long straight tracks running the length of the data path with short spurs to the individual device terminals along the way. This can be easily described by beginning the spanning tree with the connection between the two most *distant* terminals, rather than the two nearest, and then finishing the spanning tree normally. The most obvious choice for the router will then be to make the first connection with a single straight wire between the end points of the bus, and then to add short stubs for all remaining connections.

When to use these bus routing strategies is at the discretion of the designer, not decided automatically. Layout is an iterative process of improvement, not perfect first time. The first time layout is generated for a cell, the result may be an unroutable mess due to lack of regularity. The simplest strategy is then to select some nets in the cell to be routed with the bus spanning tree algorithm and try again. This may produce sufficient improvement to route the cell. If not, track assignment can be done by hand, and segments of wire added by code during placement to leave no room for error.

4.2. Power Supplies and Clocks

Power supply nets, voltage reference nets, and clock nets provide special problems for a router, since their primary function is analog, not digital. Voltage, current, capacitance and resistance of these nets are critical issues, and they do not fit the model of a net in which all the components may be connected by a spanning tree. Power and clock nets are often multiply connected in a grid pattern, for instance. In general, the geometry of these nets must largely be designed by hand as an integral part of the layout of a VLSI circuit, and automatic routing can be only help with the details. For power nets, a useful routing primitive is to extend the ends of power bars of subcells to abut with matching power bars in adjacent subcells, so that when subcells do not abut perfectly along their boundaries, the gaps in the power routing are filled in.

In clock distribution trees, it is important to define the sources and sinks of clock signals at each level in the tree so as to connect each sink to its source rather than to another nearer sink.

5. Results

CONTOUR is part of a set of tools which has been under development for several years. An earlier version was used to route BIPS-0, a bipolar processor [7]. The layout generation of this complete 700,000-device circuit took 10 hours, most of which was spent in routing. This allowed one complete iteration per day. It has also been used for the layout of large sections of BIPS-1, a 4,000,000-device BiCMOS microprocessor.
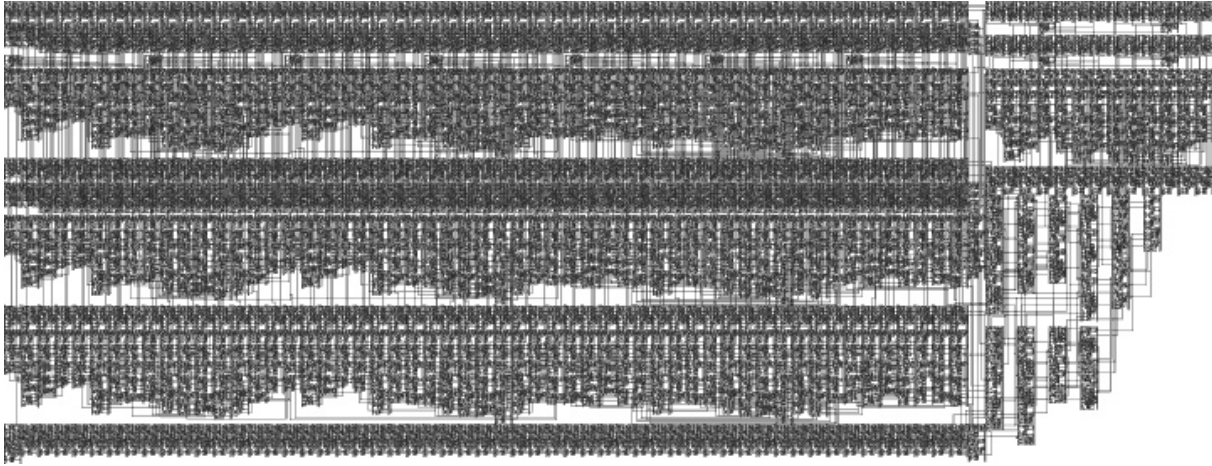

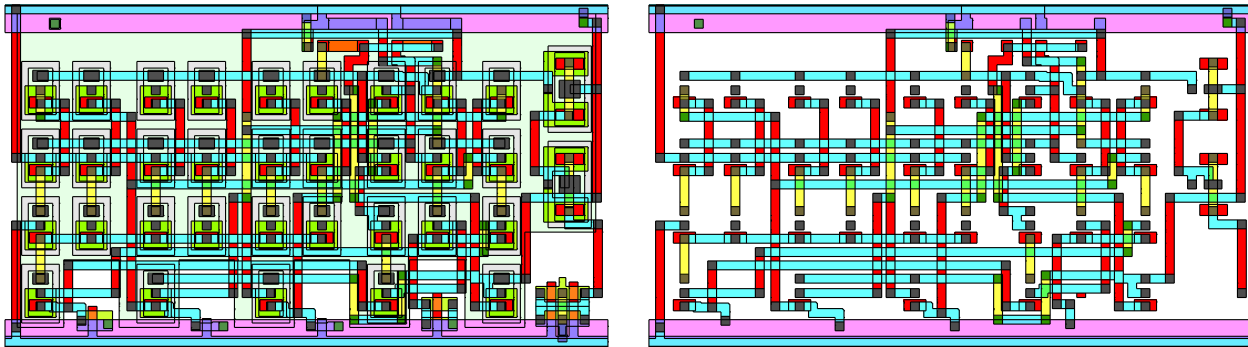**Figure 12:** BIPS-1 Floating-Point Divider routed with CONTOUR

Figure 13: A synthesized cell; the 4-layer routing added by CONTOUR

The performance of CONTOUR is illustrated in the 64-bit floating-point divider of BIPS-1, shown in figure 12. The routing is done hierarchically in 118 unique cells, 97 of which are synthesized gate-level cells like the one shown in figure 13. The wiring added by CONTOUR in figure 13, shown separately on the right, is on four layers (three metal layers plus polysilicon) and the routing was done in a few seconds on a DEC 3000/800. Another 21 cells in the divider are composite cells such as 56-bit registers, carry-lookahead adders, or blocks of control logic. Routing the 7495 connections of the divider takes 532 seconds on a DEC 3000/800 for an overall speed of 14 connections per second. Considering all 118 cell routings as a whole, 20% of the time is spent painting the contours before routing starts, and 80% actually doing the routing. Of this 80%, 38% of the time is spent generating the initial via and wire positions to start the search, and 42% of the time is spent in path propagation.

The speed and quality of CONTOUR are based on the combination of contour-based routing and the avoidance of arbitrary choices in the search algorithm. We believe that this approach is more effective than the restrictive channel model requiring an artificial partitioning of chip area, and we see no reason to restrict routing to two or three layers at a time. In our design style, CONTOUR is the only router for gate-level synthesized cells, data paths, random logic blocks and global assembly; the conventional use of different algorithms at these levels seems a historical accident. We also see no reason why CONTOUR'S algorithms and data structures cannot be adapted in a straightforward way to gate arrays, printed circuit boards, and other routing problems in electronic design.

[1]   M.H. Arnold , W.S. Scott. An Interactive Maze Router with Hints. In *25th Design Automation Conference*, pages 672-676. June, 1988.

[2]   Jeremy Dion, Louis Monier. *Design Tools for BIPS-0*. WRL Technical Note 32, Digital Equipment Western Research Laboratory, December, 1992.

[3]   G.W. Clow. A Global Routing Algorithm for General Cells. In *21st Design Automation Conference*, pages 45-51. June, 1984.

[4]   J. Dion. Fast Printed Circuit Board Routing. In *24th Design Automation Conference*. June, 1987.

[5]   J. Dion. *Fast Printed Circuit Board Routing*. WRL Research Report 88/1, Digital Equipment Western Research Laboratory, 1988.

[6]   Hightower D. A Solution to Line Routing Problems on the Continuous Plane. *Proc. Design Automation Workshop* :1-24, 1969.

[7]   N.P. Jouppi, P. Boyle, J. Dion, M.J. Doherty, A. Eustace, R.W. Haddad, R. Mayo,
        S. Menon, L.M. Monier, D. Stark, S. Turrini, J.L. Yang, W.R. Hamburgen, J.S. Fitch,
R. Kao. A 300-MHz 115-W 32-b Bipolar ECL Microprocessor. In *IEEE Journal of Solid-State Circuits*. November, 1993.

[8]   J. Ousterhout, G. Hamachi, R. Mayo, W. Scott, and G.S. Taylor. The Magic VLSI Layout System. *IEEE Design and Test of Computers* 2(1):19-30, February, 1985.

[9]   A. Margarino, A. Romano, A. De Gloria, F. Curatelli, P. Antognetti. A Tile-Expansion Router. *IEEE Transactions on Computer-Aided Design* CAD-6(4):507-517, July, 1987.

[10]   K.M. McDonald, J.G. Peters. Smallest Paths in Simple Rectilinear Polygons. *IEEE Transactions on Computer-Aided Design* 11(7):864-875, July, 1992.

[11]   L.M. Monier, J. Dion. Recursive Layout Generation. In *Proc. 16th Conference on Advanced Research in VLSI*, pages 172-184. IEEE Computer Society Press, March, 1995.

[12]   Moore E.F. Shortest Path Through a Maze. In *Annals of the Computation Laboratory of Harvard University*, pages 285-292. Harvard Univ. Press, Cambridge Mass., 1959.

[13]   R.Y. Pinter. River routing: methodology and analysis. In *3rd Caltech conference on VLSI*, pages 141-163. March, 1983.

[14]   R. E. Tarjan. *Data Structures and Network Algorithms.* CMBS-Regional Conference Series, Volume 44, 1983.

[15]   J.K. Ousterhout. Corner Stitching: A Data Structuring Technique for VLSI Layout Tools. *IEEE Transactions on Computer-Aided Design* CAD-3(1):87-89, January, 1984.

[16]   Chia-Chun Tsai, Sao-Jie Chen, Wu-Shiung Feng. An H-V Alternating Router. *IEEE Transactions on Computer-Aided Design* 11(8):976-991, August, 1992.

WRL Research Reports

''Titan System Manual.''
Michael J. K. Nielsen.
WRL Research Report 86/1, September 1986.

''Global Register Allocation at Link Time.''
David W. Wall.
WRL Research Report 86/3, October 1986.

''Optimal Finned Heat Sinks.''
William R. Hamburgen.
WRL Research Report 86/4, October 1986.

''The Mahler Experience: Using an Intermediate
Language as the Machine Description.''
David W. Wall and Michael L. Powell.
WRL Research Report 87/1, August 1987.

''The Packet Filter: An Efficient Mechanism for
User-level Network Code.''
Jeffrey C. Mogul, Richard F. Rashid, Michael
J. Accetta.
WRL Research Report 87/2, November 1987.

''Fragmentation Considered Harmful.''
Christopher A. Kent, Jeffrey C. Mogul.
WRL Research Report 87/3, December 1987.

''Cache Coherence in Distributed Systems.''
Christopher A. Kent.
WRL Research Report 87/4, December 1987.

''Register Windows vs. Register Allocation.''
David W. Wall.
WRL Research Report 87/5, December 1987.

''Editing Graphical Objects Using Procedural
Representations.''
Paul J. Asente.
WRL Research Report 87/6, November 1987.

''The USENET Cookbook: an Experiment in
Electronic Publication.''
Brian K. Reid.
WRL Research Report 87/7, December 1987.

''MultiTitan: Four Architecture Papers.''
Norman P. Jouppi, Jeremy Dion, David Boggs, Mich-
ael J. K. Nielsen.
WRL Research Report 87/8, April 1988.

''Fast Printed Circuit Board Routing.''
Jeremy Dion.
WRL Research Report 88/1, March 1988.

''Compacting Garbage Collection with Ambiguous
Roots.''
Joel F. Bartlett.
WRL Research Report 88/2, February 1988.

''The Experimental Literature of The Internet: An
Annotated Bibliography.''
Jeffrey C. Mogul.
WRL Research Report 88/3, August 1988.

''Measured Capacity of an Ethernet: Myths and
Reality.''
David R. Boggs, Jeffrey C. Mogul, Christopher
A. Kent.
WRL Research Report 88/4, September 1988.

''Visa Protocols for Controlling Inter-Organizational
Datagram Flow: Extended Description.''
Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik,
Kamaljit Anand.
WRL Research Report 88/5, December 1988.

''SCHEME->C A Portable Scheme-to-C Compiler.''
Joel F. Bartlett.
WRL Research Report 89/1, January 1989.

''Optimal Group Distribution in Carry-Skip Adders.''
Silvio Turrini.
WRL Research Report 89/2, February 1989.

''Precise Robotic Paste Dot Dispensing.''
William R. Hamburgen.
WRL Research Report 89/3, February 1989.

''Simple and Flexible Datagram Access Controls for Unix-based Gateways.''
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

''Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.''
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

''Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.''
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

''A Unified Vector/Scalar Floating-Point Architecture.''
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

''Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.''
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

''Integration and Packaging Plateaus of Processor Performance.''
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

''A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.''
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

''The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance.''
Norman P. Jouppi.
WRL Research Report 89/13, July 1989.

''Long Address Traces from RISC Machines: Generation and Analysis.''
Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.
WRL Research Report 89/14, September 1989.

''Link-Time Code Modification.''
David W. Wall.
WRL Research Report 89/17, September 1989.

''Noise Issues in the ECL Circuit Family.''
Jeffrey Y.F. Tang and J. Leon Yang.
WRL Research Report 90/1, January 1990.

''Efficient Generation of Test Patterns Using Boolean Satisfiablilty.''
Tracy Larrabee.
WRL Research Report 90/2, February 1990.

''Two Papers on Test Pattern Generation.''
Tracy Larrabee.
WRL Research Report 90/3, March 1990.

''Virtual Memory vs. The File System.''
Michael N. Nelson.
WRL Research Report 90/4, March 1990.

''Efficient Use of Workstations for Passive Monitoring of Local Area Networks.''
Jeffrey C. Mogul.
WRL Research Report 90/5, July 1990.

''A One-Dimensional Thermal Model for the VAX 9000 Multi Chip Units.''
John S. Fitch.
WRL Research Report 90/6, July 1990.

''1990 DECWRL/Livermore Magic Release.''
Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi.
WRL Research Report 90/7, September 1990.

''Pool Boiling Enhancement Techniques for Water at Low Pressure.''
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 90/9, December 1990.

''Writing Fast X Servers for Dumb Color Frame Buffers.''
Joel McCormack.
WRL Research Report 91/1, February 1991.

''A Simulation Based Study of TLB Performance.''
J. Bradley Chen, Anita Borg, Norman P. Jouppi.
WRL Research Report 91/2, November 1991.

''Analysis of Power Supply Networks in VLSI Circuits.''
Don Stark.
WRL Research Report 91/3, April 1991.

''TurboChannel T1 Adapter.''
David Boggs.
WRL Research Report 91/4, April 1991.

''Procedure Merging with Instruction Caches.''
Scott McFarling.
WRL Research Report 91/5, March 1991.

''Don't Fidget with Widgets, Draw!.''
Joel Bartlett.
WRL Research Report 91/6, May 1991.

''Pool Boiling on Small Heat Dissipating Elements in Water at Subatmospheric Pressure.''
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 91/7, June 1991.

''Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments.''
G. May Yip.
WRL Research Report 91/8, June 1991.

''Interleaved Fin Thermal Connectors for Multichip Modules.''
William R. Hamburgen.
WRL Research Report 91/9, August 1991.

''Experience with a Software-defined Machine Architecture.''
David W. Wall.
WRL Research Report 91/10, August 1991.

''Network Locality at the Scale of Processes.''
Jeffrey C. Mogul.
WRL Research Report 91/11, November 1991.

''Cache Write Policies and Performance.''
Norman P. Jouppi.
WRL Research Report 91/12, December 1991.

''Packaging a 150 W Bipolar ECL Microprocessor.''
William R. Hamburgen, John S. Fitch.
WRL Research Report 92/1, March 1992.

''Observing TCP Dynamics in Real Networks.''
Jeffrey C. Mogul.
WRL Research Report 92/2, April 1992.

''Systems for Late Code Modification.''
David W. Wall.
WRL Research Report 92/3, May 1992.

''Piecewise Linear Models for Switch-Level Simulation.''
Russell Kao.
WRL Research Report 92/5, September 1992.

''A Practical System for Intermodule Code Optimization at Link-Time.''
Amitabh Srivastava and David W. Wall.
WRL Research Report 92/6, December 1992.

''A Smart Frame Buffer.''
Joel McCormack & Bob McNamara.
WRL Research Report 93/1, January 1993.

''Recovery in Spritely NFS.''
Jeffrey C. Mogul.
WRL Research Report 93/2, June 1993.

''Tradeoffs in Two-Level On-Chip Caching.''
Norman P. Jouppi & Steven J.E. Wilton.
WRL Research Report 93/3, October 1993.

''Unreachable Procedures in Object-oriented Programing.''
Amitabh Srivastava.
WRL Research Report 93/4, August 1993.

''An Enhanced Access and Cycle Time Model for On-Chip Caches.''
Steven J.E. Wilton and Norman P. Jouppi.
WRL Research Report 93/5, July 1994.

''Limits of Instruction-Level Parallelism.''
David W. Wall.
WRL Research Report 93/6, November 1993.

''Fluoroelastomer Pressure Pad Design for Microelectronic Applications.''
Alberto Makino, William R. Hamburgen, John S. Fitch.
WRL Research Report 93/7, November 1993.

''A 300MHz 115W 32b Bipolar ECL Microprocessor.''
Norman P. Jouppi, Patrick Boyle, Jeremy Dion, Mary Jo Doherty, Alan Eustace, Ramsey Haddad, Robert Mayo, Suresh Menon, Louis Monier, Don Stark, Silvio Turrini, Leon Yang, John Fitch, William Hamburgen, Russell Kao, and Richard Swan.
WRL Research Report 93/8, December 1993.

''Link-Time Optimization of Address Calculation on a 64-bit Architecture.''
Amitabh Srivastava, David W. Wall.
WRL Research Report 94/1, February 1994.

''ATOM: A System for Building Customized Program Analysis Tools.''
Amitabh Srivastava, Alan Eustace.
WRL Research Report 94/2, March 1994.

''Complexity/Performance Tradeoffs with Non-Blocking Loads.''
Keith I. Farkas, Norman P. Jouppi.
WRL Research Report 94/3, March 1994.

''A Better Update Policy.''
Jeffrey C. Mogul.
WRL Research Report 94/4, April 1994.

''Boolean Matching for Full-Custom ECL Gates.''
Robert N. Mayo, Herve Touati.
WRL Research Report 94/5, April 1994.

''Software Methods for System Address Tracing: Implementation and Validation.''
J. Bradley Chen, David W. Wall, and Anita Borg.
WRL Research Report 94/6, September 1994.

''Performance Implications of Multiple Pointer Sizes.''
Jeffrey C. Mogul, Joel F. Bartlett, Robert N. Mayo, and Amitabh Srivastava.
WRL Research Report 94/7, December 1994.

''How Useful Are Non-blocking Loads, Stream Buffers, and Speculative Execution in Multiple Issue Processors?.''
Keith I. Farkas, Norman P. Jouppi, and Paul Chow.
WRL Research Report 94/8, December 1994.

''Recursive Layout Generation.''
Louis M. Monier, Jeremy Dion.
WRL Research Report 95/2, March 1995.

''Contour: A Tile-based Gridless Router.''
Jeremy Dion, Louis M. Monier.
WRL Research Report 95/3, March 1995.

''The Case for Persistent-Connection HTTP.''
Jeffrey C. Mogul.
WRL Research Report 95/4, May 1995.

''Network Behavior of a Busy Web Server and its Clients.''
Jeffrey C. Mogul.
WRL Research Report 95/5, June 1995.

WRL Technical Notes

''TCP/IP PrintServer: Print Server Protocol.''
Brian K. Reid and Christopher A. Kent.
WRL Technical Note TN-4, September 1988.

''TCP/IP PrintServer: Server Architecture and Implementation.''
Christopher A. Kent.
WRL Technical Note TN-7, November 1988.

''Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer.''
Joel McCormack.
WRL Technical Note TN-9, September 1989.

''Why Aren't Operating Systems Getting Faster As Fast As Hardware?''
John Ousterhout.
WRL Technical Note TN-11, October 1989.

''Mostly-Copying Garbage Collection Picks Up Generations and C++.''
Joel F. Bartlett.
WRL Technical Note TN-12, October 1989.

''The Effect of Context Switches on Cache Performance.''
Jeffrey C. Mogul and Anita Borg.
WRL Technical Note TN-16, December 1990.

''MTOOL: A Method For Detecting Memory Bottlenecks.''
Aaron Goldberg and John Hennessy.
WRL Technical Note TN-17, December 1990.

''Predicting Program Behavior Using Real or Estimated Profiles.''
David W. Wall.
WRL Technical Note TN-18, December 1990.

''Cache Replacement with Dynamic Exclusion''
Scott McFarling.
WRL Technical Note TN-22, November 1991.

''Boiling Binary Mixtures at Subatmospheric Pressures''
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Technical Note TN-23, January 1992.

''A Comparison of Acoustic and Infrared Inspection Techniques for Die Attach''
John S. Fitch.
WRL Technical Note TN-24, January 1992.

''TurboChannel Versatec Adapter''
David Boggs.
WRL Technical Note TN-26, January 1992.

''A Recovery Protocol For Spritely NFS''
Jeffrey C. Mogul.
WRL Technical Note TN-27, April 1992.

''Electrical Evaluation Of The BIPS-0 Package''
Patrick D. Boyle.
WRL Technical Note TN-29, July 1992.

''Transparent Controls for Interactive Graphics''
Joel F. Bartlett.
WRL Technical Note TN-30, July 1992.

''Design Tools for BIPS-0''
Jeremy Dion & Louis Monier.
WRL Technical Note TN-32, December 1992.

''Link-Time Optimization of Address Calculation on a 64-Bit Architecture''
Amitabh Srivastava and David W. Wall.
WRL Technical Note TN-35, June 1993.

''Combining Branch Predictors''
Scott McFarling.
WRL Technical Note TN-36, June 1993.

''Boolean Matching for Full-Custom ECL Gates''
Robert N. Mayo and Herve Touati.
WRL Technical Note TN-37, June 1993.

''Ramonamap - An Example of Graphical Group-
    ware''
Joel F. Bartlett.
WRL Technical Note TN-43, December 1994.

''Circuit and Process Directions for Low-Voltage
    Swing Submicron BiCMOS''
Norman P. Jouppi, Suresh Menon, and Stefanos
    Sidiropoulos.
WRL Technical Note TN-45, March 1994.

''Experience with a Wireless World Wide Web
    Client''
Joel F. Bartlett.
WRL Technical Note TN-46, March 1995.

''I/O Component Characterization for I/O Cache
    Designs''
Kathy J. Richardson.
WRL Technical Note TN-47, April 1995.

''Attribute caches''
Kathy J. Richardson, Michael J. Flynn.
WRL Technical Note TN-48, April 1995.

''Operating Systems Support for Busy Internet Ser-
    vers''
Jeffrey C. Mogul.
WRL Technical Note TN-49, May 1995.