Routing Resource Management for Post-route Optimization

Paul B. Morton
Wayne Dai

UCSC-CRL-02-12
February 19, 2002

Jack Baskin School of Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064 USA

ABSTRACT

In this paper we present a post-route routing resource manager which is based on a routing resource migration strategy. In the past, post-route routing resource management has been handled using rip-up and reroute strategies, however, this approach is not adequate to meet the needs of post-route optimization processes such as crosstalk noise management which require continuity of net adjacency information, and the ability to concentrate dispersed routing resources. Our new approach moves uncommitted routing resources through the routing so that they can be concentrated around specific routing elements while minimizing the disruption to the rest of the routing.

Keywords: routing, wire sizing, post-route optimization, routing resource management

1 Introduction

As IC feature size decrease, parasitic resistance, capacitance and inductance elements are having a significant influence on the performance and reliability of on-chip interconnections. In particular, the effects of parasitic coupling capacitance between adjacent nets are making on-chip interconnections susceptible to cross-talk noise problems. Since the full extent of parasitic coupling capacitance cannot be accurately determine until after the completion of detailed routing, it is very difficult to implement an effective cross-talk noise management strategy in the global and detailed routing phase of the design process. Because of this a significant portion of the cross-talk nose management process must be carried out after the completion of detailed routing.

In the past, post route optimization and repair strategies have been based on "rip-up and reroute" routing resources managers. However, rip-up and reroute leads to unpredictable changes in net adjacencies, and thus coupling capacitance. This makes it very difficult for a rip-up and reroute based cross-talk noise management strategy to converge to a solution. Additionally, while there may be adequate uncommitted routing resources available to resolve a cross-talk noise problem, those resources tend to be dispersed, which makes it difficult for a rip-up and reroute strategy to use them effectively.

In this paper we will develop a more sophisticated approach to post-route routing resource management. Our approach will be based on a routing resource migration strategy. This approach allows us to move uncommitted routing resources through the routing and concentrate them around specific routing elements while minimizing changes in net adjacencies.

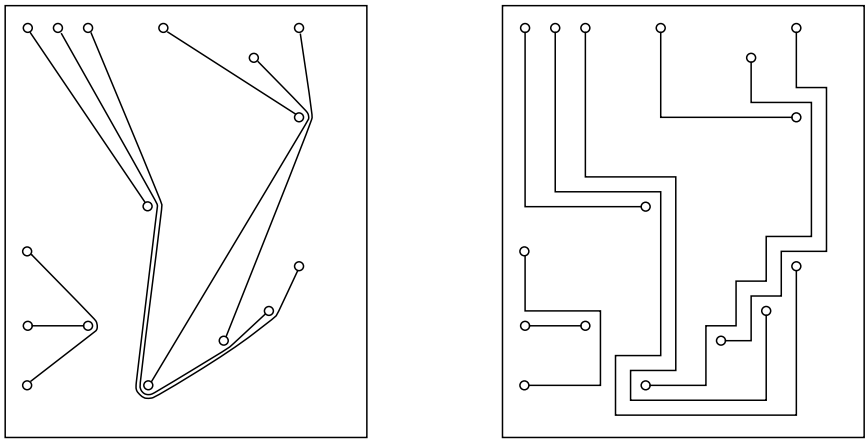

Figure 1: A topological routing (a), and its geometric routing (b).

Since geometric routers are designed to maximize the performance of their geometric path generation abilities, their underlying data structures have been specifically tuned to support this operation and thus are not very efficient at performing the path deformation operations needed for a resource migration strategy. In a topological routing system the routing is represented as the topological abstraction of the geometric routing, as illustrated in Fig. 1, which can be easily transformed back into a geometric routing [1]. Since the topological representation does not need to keep track of all the details associated with an exact geometric representation of the routing, path deformation operations can be

performed very efficiently. Because of this we will base our routing resource migration strategy on a topological routing system.

2 Previous Work

One of the first uses of topological routing for post route resource management was in the area of compaction (see for example [2] and [3]). Compaction is a global optimization process which seeks to minimize the area of a design by migrating uncommitted placement and routing resources to the edges of the design. Another use of topological routing for post route resource management is the area of routing congestion management [4]. In congestion management, uncommitted routing resources are migrated through the routing in a way that evens out the routing congestion across the entire design. While both of these processes use routing resource migration to manage post-route routing resources, they are global process which do not concentrate the routing resources for use by specific routing elements in the design.

In [5], a post route crosstalk noise management strategy was presented which contained a topological post-route routing resource manager. In particular, it allowed for the identification of uncommitted routing resources available to specific nets. After determining how best to use these resources, they were migrated to and concentrated around specific routing elements for the net. The main draw back with this routing resource manager was that it could only identify and migrate "local" routing resources. That is, routing resources that could be migrated without moving any vias or Steiner points.

3 Topological Routing

A topological router represents the routing on each plane of a design using a set of terminals which are connected by a set of branches. Terminals represent vias, pins, and steiner points. Branches represent the wire connections between terminals and behave like flexible rubber bands. Our post-route routing resource manager is primarily based on three basic topological routing operations: branch sizing, terminal move, and incremental design rule checking. The branch sizing operation is used to change the width of a branch. The terminal move operation moves a terminal along a straight line path while maintaining the connectivity of all branches that surround and connect to that terminal, as illustrated in Fig. 2. The incremental design rule checker (DRC) identifies any design rule violations (DRVs) that are created as a result of a change to the topological routing.

In order to locate DRVs in a topological routing, the DRC takes advantage of Maley's routability theorem [6]. Maley's routability theorem states that a topological routing is routable if and only if all of its cuts are safe. A cut is defined as the shortest straight line between two features that are visible to each other, where a feature is any object through which a branch cannot be routed (excluding other branches). A safe cut is one whose flow does not exceed its capacity. A cut's capacity, as illustrated in Fig. 3, is a measure of the amount of routing resources available across the cut, and a cuts flow is a measure of the routing resources needed to route all the branches that need to cross the cut. Thus, in a topological routing, a design rule violation occurs when the flow of a cut exceeds its capacity.

To implement our routing resource manager using these basic operations our algorithms will be based on feed back driven probing methods. The basic idea behind these methods

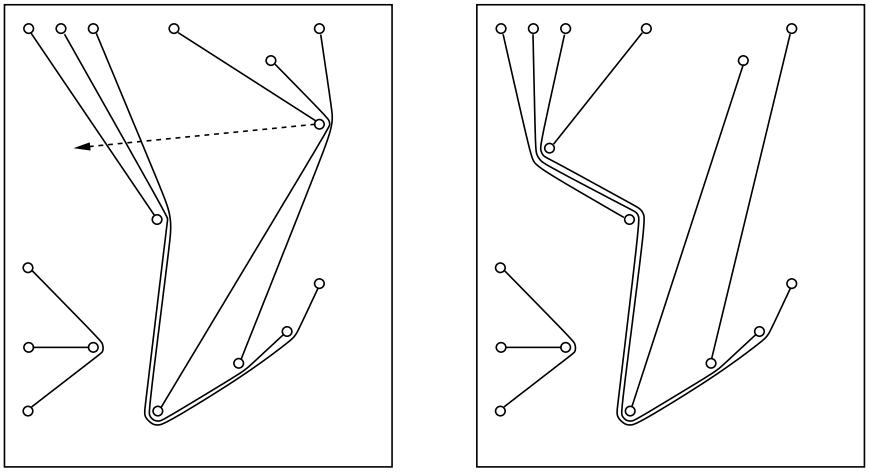

Figure 2: An illustration of the terminal move operation; (a) The original topological routing and a move path (dotted line); (b) The topological routing that results from the terminal being moved along the move path.

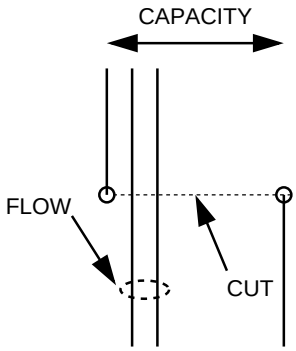

Figure 3: The flow and capacity of a cut between two terminals of a topological routing.

is to perform a probing operation, such as an increase in branch width or a terminal move, followed by a more refined probing operation based on the feed back gained from an analysis of the DRVs generated by the previous probe. This sequence of probes continues until the probing operation is refined to the point where it does not generate any more DRVs. In our implementation of these methods, each successive probing operation is less "intrusive" than the previous probing operation. Because of this characteristic we will refer them as "probe-and-retreat" methods.

4 Resource Identification and Migration

In this paper we are interested in post-route routing resource management for optimization processes, like cross-talk noise management, which work by increasing the amount of routing resources available to a well defined group of routing elements in order to affect a behavior related to those routing elements. These optimization processes can be broken down into four main steps:

- Identify a group of routing elements to be optimized.
- Identify uncommitted routing resources available to those routing elements

- Perform an optimization, constrained by the available routing resources, to determine how best to use the available routing resources to solve the problem associated with the group of routing elements.
- Migrate the routing resources, as determined by the optimization, and assign them to the routing elements.

Of these four steps, the resource identification step (step 2) and the resource migration step (step 4) constitute the post-route routing resource management process.

The objective of the identification step is to quantify the "useful" uncommitted routing resources that are available to a given branch. In this paper, useful routing resources are defined as those routing resources that will allow an increase in the width or spacing of an entire branch. Since routing resources that can be used to increase branch width can also be used to increase branch spacing, we will measure all useful routing resources available to a branch in terms of equivalent branch width.

In most post-route optimization, additional routing resources beyond some reasonable upper bound are of little or no value. Because of this the resource identification process can be terminated when the amount of useful uncommitted routing resources has exceeded this bound. Additionally, since CPU time is also a valuable resource, the process of identifying uncommitted routing resources cannot be open ended. The identification process must incorporate some mechanism to limit the amount of effort that is expended in finding uncommitted routing resources.

The identification process can be decomposed into three steps. The first step is an "open ended" migration process, which is a process that continues to migrate uncommitted routing resources until no more resources can be found. The second step is an accounting process which tabulates all of the useful routing resources that have been accumulated by the first step. The third, and final step, is to perform a "reverse" migration process, which migrates the resources back to their original locations. From this we can see that the identification and migration processes have a significant duplication of effort. To avoid this duplication and incorporate the effort limits, and the upper bound limits on the amount of useful uncommitted routing resources that need to be found, we repartition the identification and migration processes into an adaptive plowing process and a relaxation process.

The adaptive plowing process simultaneously concentrates and quantifies the useful uncommitted routing resources around the set of branches. Once the optimization process determines how much of those resources will be needed, the relaxation process attempts to move the unneeded routing resources back, as close as possible, to their point of origin.

A plowing process is simply another way to look at the resource migration process. in particular, given a branch and a desired width increase, a plowing process "pushes" all the routing adjacent to the branch far enough from the branch to accommodate the desired increase in branch width. Extending this idea, we define an adaptive plowing process as one which attempts to push all the adjacent routing far enough from the branch to accommodate the desired increase in branch width, but if, due to effort constraints, or insufficient routing resources, it is unable to achieve this goal, then it determines the increase in branch width that can actually be accommodated.

4.1 Adaptive Plowing

Our approach to adaptive plowing is to set the width of a branch to a reasonable upper bound, then determine if this width increase creates any DRVs. If it does, then we attempt

```
PLOW_BRANCHES(BranchSet, WidthBound)
 1. FOR EACH Branch ∈ BranchSet
 2.     SavedWidth ← GET_WIDTH(Branch)
 3.     DesiredWidth ← WidthBound
 4.     REPEAT
 5.         SET_WIDTH(Branch, DesiredWidth)
 6.         CHECK FOR DRVs
 7.         IF DRVs FOUND
 8.             GET Cut OF FIRST DRV
 9.             SET_WIDTH(Branch, SavedWidth)
10.             DesiredWidth ← EXPAND_CUT(Cut, Branch, DesiredWidth)
11.         END IF
12.     UNTIL NO DRVs FOUND
13. END FOR
```

Figure 4: Pseudo code for plowing a set of branches.

to fix these DRVs. If there are DRVs that we cannot fix, then we reduce the width of the branch until the DRVs are eliminated.

Our approach to adaptive plowing is outlined in the pseudo code in Fig. 4. The input to this procedure is the set of branches to be plowed, and the upper bound on the routing resources to concentrate around each branch. We plow the the set of branches one branch at a time. To plow each branch we first save its current width (line 2), then set the branch width to the width bound (line 5). We then check for and attempt to fix the first of any DRVs created as a consequence of increasing the branch width (lines 6 to 11). To fix a DRV we attempt to expand the capacity of the failing cut associated with the DRV (line 10). If we are unable to expand the cut's capacity enough to accommodate the increased branch width, we reduce the width bound for the branch accordingly (line 10). In order to simplify the cut expansion process, we first reset the width of the branch back to its original size (line 9). By resetting the branch width, the cut expansion process does not have to contend with a set of preexisting DRVs. Once the cut expansion has been completed, we repeat the process of branch sizing (line 5), DRV checking (line 6) and cut expansion (lines 7 to 11) until no more DRVs are created by the change in branch width.

Since it may not be possible to fix the DRV within the specified effort and uncommitted routing resource constraints, we expand the cut's capacity as much as resources allow and then reduce the branch's maximum width accordingly.

4.2 Cut expansion

To expand the capacity of a cut we attempt to move the two terminals that define the end points of the cut in opposite directions. Our approach is outlined in the pseudo code in Fig. 5. The inputs to this function are the cut to be expanded, the branch causing the expansion, and the desired width of the branch. After expanding the cut, the function returns the width of the branch that can be accommodated by the expanded cut.

To expand the capacity of a cut we first need to identify the two terminals that define the ends of the cut (line 1), then compute the increase in the size of the cut necessary to accommodate the desired width of the branch (line 2), and then determine what direction

$EXPAND\_CUT(Cut, Branch, DesiredWidth)$
1. $MoveDist \leftarrow COMP\_MOVE\_DIST(Cut, Branch, DsiredWidth)$
2. $[Term1, Term2] \leftarrow GET\_CUT\_TERMS(Cut)$
3. $MoveDir \leftarrow COMP\_MOVE\_DIR(Term1, Term2)$
4. $DistMoved \leftarrow PLOW\_TERM(Term1, MoveDist, MoveDir)$
5. $RemainingDist \leftarrow MoveDist - DistMoved$
6. IF $RemainingDist > 0$
7. $\quad DistMoved \leftarrow PLOW\_TERM(Term2, RemainingDist, -MoveDir)$
8. $\quad RemainingDist \leftarrow RemainingDist - DistMoved$
9. END IF
10. $NewWidth \leftarrow COMP\_NEW\_WIDTH(Cut, Branch, RemainingDist)$
11. RETURN $NewWidth$

Figure 5: Pseudo code for expanding a cut to accommodate the desired width of a branch.

to move each of the terminals (line 3). Since we are interested in rectilinear routing, the minimum amount of terminal movement to achieve the desired increase in capacity is achieved by either a horizontal or vertical movement. To see this, recall that the capacity of a cut for rectilinear routing is measured as $MAX\{\Delta x, \Delta y\}$, where Δx and Δy are the horizontal and vertical separations, respectively, between the two end points of the cut. Because of this we only consider horizontal or vertical terminal movements. We then attempt to move one of the terminals the entire distance necessary to accommodate the increase in branch width (line 4). Note that this terminal move operation is a plowing operation which will move other terminals in order to avoid creating other DRVs. If, due to routing resource or effort constraints, it is not possible to move the terminal the entire distance, then we attempt to move the cut's other terminal the remaining distance in the opposite direction (line 8). We then compute the width of the branch that can be accommodated by the newly expanded cut (line 10).

4.3 Terminal plowing

We implement terminal plowing using a recursive probe-and-retreat algorithm. We use a series of probe-and-retreat steps to move the terminal as close as possible to the desired destination. If we did not reach this destination, then we identify the terminal that is blocking further movement, and recursively attempt to plow this terminal out of the way. We then repeat the process of probe-and-retreat followed by recursive plowing until we either reach the desired destination or we exhaust the available uncommitted routing resources, or we reach the limit on the amount of effort we want to expend on plowing a single terminal.

Our terminal plowing algorithm is outlined in the pseudo code in Fig. 6. The inputs to this function are the terminal to be plowed, the distance and direction to plow it, and a limit, in the form of a recursion depth limit, on the amount of effort to expend plowing this terminal. The function returns the actual distance that the terminal was moved. We first determine the move path for the terminal, as determined by the starting location and desired destination location for the terminal (lines 1 and 2). We then move the terminal forward along this path until it reaches the desired destination, or its movement is blocked by an adjacent terminal. If its movement is blocked, we recursively attempt to plow the blocking terminal, parallel to the original move path, a distance that will allow for the completion

```
PLOW_TERM(Term, Dist, Dir, Depth)
 1. CurrentLoc ← StartLoc ← LOCATION(Term)
 2. DestLoc ← COMP_DEST(StartLoc, Dist, Dir)
 3. WHILE CurrentLoc ≠ DestLoc AND Depth > 0
 4.     CurrentLoc ← DestLoc
 5.     REPEAT
 6.         MOVE_TERM(Term, CurrentLoc)
 7.         CHECK FOR DRVs
 8.         IF DRVs FOUND
 9.             GET Cut OF FIRST DRV
10.             [CurrentLoc, BlockingTerm] ← RETREAT(Term, Cut, StartLoc)
11.         END IF
12.     UNTIL NO DRVs FOUND
13.     RemainingDist ← ||DestLoc − CurrentLoc||
14.     IF RemainingDist > 0
15.         DistMoved ← PLOW_TERM(BlockingTerm, RemainingDist, Dir, Depth − 1)
16.         DestLoc ← NEW_DEST(CurrentLoc, DistMoved, Dir)
17.     END IF
18. END WHILE
19. DistMoved ← ||CurrentLoc − StartLoc||
20. RETURN DistMoved
```

Figure 6: Pseudo code for plowing a terminal.

of the plowing operation on the original terminal (lines 13 to 15). If the blocking terminal cannot be moved far enough to allow the original terminal to reach the desired destination, then we determine a new destination for the original terminal based on the final location of the blocking terminal (line 16). The plowing operation is terminated when the current location of the original terminal and the destination location of the move path converge (line 3).

The key to this algorithm is to be able to identify the blocking terminals in the proper order. In particular, we are interested in finding the blocking terminal that is closest to the starting point on the move path, since blocking terminals farther along the path may be plowed out of the way by recursive plowing operations on blocking terminals closer to the start of the path. To identify the closest blocking terminal we use a probe-and-retreat method. That is, we advance the original terminal to the desired destination location (lines 4 and 6), then we check for DRVs (line 7). If there are any, we select the first one and determine how far back along the move path to retreat in order to avoid this DRV (lines 8 to 11), then move the original terminal back to this location (line 6). We then repeat the DRC and retreat process until no more DRVs are found. The cut associated with the last DRV found, along with the starting location of the move path, and the current location of the original terminal are used to determine the closest blocking terminal (line 10).

In order to limit the amount of computing resources consumed by the plowing process, we set a recursion depth limit. With out this depth limit the recursive plowing process would only terminate when it becomes hemmed in by a "fence" of immovable terminals, such as those associated with pins or the edge of the design. Once the recursion depth limit has been reached, all terminals encountered at that level of recursion are considered to be

```
PLOW_RESTRICTED_TERM(Term, Dist, Dir)
 1. StartLoc ← LOCATION(Term)
 2. MoveDist ← Dist
 3. GET List OF TERMINALS RESTRICTING Term
 4. FOR EACH RestTerm ∈ List
 5.     DestLoc ← COMP_DEST(StartLoc, MoveDist, Dir)
 6.     DistRest ← DIST_REST(Term, RestTerm)
 7.     IF MIN_DIST(RestTerm, PATH(StartLoc, DestLoc)) < DistRest
 8.         Loc ← REST_LOC(RestTerm, PATH(StartLoc, DestLoc), DistRest)
 9.         RemainingDist ← MoveDist − ‖Loc − StartLoc‖
10.         DistMoved ← PLOW_TERM(RestTerm, RemainingDist, Dir)
11.         MoveDist ← MoveDist − (RemainingDist − DistMoved)
12.     END IF
13. END FOR
14. RETURN MoveDist
```

Figure 7: Pseudo code for plowing restricted terminals.

immovable, which forces the plowing process to terminate.

4.4 Restricted terminal movement

Recall from section 4.1 that we reset the branch width to its original size (line 9 in Fig. 4) so that the cut expansion process does not have to contend with a set of preexisting DRVs. Because of this, once a cut has been expanded, we need a mechanism to maintain the extra capacity throughout the remainder of the plowing process for the corresponding branch. The mechanism we use is to maintain a table of terminals whose movement we want to restrict. That is, the terminals in the table must maintain a minimum distance between themselves and at least one other restricted terminal. Each entry in this table can be easily constructed by $EXPAND\_CUT$.

To maintain the required minimum distances between restricted terminals, we need to add a preprocessing step to $PLOW\_TERM$. Our approach to this preprocessing step is outlined in the pseudo code in Fig. 7. The inputs to this function are the terminal to be plowed, and the distance and direction it should be plowed. The function returns the actual distance that the terminal was moved. $PLOW\_RESTRICTED\_TERM$ checks to see if the terminal we are about to plow has been restricted. If the terminal has been restricted, we attempt to move all terminals restricting its movement such that the movement of the restricted terminal will not reduce the distance, between its self and the restricting terminals, below the required minimum distances. If a restricting terminal cannot be moved far enough to guarantee this minimum distance, then the move distance of the restricted term is reduced accordingly.

We begin by determining the starting location of the terminal's move path (line 1). We then get a list of terminals that are restricting the movement of the restricted terminal (line 3). For each of the restricting terminals we determine if the movement of the restricted terminal, along its move path, could violate the minim distance requirement. If the minimum distance requirement can be violated, then we attempt to move the restricting terminal out of the way, or if this is not possible, shorten the move path accordingly (lines 4 to 13). To

$PLOW\_IN\_LINE\_TERM(Term, Dist, Dir)$
1. $StartLoc \leftarrow LOCATION(Term)$
2. $DestLoc \leftarrow COMP\_DEST(StartLoc, Dist, Dir)$
3. WHILE TERMINALS BLOCKING $PATH(StartLoc, DestLoc)$
4. $\quad InLineTerm \leftarrow NEAREST\_TERM(StartLoc, DestLoc)$
5. $\quad InLineLoc \leftarrow LOCATION(InLineTerm)$
6. $\quad RemainingDist \leftarrow \|DestLoc - InLineLoc\| + \delta$
7. $\quad DistMoved \leftarrow PLOW\_TERM(InLineTerm, RemainingDist, Dir)$
8. $\quad DistMoved \leftarrow DistMoved - \delta$
9. $\quad DestLoc \leftarrow COMP\_DEST(InLineLoc, DistMoved, Dir)$
10. END WHILE
11. $NewMoveDist \leftarrow \|DestLoc - StartLoc\|$
12. RETURN $NewMoveDist$

Figure 8: Pseudo code for plowing in-line terminals.

determine if the distance between the restricted terminal and a restricting terminal could be less than the minimum required distance, we first determine the destination of the current move path (line 5), then we get the minimum required distance between the terminals (line 6), then we check to see if the minimum distance between the restricting terminal and any point on the move path is less than the minimum required distance between the restricted and restricting terminals (line 7). If there are any points on the move path that are too close to the restricting terminal, we determine the location on the move path that is closest to the starting point of the move path and which still maintains the required minimum distance from the restricting terminal (line 8). From this we determine how far to move the restricting terminal, on a path parallel to the move path, such that the original terminal move can be completed without compromising the minimum required distance (line 9). We then attempt to move the restricting terminal (line 10), and if the terminal cannot be moved far enough, we adjust the move path accordingly (line 11).

4.5 In line plowing

Since terminals are not flexible objects, one terminal cannot be moved through another. Because of this, line 6 of Fig. 6 cannot be reliably executed until we are certain that the move path does not intersect any blocking terminals. We will refer to any terminal intersected by the move path as an "in-line terminal". If the move path contains any in-line terminals, these terminals must first be plowed out of the way, or if they cannot be plowed out of the way, the move path must be shortened. We accomplish this by adding another preprocessing step to the beginning of $PLOW\_TERM$.

Our approach to in-line plowing is outlined in the pseudo code in Fig. 8. The inputs to this function are the terminal to be plowed, and the distance and direction it should be plowed. The function returns the actual distance that the terminal was moved. We first determine the move path of the original terminal plowing operation (lines 1 and 2). We then check to see if there are any in-line terminals blocking the move path (line 3). If there are, we select the one closest to the start of the move path (line 4 and 5), determine how far it needs to be moved to get it off of the move path (line 6), then attempt to plow it that distance in the direction of the move path (line 7). Note that in order to move the in-line

```
RELAX_TERMS(MoveStack)
 1. WHILE MoveStack NOT EMPTY
 2.     Move ← POP(MoveStack)
 3.     Term ← GET_TERM(Move)
 4.     DestLoc ← GET_ORIG_LOC(Move)
 5.     StartLoc ← LOCATION(Term)
 6.     IF TERMINAL BLOCKING PATH(StartLoc, DestLoc)
 7.         InLineTerm ← NEAREST_TERM(StartLoc, DestLoc)
 8.         DestLoc ← LOCATION(InLineTerm)
 9.         DestLoc ← DestLoc − δ DIRECTION(StartLoc, DestLoc)
10.     END IF
11.     REPEAT
12.         MOVE_TERM(Term, DestLoc)
13.         CHECK FOR DRVs
14.         IF DRVs FOUND
15.             GET Cut OF FIRST DRV
16.             DestLoc ← RETREAT(Term, Cut, StartLoc)
17.         END IF
18.     UNTIL NO DRVs FOUND
19. END WHILE
```

Figure 9: Pseudo code for relaxing a sequence of terminal moves.

terminal off of the move path we need to include an incremental distance, δ, in line 6, since without it the in-line terminal would end up at the destination location on the move path. If the plowing operation on the in-line terminal could not move it far enough to get it off of the move path, then we shorten the move path by moving its destination back towards its starting location (lines 7 to 9).

It should be noted that the loop in lines 3 to 10 can be repeated at most n times, where n is the number of layers that the original terminal spans. This can be seen by the fact that line 4 selects the in-line terminal that is closest to the start of the move path, and the plowing operation on line 7 will recursively plow any other in-line terminals farther down the move path which have a layer in common with this in-line terminal.

4.6 Relaxation

In order to relax the terminal moves, we need to build a stack of terminal moves. This is easily accomplished by adding a *PUSH* operation to the end of *PLOW_TERM*. Each element of this stack contains the terminal that was moved, and its original location. The relaxation process attempts to move each terminal in the stack back, as close as possible, to its original location. This is accomplished by defining a move path starting at the terminals current location and ending at the terminals original location. We then move the terminal to a point on this move path which is as close as possible to its original location and which does not create any DRVs.

Our approach to relaxing the terminal moves is outlined in the pseudo code in Fig. 9. The input to this procedure is the stack of terminal moves that was created by *PLOW_TERM*. We *POP* a move from the *MoveStack* from which we identify the terminal and construct

a move path (lines 2 to 5). We then check to see if there are any in-line terminals on the move path and adjust the destination of the move path accordingly (lines 6 to 10). We then use a probe-and-retreat strategy, similar to that used in $PLOW\_TERM$, to determine the location on the move path that is closest to the destination and does not generate any DRVs (lines 11 to 17).

5 Results

To demonstrate our new approach to post-route routing resource management we have implemented it using the SURF topological routing system [7]. We then used this implementation to significantly increase the width of a branch in a routed three level metal chip design. Fig. 10 shows a portion of the geometric routing for layers two and three of this design. In particular, the highlighted metal three wire is a section of the branch whose width we wish to increase. Fig. 11 shows the topological routing which corresponds to this geometric routing. Note that Fig. 11 is an "expanded" topological routing which, unlike the topological routing in Fig. 1 and Fig. 2, takes into account the width and spacing of the routing when determining the precise shape of a branch.

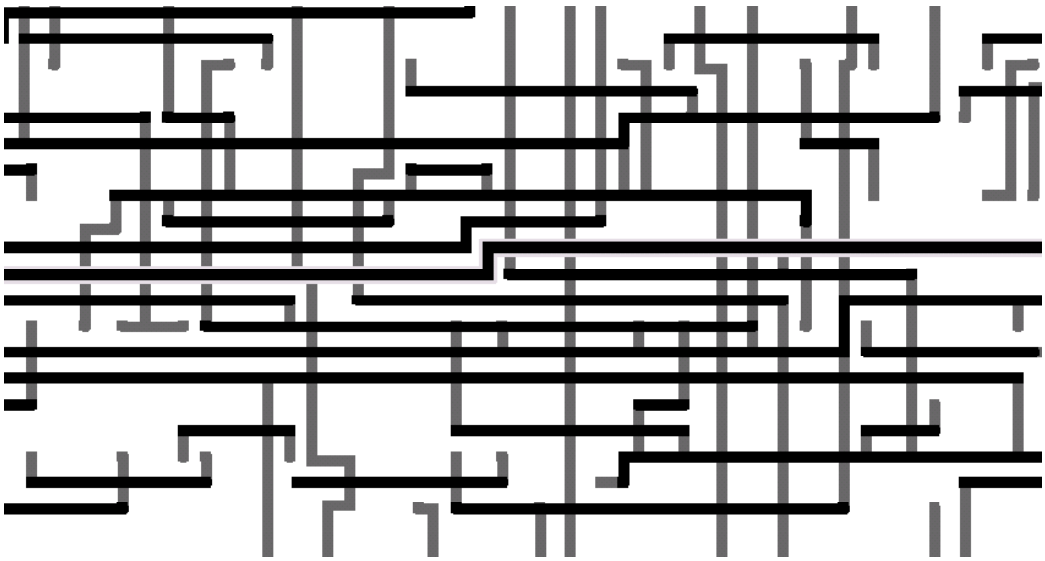

Figure 10: Original geometric routing.

Fig. 12 shows the topological routing that results from an adaptive plowing operation which seeks to accumulate enough routing resources to increase the width of the highlighted branch by a factor of 10, with a recursion depth limit of 5 terminals. Because of the recursion depth limit, the plowing operation was only able to accumulate enough routing resources to increase the branch width by a factor of 5, as shown in Fig. 13.

Comparing Fig. 11 to Fig. 12 we see that terminals 1, 2, 3 and 4 required a terminal plowing operation that needed 3 levels of recursion, that is, before terminal 1 can be moved, terminals 2, 3, and 4 must be moved. Terminals 5 and 6 required a plowing operation with only one level of recursion. Note that while the movement of the terminals is primarily in the vertical direction, as we would expect given the horizontal orientation of the branch, there are some terminals, such as terminal 7, which have been moved horizontally. This is due to the fact that the associated cut that was being expanded had a strong horizontal orientation and thus terminals 7 required a shorter horizontal move than a vertical move to

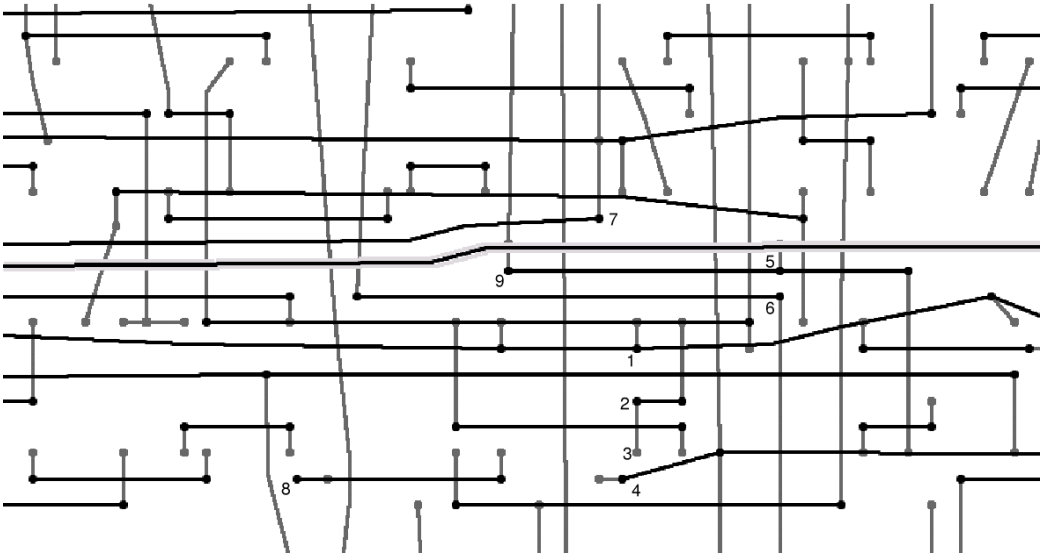

Figure 11: Original topological routing.

achieve the desired cut capacity. Finally, note that terminal 8 seems to have been moved unnecessarily, and terminal 9 seems to have been moved farther than necessary. This is due to the fact that the first terminal moves in the plowing process are attempting to to move the terminals so that the wire width can be increased by a factor of 10, however, as the plowing process proceeds, the recursion limit and limits on uncommitted routing resources reduce the upper bound on the width increase to a factor of 5. Since the final width bound is half the original width bound, some of the first terminal moves appear to be unnecessary or unnecessarily large.

Fig. 14 shows the topological routing that results when we triple the width of the original branch routing and then relax the terminals that were moved by the plowing operation. Comparing Fig. 12 and Fig. 14 we see that the unnecessary move of terminal 8 and the exaggerated move of terminal 9 have been eliminated while only half the movement of terminals 5 and 6 have been eliminated. Fig. 15 is the resulting geometric routing.

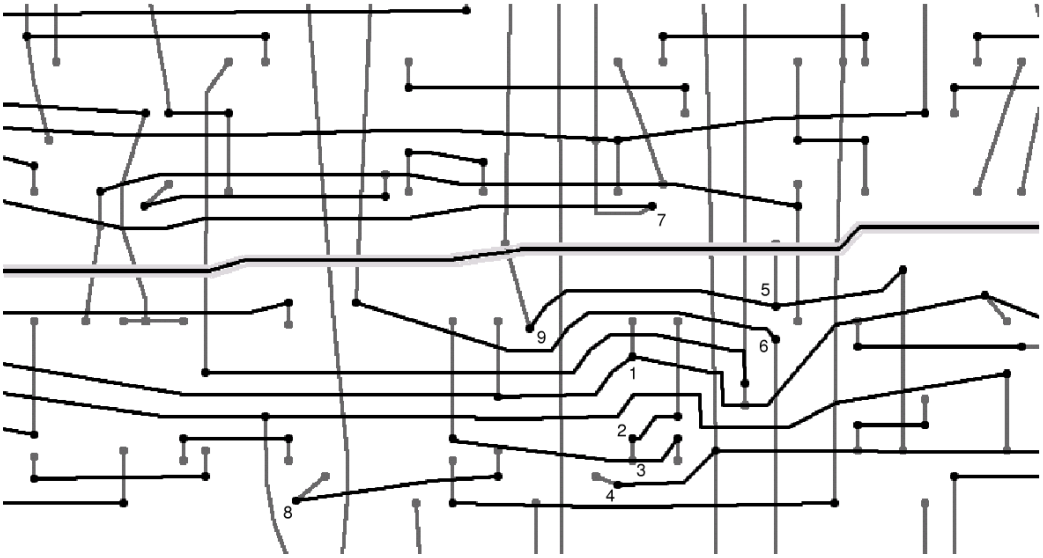

Figure 12: Topological routing after adaptive plowing.

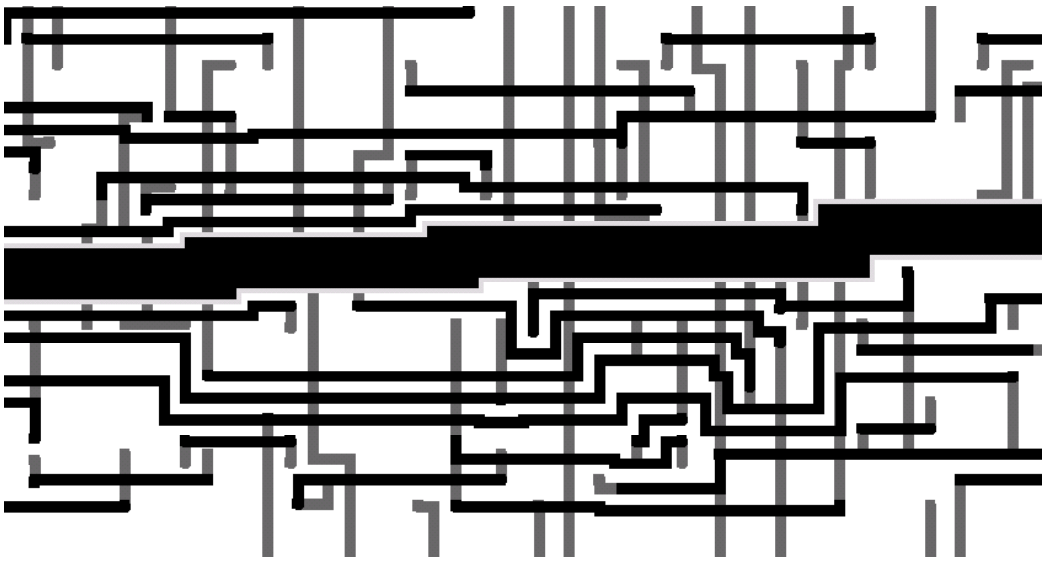

Figure 13: Geometric routing after adaptive plowing.

Comparing Fig. 10 and Fig. 15 we see that we have been able to concentrate enough routing resources around the branch to triple its width while minimizing the disruption to the adjacent routing.

6 Conclusion

In this paper we have presented a post-route routing resource manager which is based on a routing resource migration strategy. Since geometric routers are not well suited for performing the path deformation operations needed for a resource migration strategy we have based our work on a topological router. Our post-route routing resource manager is partitioned into two phases, an adaptive plowing phase, which concentrates and quantifies useful uncommitted routing resources around a set of routing elements, and a relaxation phase, which moves any routing resources that are not needed by the routing elements back, as close as possible, to their point of origin. Our plowing and relaxation processes are based on three basic topological routing operation: branch sizing, terminal move, and incremental design rule checking. The plowing and relaxation processes are constructed from these basic operations using probe-and-retreat algorithms. Our new approach to post-route routing resource management allows us moves uncommitted routing resources through the routing so that they can be concentrated around specific routing elements while minimizing the disruption to the rest of the routing.

References

[1] D. J. Staepelaere, "Geometric transformation for a rubber-band sketch," Master's thesis, University of California Santa Cruz, Septemeber 1991.

[2] N. Sherwani, *Algorithms for VLSI Physical Design Automation.* Boston, Mass: Kluwer Academic Publishers, 1995.

[3] J. Valainis, S. Kaptanoglu, E. Liu, and R. Suaya, "Two-Dimensional IC Layout Compation Based on Topological Design Rule Checking," *IEEE Trans. Computer-Aided Design*, vol. 9, no. 3, pp. 260–275, 1990.

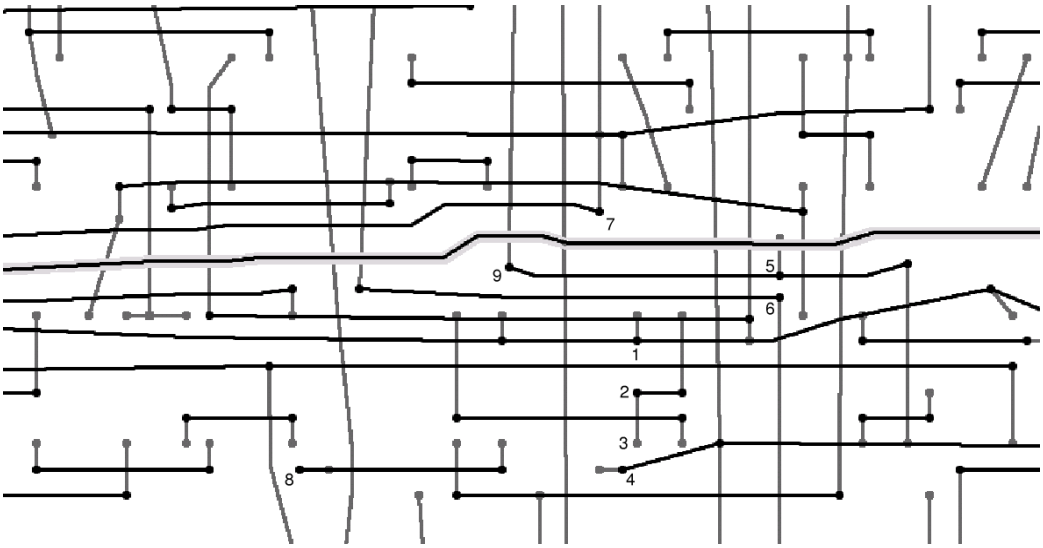

Figure 14: Relaxed topological routing.

[4] J. Su and W. Dai, "Post-Route Optimization for Improved Yield Using a Rubber-Band Wiring Model," in *Proc. IEEE/ACM Int. Conf. on Computer Aided Design*, pp. 700–706, 1997.

[5] P. Morton and W. Dai, "A Routing Optimizer for Cross-Talk Noise Avoidance in Resistive VLSI Interconnections," Tech. Rep. UCSC-CRL-00-03, UCSC, 2000.

[6] F. Maley, *Single-Layer Wire Routing and Compaction.* Cambridge, Mass: The MIT Press, 1989.

[7] D. J. Staepelaere, J. Jue, T. Dayan, and W. W.-M. Dai, "Surf: a rubber-band routing system for multichip modules," in *Proc. IEEE Design and Test of Computers*, 1993.

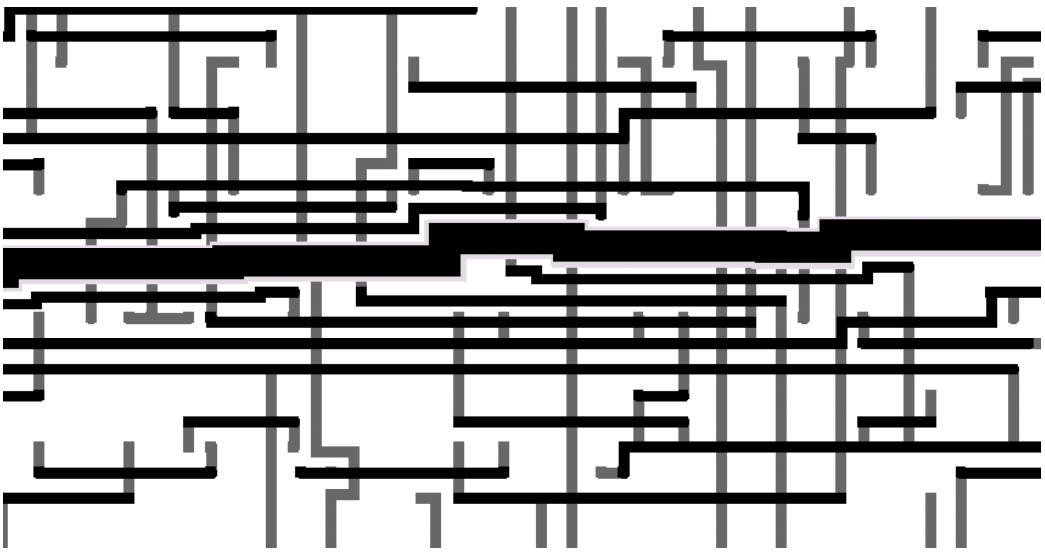

Figure 15: Relaxed geometric routing.