Layout Optimization by Pattern Modification

Ramin Hojati

Cadence Design Systems, Inc.
555 River Oaks Parkway
San Jose, California 95134

Abstract

This paper introduces a new and practical approach to several layout optimization problems. A novel two-dimensional pattern generator, in connection with a set of routing and placement transformations, is employed to efficiently solve problems ranging from Wire Crossing Minimization and Topological Via Minimization to Minimum Steiner Tree Optimization and IO Alignment. The expected running time is O(nlogn) and the space requirement is O(n), where n is the number of layout objects. The system is fully coded and tested, and excellent results in both laboratory and real-life examples have been achieved.

Introduction

Many times as one nears completion of a layout, various optimizations can still be performed. We categorize some of the most common layout optimization problems as follows:

■ **Wire Crossing Minimization.** This problem has two facets: first, the global wire ordering problem, which is rearranging junction terminals between routing regions in an optimal manner [2]; second, minimizing wire crossing inside the routing areas.

■ **Topological (Unconstrained) Via Minimization.** This problem concerns producing a layout with the primary cost function equal to the number of vias [10]. In reality, this is usually not the cost function of choice. In systems with this cost function, sometimes there is an explosion in wire length and area. This explosion occurs because detours around vias are taken to use fewer vias [8].

■ **Minimum Steiner Tree (MST).** Non-optimal MSTs (that is, MSTs with extra twistings and corners) may be observed in the layout [6]. We set two criteria for deciding between two Steiner trees. The first one is wire length, and the second one is number of corners. (Corners decrease the manufacturability of a chip.) Wire length takes precedence in our formulation; that is, if one Steiner tree has less length and more corners, it is considered better than one that has more wire length and fewer corners. We call the first problem Wire Length Minimization and the second one Corner Minimization. By solving the above two problems simultaneously, we present a heuristic for the Minimum Steiner Tree problem.

■ **IO Alignment.** In many designs, the position of top-level IOs (terminals and IO pads) is flexible. Given the side to which a top-level IO belongs, generating a solution with the minimum number of crossings, wire length, vias, and corners is called the IO alignment problem. The precedence of optimization criteria is as given in the previous sentence. Optimizing the above criteria often helps to reduce the chip area.

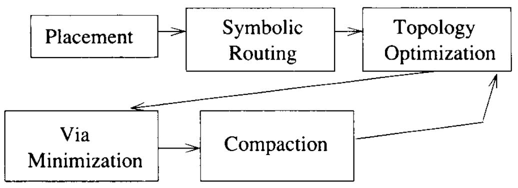
Topology Optimization gives reasonable and user-controlled solutions for these four optimization problems. It works in a symbolic environment and requires a compactor [5]. The tool makes very few assumptions. However, to achieve maximum quality, it needs a router that honors a direction for each routing layer [2], a re-layering routine, and a compactor with automatic jog insertion capability.

Methodologically, Topology Optimization solves the four optimization problems as follows. The working unit of the program is a pattern (a small, connected piece of routing), which we define precisely later. The patterns are big enough so that their number is manageable (in our case, linear), but they are also small enough so that almost every point in the solution space is reachable. (The solution space of the program is the set of all possible topologies.) To reduce wire crossings, a pattern is taken and the number of wire crossings is minimized for that pattern. To build better Steiner trees, patterns are optimized with respect to wire length and number of corners. To solve the IO alignment problem, routings around IOs are taken as inputs and are optimized with respect to wire crossing and twisting. To solve the Unconstrained Via Minimization problem, the number of wire crossings is reduced, and then a standard via minimizer [9] is called to minimize vias.

Overview of Topology Optimization

Topology Optimization lets the user perform some or all of the above optimizations. The input can come from an automatic place and route system or can be manually created; but it has to be in symbolic form. The tool can also be run on selected areas or partial layouts.

A typical design flow is shown in Figure 1. After placement, the symbolic router is run. Topology Optimization then optimizes the layout topologically. A re-layering routine then minimizes the number of vias. Finally, a compactor with automatic jog insertion capability produces compact and design-rule-correct solutions.



Typical Design Flow
Figure 1

After symbolic routing, the user might choose to iterate a few times through the topology optimization, via minimization, and compaction to achieve better results. At all times, manual changes are accepted. We refer the reader to [4] for a discussion of our optimization environment. In this paper, we concentrate on Topology Optimization's internal structure.

Topology Optimization can perform one or more of the four transformations—wire crossing minimization, corner minimization, wire length minimization, and IO alignment—simultaneously. Each transformation is controlled by a variable, which takes discrete values between 0 and 5. If the user specifies 0, no optimization of that kind happens. If the user specifies 5, the program iterates until the slope of improvement is very small or 5 iterations are performed. The number 5 is chosen because, in our experience, the optimization process converges after 2 or 3 iterations. By providing these variables explicitly, the user can control time versus quality trade-offs. The high-level algorithm follows.

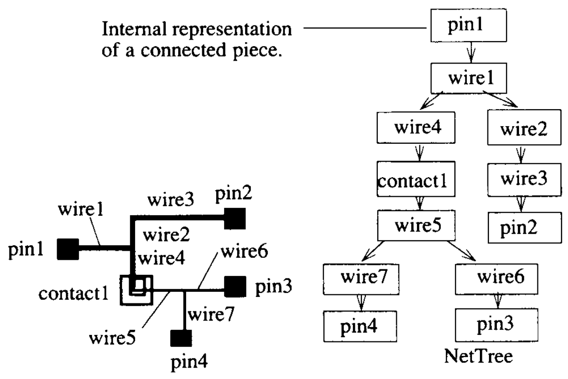
Topology Optimization Algorithm

```
1) Get input and build data structures.
2) For each net
      repeat
            - Generate a pattern.
            - Call the appropriate moving transformations.
            - Adjust the internal state if a change occurs.
      until no more patterns can be generated.
```

During the first step, data is read in and data structures are built. Then, all the connected pieces are extracted and are represented as trees. This second step is known as net tree extraction. The nets are sorted according to the number of terminals and wire length. We have experimented with several net selection strategies. The best strategy is going from bigger to smaller nets, which resolves net dependencies usually in only two or three iterations. After we present the pattern matcher and the moving transformations, we will analyze this algorithm for complexity.

Pattern Matcher

The pattern matcher is the heart of Topology Optimization. It takes a net tree as input (Figure 2) and produces patterns one at a time. It calls the appropriate moving transformation on each pattern. If a move happens, the pattern matcher's state is re-adjusted, and pattern generation continues. The pattern matcher assumes that the first element is a terminal and there is only one wire connected to it.



Nets Represented Internally as Trees

Figure 2

Definitions

1) A connectivity change is a transformation that changes connectivity information.

2) A topological change is a transformation that does not change connectivity information.

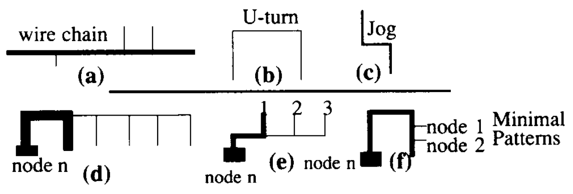
3) A wire chain is a sorted set of connected wires, where all the wires have the same direction but can be on different layers (Figure 3a).

4) A jog is a pattern composed of three connected wire chains. The first and third wire chains are parallel to each other and are perpendicular to the second one. Moreover, the first and third wire chains are on two opposite sides of the second wire chain (Figure 3c). There are four kinds of jogs depending on the orientation.

5) A U-turn is the same as a jog, except that the first and the third wire chains are on the same side of the second chain (Figure 3b).

6) A useful pattern is either a jog or a U-turn.

7) A Minimal Useful Pattern (MUP) of a kind from a (net tree) node is the smallest pattern of that kind visible from the node. There are eight kinds of patterns since there are two useful patterns and each can have four orientations. Note that a minimal useful pattern is a function of two variables, node and kind of pattern. There might be many choices. However, only the "smallest" ones are considered. By smallest, we mean the pattern with the smallest second wire chain (Figures 3d and 3e). However, for the last wire chain, the maximal wire chain is taken (Figure 3f).



Only the thicker patterns are minimal among all the useful patterns seen from the node n.

Pattern Terminology

Figure 3

8) The Set of Minimal Useful Patterns (SMUP) of a connected piece is the set of all minimal useful patterns visible from all the nodes, subtracted from the set of all the patterns that are not minimal from a node. The definition of minimal useful pattern is chosen carefully so that an efficient algorithm for recognizing the SMUP can be devised.

The example in Figure 3e clarifies the definition of SMUP. If we look from nodes 2 and 3, we see two minimal jogs. These are not in the SMUP of the connected piece since they are non-minimal from node n. Figure 3f shows why we include the condition of taking the maximal last wire chain in the definition of MUP. The reason is we do not want to generate U-turns from nodes 1 and 2 since they are non-minimal from node n.

To show the difference between one-dimensional matching and two-dimensional matching, let us concentrate on Figure 2. Assume we start the search at pin1 and visit wire1, wire2, wire3, and pin2. As we stand at pin2, we remember all the pieces we have seen so far. Now, we backtrack to the junction of wire1 and wire2. To recognize all the MUPs from pin3 and pin4, we need to know about wire2 and wire3. Therefore, in two-dimensional matching we not only need to know of things we have seen before, but also we need to know about things that are ahead of us.

To cope with the above problem, we extend the notion of a finite state automata as follows. In a finite state automata, a state contains a number. A two-dimensional state is a set of four lists, corresponding to four directions: north, south, west, and east. Each list contains a set of states (the states of one-dimensional matching). Repetition of states is not allowed in the state lists.

To define a finite automata, we need to define five parameters [1]. The set of states Q is the set of all two-dimensional states, which we show is still finite (Lemma 1). The input alphabet contains two symbols: horizontal and vertical wires. The transition function is a function of previous state, input symbol, and moving direction. The transition function produces a list of states. The set of final two-dimensional states F contains those two-dimensional states with at least one final state. In our case, if at least one state in any of the state lists corresponds to a useful pattern, the state is final. The initial state is q0.

The pattern matching algorithm follows.

PatternMatch Algorithm

```
MatchNet(netTree)      /** Sets up the environment for MatchLine . **/
        1) Make the initial state "state0."
        2) Get the child node.
        3) MatchLine(childNode, state0).
MatchLine(netTreeNode, lastState)      /** main matching routine **/
        0) Make a state, "currentState."
        1) Update the list in direction of lastState.
        2) If there is a final state and none of the children
            has the same direction as the current one, call the
            moving routines. Adjust internal states if a move occurs.
        3)
            3.0) lastChildNode = NULL.
            3.1) For all the child nodes that are wires
                ("curChildNode")
                    if last ChildWire is not NULL
                            propState(currentState, lastDir, 2).
                    MatchLine(curChildNode, currentState).
                    lastChildNode = curChildNode.
PropState(state, dir, propLevel)  /** This routine looks back
            propLevel in the given direction to update state list
            of the state.  **/
        1) Go back propLevel in the given direction.
        2) Get all the unvisited nodes in a sorted manner.
        3) Expand states one after another until hitting the
            given state.
```

The subroutine MatchNet sets up the pattern matching. It creates the initial state and calls the pattern matcher on the first node's child (assuming there is only one child). Note that here we are only interested in wires. The algorithm MatchLine walks through a net tree until it hits a leaf of the tree. Then it backtracks to a branch-point. That is when it calls the routine PropState to get information about neighboring nodes. Since the length of the useful pattern is 3, we only need to look 2 steps back. We will show that each node is visited by PropState only once. Thus, this operation takes total linear time. If a final state is found, a routine is called that returns the MUP corresponding to that state, on which the appropriate moving transformations are called.

Because of space limitations, all proofs for the following lemmas and theorems have been omitted.

Lemma 1: The maximum number of states in a two-dimensional state recognizing useful patterns is 16; that is, a constant.

Theorem 1: The PatternMatch algorithm recognizes the set of all minimal useful patterns.

Lemma 2: The total cost of generating minimal useful patterns after a final state is recognized is linear. In other words, the amortized cost of generating an MUP is constant.

Lemma 3: The total cost of adjusting the pattern matcher's state in the absence of connectivity changes is linear. In other words, in the absence of connectivity changes, the amortized cost of adjusting a two-dimensional state is constant.

Lemma 4: The total cost for PropState is linear over the running time of the algorithm on a connected piece; that is, constant in an amortized sense.

Theorem 2: If there are no connectivity changes, the PatternMatch algorithm runs in linear time and space.

Corollary 1: The PatternMatch algorithm generates all the minimal useful patterns in linear time and space. Moreover, the set of minimal useful patterns is linear.

Proof: Follows directly from Theorems 1 and 2

There is one question unanswered: What happens when a connectivity change occurs? Although it seems that after a connectivity change we should be able to adjust the automata's state in constant amortized time, the process is actually rather involved and error-prone. Thus, we restart the pattern matcher every time a connectivity change happens. The worst-case running time now increases to quadratic time in terms of the number of elements in a net (not layout objects). Since most of the program's time is spent in the region query routines (for all practical cases), we need a scheme so that, if we visit a pattern and no changes are possible on that pattern, that pattern is not generated over and over as we iterate. This means that only a linear number of patterns are passed to moving routines, which saves us from a time-complexity explosion.

The scheme is as follows: As we visit an element if no changes occur, we mark the element as visited and not changed. If a change happens, the pattern is marked changed. If a pattern is marked changed and no changes happen, we reset the element's flag to not changed. If all the elements of a pattern are marked visited and unchanged, we do not pass it to the moving routines. This scheme achieves the above goal: as we iterate, we do not look at the patterns that we have previously considered, where no improvements could be done.

Moving Transformations

The moving transformations take a pattern and apply local changes to that pattern. They are broken down into two sets: routing and placement transformations.

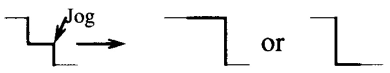
Routing Transformations

There are two kinds of useful patterns: jogs and U-turns. For each one, there is a corresponding transformation, which is invoked by the pattern matcher. We present simplified versions of our transformations here.

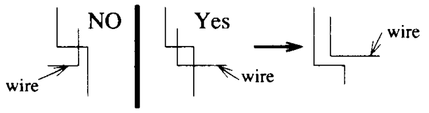
Jog Transformation Algorithm

```
1) If minimize corner flag is ON, try to flip the corner by
    pushing the second wire chain of the jog to the level
    of the first or the last.
2) If minimize wire crossing is ON
    2.1) Get all the wires parallel to the second wire chain in
        the jog's rectangle.
    2.2) Break them into two pieces, the ones higher and lower
        than the second wire chain.
    2.3) Sort each group so that the first one in each group is the
        furthest from the second wire chain.
    2.4) Consider the position between each wire and its
        predecessor in the above groups. Push the second
        wire chain to this position if less wire crossings would
        result.
```

The above algorithm has two functions. First, it tries to minimize the number of corners by pushing the second wire chain to maximum or minimum locations defined by the jog (Figure 4). Second, it tries to heuristically minimize wire crossings. The heuristic for choosing a move is whether the net of the parallel wire crosses the jog twice (Figure 5).



Corner Minimization Strategy for Jogs
Figure 4



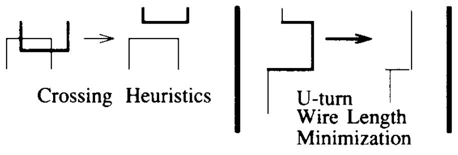
Crossing Minimization Heuristics for Jogs
Figure 5

The next routing algorithm manipulates U-turns. It minimizes wire length, corners, and crossings.

U-turn Transformation Algorithm

```
1) If minimum wire length or minimum corner flag is ON, try to push
   the second wire chain as far as possible if wire length or number
   of corners will be minimized.
2) If minimum wire crossing flag is ON
   2.1) Get the wires parallel to the second wire chain in the U-turn
        rectangle.
   2.2) Sort the wires so that the first one is furthest from the second
        wire chain.
   2.3) For each parallel wire, if wire crossings are minimized, try to
        push the second wire chain between the parallel wire
        and its predecessor.
```

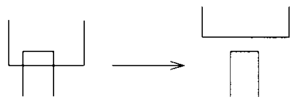
The algorithm tries to push the second wire chain to the maximum position allowed by the U-turn (Figure 6, right), if as a result wire length or corners are minimized. If the operation is not successful or the user does not want to optimize Steiner trees, the algorithm tries to heuristically minimize wire crossings. The heuristic is whether the wire chain is not completely contained in the second wire chain (Figure 6, left).



U-turn Crossing and Wire Length Minimization

Figure 6

Each topological move is considered twice. Figure 7 shows a situation where, during the processing of the first pattern, the better topology is not found; but it is found when the second pattern is processed.

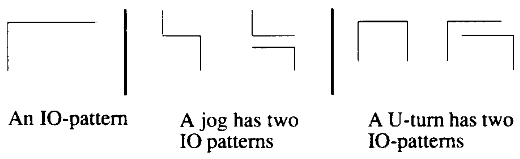


The move happens when the smaller U-turn is processed.

Order Dependency of Moves
Figure 7

Placement Transformations

The placement modifications are limited to top-level IO. For placement modifications, the concept of useful patterns is too general. To cope with this problem, we define the concept of IO-pattern. An IO-pattern is a jog or U-turn with the last piece left out. A jog or U-turn can have at most two IO-patterns. This happens when the first and last wire chains are connected to IOs (Figure 8).



IO-pattern
Figure 8

The algorithm we present here also takes care of equivalent logical pin swapping. However, no results of this kind of optimization are reported in this paper. There are three data objects: terminals, terminal instances, and IO pads. Two operations are defined: move and swap. Terminals can only move. Terminal instances can only swap. IO pads can both move and swap. The move operation first tries to align terminals and pad cells (Figure 9, left) to decrease wire length, contacts, and corners. If the operation can not align the IO in question, it tries to re-position the IO to decrease wire crossings.

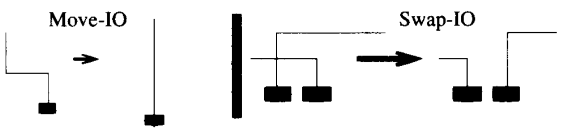


Figure 9

Swap attempts to minimize wire crossings by re-positioning IOs (Figure 9, right). Consequently, wire length and the number of vias might decrease; however, this decrease is not guaranteed.

Move-IO Algorithm

```
1) Get all the relevant IOs in the rectangle formed by the
   IO-pattern.
2) Sort all the IOs with respect to the moving direction.
3) Until no more choices are available or a move happens, do
   3.1) If a jog can be formed with the IO-pattern as part of it
            Try to straighten the jog (align IO).
   3.2) - Get all the wires crossing the IO-pattern
        - For each IO in the IO list
              If the IO net crosses the IO-pattern
                  try to move the IO beyond the crossing wire.
```

Swap-IO Algorithm

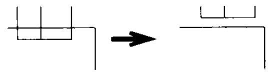
```
1) Get all the relevant IOs in the rectangle formed by the
   IO-pattern.
2) Sort all the IOs with respect to the moving direction.
3) Until no more choices are available or a move happens
   3.1) Get all the wires crossing the IO-pattern.
   3.2) For all the IOs in the IO list
        3.2.1) If the IO's net crosses the IO-pattern
                   Try to swap this IO with the pattern's IO.
```

Analysis

In this section, we look first at the program from a functional point of view. Then, we discuss the time and space complexity of the program.

Functional Analysis

Instead of using heuristics to guide moving transformations, one can use actual calculations. Although actual calculations are easy to incorporate into our system, we have not done this for a few reasons. First, our heuristics work well. In almost all cases, they find the optimal solution. This is especially true when each layer has a fixed direction, implying wires of one direction will not prevent movements of wires of the other direction. Second, the incorporation of actual calculations would increase the running time. Third, there are cases where a "bad" move is taken and is corrected by subsequent moves. If actual calculations are used, such bad moves might be discarded. In Figure 10, a three-terminal net moves beyond the horizontal wire in a sequence of two moves. After the first move, the number of corners and contacts might increase, which might be regarded as a bad move.



The move happens in two steps. The first move might increase the number of contacts or corners. However, the second move corrects this problem.

Stepwise Refinement
Figure 10

Although our transformations are only concerned with the second wire chains, a very good portion of the topological solution space is covered. The reason is that every wire chain is the second wire chain of some pattern, with the exception of the first and last. However, the first and last wire chains are connected to IOs; therefore, if they are movable, they are considered by IO moving transformations.

At this point, we will compare our work to the work of others. Wire crossing minimization is studied in [3] and [10]. In [3], wire crossing minimization between the routing areas is attacked. Although an elegant approach is proposed, since the optimization happens during global routing, some information might be lost during detailed routing. Also, wire minimization inside routing areas is not addressed. However, wire crossing minimization inside the routing areas is studied in [10]. The same methodological approach to Unconstrained Via Minimization as ours is proposed; that is, minimizing wire crossings to create new topologies, and then using a standard re-layering routine to minimize vias. Because of insensitivity of the approach in [10] to area-contact trade-off, a practical algorithm is not presented (results are obtained manually).

Our Steiner tree heuristics are very good in reducing the meandering paths, thereby producing better Steiner trees. Our heuristics handle multi-terminal nets effectively. However, our heuristics do not make any new global decisions; that is, they do not perform a rip up and re-route function. Our approach is superior to traditional wire channel straighteners [7] since it optimizes across various layers. A different approach to reduce meandering paths is presented in [6], which involves creating alternating paths to reduce meandering paths.

The IO alignment problem is handled effectively by the program: given the side of an IO, optimal solutions are often found. Our experience tells us that it is better to perform IO swapping during global routing. Nonetheless, the IO move operation is necessary to clean up after compaction. Many placers try to come up with optimal positions for top-level IOs during placement.

Complexity Analysis

Let n be the number of wires, contacts, pins, pin instances, and cell instances.

Time: We use sophisticated data structures, which are similar to k-d trees. The expected look-up time is O(logn) if the number of retrieved elements is small. In the worst case, the look-up time can be linear.

Let M be the maximum number of elements in a net. If the number of connectivity changes is proportional to M, the pattern matcher can take time O(M). If the look-up time is linear, the worst-case time for a transformation is O(n). By the modification we presented in the pattern matching section, the number of times we call the moving rules is linear. Thus, the worst-case running time is O(M + n). Since n is bigger than M, the worst-case running time is O(n). However, the expected running time is O(nlogn), where a linear number of patterns is generated and the look-up times are logarithmic. Our experiments show that, in fact, the number of generated patterns is linear, and the look-up time is layout-dependent but seems to be logarithmic. For example, the complexity of switch-box areas has a great effect on the running time. Interestingly enough, in cases where Topology Optimization takes longer, both the router and compactor take longer, too.

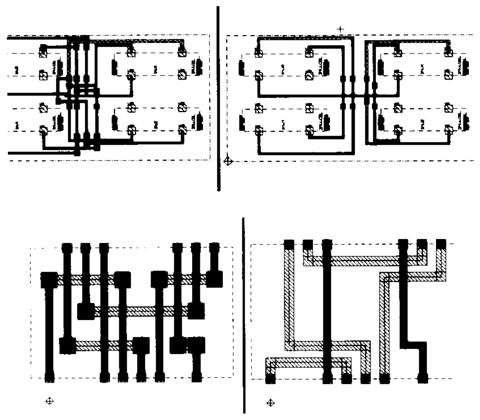
Space: Empirically the program's space is divided roughly into three pieces: one for object size, one for connectivity size, and one for data structures. Empirically the average space is 20n words (4 bytes). The upper bound on space is 40n words.

Experimental Results

We present two sets of examples. In the first section, we give two laboratory examples that have appeared in the recent works. Then we present our results on actual circuits. More examples are provided in [4].

Laboratory Examples

In [3], a test case for a wire-ordering problem is shown. The diagram below shows that Topology Optimization produces the optimal solution (Figure 11, top). In [8], a test case is shown, on which topology is changed to minimize vias. The solution reported has three vias. By using Topology Optimization and Via Minimization, we got the planar solution (Figure 11, bottom).



In each diagram, left is before optimization and right is after optimization.

Laboratory Examples
Figure 11

Real-Life Circuits

We conducted tests on six industrial circuits, which mostly represent block-oriented layouts. Compared to standard-cell designs, block-oriented layouts are more complicated; therefore, we expect more optimizations can be done. The results are compared in three different modes. First, only routing and via minimization is done. Then, Topology Optimization with only routing modification is executed. Finally, Topology Optimization with both routing and placement changes is performed. In some cases, no IO optimization is possible because the IO cells are big IO rings (IO optimization is done only on pad cells with one routed pin instance).

In the following table, the percentage appearing in "Route & Via Min" under "Route Area" is the ratio of routing area to total area. The other percentages are the improvements over via minimization as the only optimization performed. The table shows that, compared to routing without topological optimization, on average we achieve a 14.5% reduction in routing area, a 27% reduction in the number of vias, and a 10.5% reduction in wire length. The CPU times are measured on a Sun 4/260 and are reported in seconds.

Conclusion

In this paper, we present a new approach to several layout optimization problems: Wire Crossing Minimization, Topological Via Minimization, Minimum Steiner Tree Optimization, and IO Alignment. Our approach is based on pattern-recognition techniques and is accompanied by efficient heuristic transformations. The system is fully coded and tested, and will be included in the next Cadence IC design software release. It performs very well on both laboratory and real-life examples. For our industrial test cases, compared to layout without topological optimization, we have recorded average reductions of about 14.5% in routing area, 27% in the number of vias, and 10.5% in wire length. The running time is very reasonable (expected O(nlogn)) and falls between our symbolic router and compactor. The space requirement of our system is linear and is empirically measured at about O(20n) words.

References

[1] John E. Hopcroft, Jeffrey D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 1979.

[2] Nang-Ping Chen, "Building Block Routing - A Symbolic Approach," Custom Integrated Circuits Conference, 1988.

[3] P. Groenveld, "On Global Wire Ordering For Macro-cell Routing," Proc. 26th Design Automation Conference, pp. 155-160, 1989.

[4] Ramin Hojati, Duan-Ping Chen "Transformation-Based Layout Optimization," Custom Integrated Circuits Conference, 1990.

[5] Yuh-Zen Liao, C.K. Wong, "An Algorithm to Compact a VLSI Symbolic Layout With Mixed Constraints," IEEE Transactions On Computer-Aided Design of Integrated Circuits and Systems, Vol. CAD-2, No. 2, April 1983.

[6] H. Nelson, Robert J. Smith, II, "Verification and Optimization for LSI & PCB Layout," Proc. 18th Design Automation Conference, pp. 140-144, 1981.

[7] J. Royle, M. Palczewski, H. Verheyen, N. Naccache, and J. Soukup, "Geometrical Compaction in One Dimension for Channel Routing," Proc. 24th Design Automation Conference, pp. 140-144, 1987.

[8] Khe-Sing The, D.F. Wong, Jingsheng Cong, "Via Minimization By Layout Modification," Proc. 26th Design Automation Conference, pp. 799-802, 1989.

[9] Xiao-Ming Xiong, Ernest Kuh, "The Constrained Via Minimization Problem for PCB and VLSI Design," Proc. 25th Design Automation Conference, pp. 155-160, 1988.

[10] Xiao-Ming Xiong, "A New Algorithm for Topological Routing and Via Minimization," ICCAD, 1989.

Test Case				Route & Via Min	Routing, Topology, Optimization, Via Min	Routing, Topology Optimization, IO Align, Via Min
1	# Wires	480	Route Area	392.0 (34.8%)	331.9 (15.3%)	311.4 (20.6%)
	# Contacts	326	Contacts	176	107 (39.2%)	103 (41.5%)
	# Term & Term Instances	33 + 210	Wire Length	4104.8	3785.1 (7.8%)	3727.7 (9.2%)
	# Pads & Cells	0 + 11	CPUTime		7.1	8.0
2	# Wires	2105	Route Area	679.0 (62.1%)	573.2 (15.6%)	483.9 (28.7%)
	# Contacts	1422	Contacts	1337	1042 (22.1%)	723 (45.9%)
	# Term & Term Instances	197 + 824	Wire Length	7977.3	7019.8 (12.0%)	6099.3 (23.5%)
	# Pads & Cells	0 + 20	CPUTime		55.8	52.3
3	# Wires	1002	Route Area	950.2 (40.2%)	841.9 (11.4%)	834.8 (12.1%)
	# Contacts	777	Contacts	627	505 (19.5%)	501 (20.1%)
	# Term & Term Instances	0 + 399	Wire Length	373267	355787 (5.2%)	347700 (6.8%)
	# Pads & Cells	55 + 17	CPUTime		15.0	16.7
4	# Wires	1957	Route Area	9583.5 (52.7%)	8999.6 (6.1%)	8210.8 (14.3%)
	# Contacts	1366	Contacts	1289	1281 (.6%)	1139 (11.6%)
	# Term & Term Instances	101 + 660	Wire Length	1061.2	1005.5 (5.3%)	946 (10.9%)
	# Pads & Cells	0 + 17	CPUTime		61	69
5	# Wires	4028	Route Area	5323.7 (40.2%)	5028.0 (5.6%)	
	# Contacts	3141	Contacts	3038	2624 (13.6%)	
	# Term & Term Instances	0 + 1476	Wire Length	4377.1	4150.9 (5.2%)	
	# Pads & Cells	4 + 14	CPUTime		233.3	
6	# Wires	12237	Route Area	5391.4 (44.3%)	5090.7 (5.6%)	
	# Contacts	7419	Contacts	5884	4180 (29.0%)	
	# Term & Term Instances	0 + 3254	Wire Length	2306.9	2140.2 (7.2%)	
	# Pads &Cells	38 + 102	CPUTime		251	