Trunk Decomposition Based Global Routing Optimization

Devang Jariwala
Dept of Computer Science
Uni of Illinois at Chicago
djariwal@cs.uic.edu

John Lillis
Dept of Computer Science
Uni of Illinois at Chicago
lillis@uic.edu

ABSTRACT

We present global routing optimization methods which are not based on rip-up and re-route framework. In particular, the routing optimization is based on *trunk decomposition* [13] of the global routing. In this framework, the route of a net is decomposed into sets of wiring segments. By viewing a wiring segment as an "atomic object" of perturbation, we can efficiently evaluate the effect of routing tree perturbation. We propose two complementary routing optimization methods, namely *segment partitioning* and *segment migration*. These targeted optimizers can improve congestion related routing objectives by quickly shuffling wiring segments across different routing channels. Our routing approach produces better results compared to rip-up and re-route method based router Labyrinth [14] with average total overflow reduction of more than 88% while taking only 61% of runtime required by ripup and reroute phase of Labyrinth. When applied to the output of Labyrinth, the approach, on average, reduces the total overflow by more than 97% with complete overflow elimination for four circuits, while requiring additional runtime of just 33%. On a larger benchmark suite, the total overflow reduction of more than 86% is obtained, with complete overflow elimination for eight circuits, while requiring only 19% additional runtime.

1. INTRODUCTION

Routing is a crucial part of the VLSI design process and this is increasingly true in modern designs in which design closure and total design time are impacted by the effectiveness and speed of the routing engine. In the nanometer designs, variability and manufacturability related issues have posed new challenges in routing domain, making it even more critical for achieving closure. In a traditional design flow, the routing problem is divided in two phases: global routing and detailed routing. The focus of this paper is on the global routing phase in which "rough" routes for nets are determined so as to manage the routing demand over the chip (given the guidelines of the global routing, detailed routing determines exact routes). During global routing, the routing region is typically divided into a 2-D array of tiles. A routing graph is constructed in which each node corresponds to a routing tile and each edge corresponds to a *routing channel* connecting two adjacent tiles. Each of these tiles has physical dimensions which determines the *capacity* of edges incident on the tile – i.e., maximum number of wires which may flow through an edge. The simplest version of the global routing problem then is to find a routing tree in the routing graph for each net in the design such that no edge capacity is exceeded. Optimization variants might attempt to minimize the maximum edge usage (or density) or to minimize the total overflow (violation of edge capacities).

The techniques employed for solving routing problems can broadly be classified into the following categories: ripup and reroute based techniques, multicommodity flow based techniques and hierarchical/multilevel techniques. Ripup and reroute techniques are essentially sequential routing methods in which each net is routed in some order while taking into account the demand contribution from nets already routed [16], [14], [10], [8]. For individual net routing, a number of heuristic methods have been proposed, e.g. maze routing [15] or Steiner tree construction heuristics [12]. These techniques differ from each other in terms of the objective to be optimized and the criteria for the selection of the nets to be ripped up and rerouted.

To the best of our knowledge, the majority of industry tools are based, at least in part, on the general concept of ripup and reroute. Thus, the idea has stood the test of time. Nevertheless, there are some obvious issues associated with the approach – the results can be sensitive to net ordering; defining "edge cost" functions in the best way is a technical challenge; runtime tends to be substantial, particularly as tile-size is decreased (to better model what the detailed router sees) and as the design size increases.

A second class of techniques is based on multicommodity flow [17] [6] [2] and can model the simultaneous routing of multiple nets. The main idea here is to model nets as different commodities that flow through the network of routing resource graph. In general the linear programming formulation will result in fractional flows corresponding to nets "partially" using certain routing resources. Therefore, a randomized rounding procedure is used to discretize the solution. Since solving the LP itself is a computationally intensive task, Albrecht [2] developed methods to approximate the LP solution with provable error bounds. To deal efficiently with multi-pin nets (i.e., to avoid enumerating all possible Steiner trees), a limited number of candidate

ICCAD'06, November 5-9, 2006, San Jose, CA

Steiner trees are usually constructed for each net and LP relaxation of the flow problem (or an approximation thereof) is solved, followed of course by randomized rounding. Ripup and reroute can be used at the end to further optimize the routing.

Hierarchical and multilevel techniques [4], [7] use, as the name suggests, multilevel optimization approach to reduce the complexity of the problem. Hierarchical techniques solve the routing problem in top-down flow, while multilevel techniques use combination of bottom-up and top-down flows. In [7], the authors compute resource capacities in a bottom up phase and start routing at the highest level using multicommodity flow based algorithm. During a top-down phase, maze routing is used to refine the routes.

Contributions

The goal of this paper is to explore routing optimization from different point of view and devise optimizers for this alternative view. This may be complementary to the other views of the solution space.

In particular, the view of the solution space we propose is based on the *trunk decomposition* [13] of existing global routing structures (determined by any desired means) and then perform congestion optimization by moving wiring segments in the decomposition. Within this framework, we propose a set of perturbation operators for optimizing routing configuration of a net. We prove that the set of perturbation operator is "complete." In other words, with the perturbation operators in this set, all global routing configurations are reachable. Within trunk decomposition framework, we propose two complementary optimizers. The first one examines vertical (horizontal) segments flowing through a pair of tiles in adjacent columns (rows) in the routing grid. These segments are partitioned between the two columns (rows) to minimize congestion related objectives (details of these objectives are important and discussed in section 3). Segment partitioning can simultaneously change the routing topologies of multiple nets. The second optimizer is based on greedy segment migration which allows segments to be displaced greater distance.

Similar concepts have been employed in the iterative improvement of Steiner trees (e.g. [3], [11]). In [3], the authors use the *flexibility* of the Steiner tree to flip the corners in order to reduce the overflow. But in this case the topology of the Steiner tree is never changed. In [11], a Steiner tree is improved by including non-hannan points to reduce the arrival time violation at sinks. The characteristics of the Elmore delay function are used to efficiently *reconnect* the sink with violation. Both these works, and several others, target individual Steiner trees. Thus, the similarities with our work is limited by the fact that we consider modification of Steiner trees of multiple nets at the same time. Moreover, unlike [3], we allow the changes in the topology of the original Steiner tree of nets. In [13], the authors introduced a simultaneous placement and routing optimization by capturing placement and routing structures in a unified combinatorial framework. But the dynamic programming formulation proposed has stringent requirements on nature of the objective functions it can optimize and it is difficult to implement and has relatively high computational complexity.

The rest of the paper is organized as follows: section 2 discusses the preliminaries. We also discuss trunk decomposition [13] for the sake of completeness in this section, followed by the partitioning algorithm for the routing segments in section 3. Another simple and fast greedy heuristic to reduce routing congestion is discussed in section 4. The experimental set up and the results are discussed in section 5 and we draw some conclusions in section 6.

2. PRELIMINARIES

A netlist is a set of logic gates or cells (C) and a set of interconnections (N), also known as nets. The placement phase determines the exact location of each cell $c \in C$. During global routing the entire chip is divided into routing tiles and pin locations are mapped onto this routing grid in a straightforward way. A routing graph $G = (V, E)$ is then constructed where each vertex $v \in V$ corresponds to a global routing tile and an edge $(u, v) \in E$ exists between the vertices u and v, if and only if the corresponding global routing tiles are adjacent. A global router must find a realization of each net as a Steiner tree in G connecting all of the vertices (tiles) belonging to the net. Each routing edge $e = (u, v) \in E$ has a *capacity* $c(e)$ determined by the technology and the granularity of the global routing grid; it models how many wires are available between the corresponding global routing tiles.

Given a routing solution in which all nets are realized, the number of wires actually passing through an edge e is known as the demand on the edge e and is denoted as $d(e)$. If the number of nets passing through an edge is larger than the capacity of the edge, the edge is denoted as an *overflowing* edge. The amount of overflow on an edge ($oflow(e)$) is defined as follows:

$$oflow(e) = \begin{cases} 0 & \text{if } d(e) \leq c(e); \\ d(e) - c(e) & \text{otherwise.} \end{cases}$$

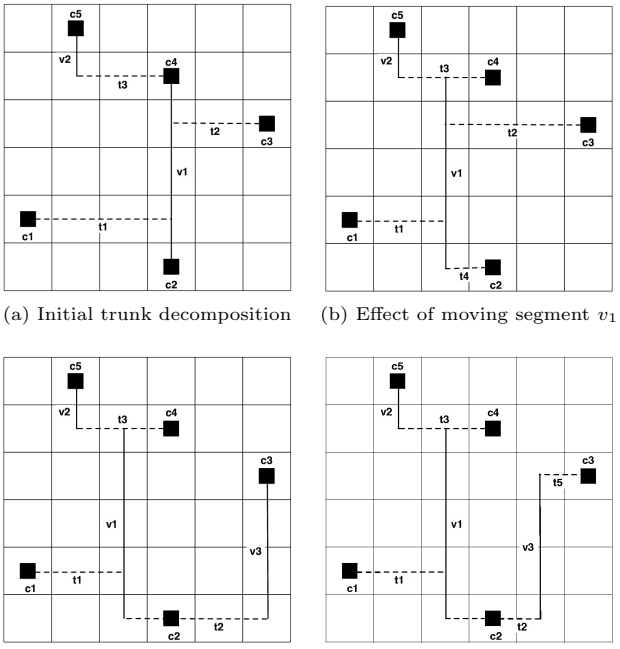
Total overflow (OF) is the sum of the overflow over all the edges in the routing graph ($OF = \sum_{e \in E} oflow(e)$). Total overflow has been used as an indicator of the quality of a routing solution: generally, the higher the total overflow, the less likely detailed routing will complete. Another important metric of a routing solution is the *maximum overflow*, defined as $OF_{max} = \max_{\forall e \in E}\{oflow(e)\}$. The maximum routing density or demand is defined as $D_{max} = \max_{\forall e \in E}\{d(e)\}$, while number of edges at D_{max} is $nMax = |\{e \mid d(e) = D_{max}\}|$. The total routed wirelength of a routing is defined as $WL = \sum_{\forall e \in E} len(e) \times d(e)$, where $len(e)$ is the length of the edge $e \in E$. Traditionally, routing cost is some combination of total overflow, wirelength and maximum overflow.

Trunk Decomposition

In [13], the authors proposed a different way to represent the routing information for a net, called *trunk decomposition*. Unlike traditional routing approaches, in which the "atomic" object of perturbation is a net, in this framework, individual wiring segments are the atomic objects of perturbation. This allows us to view the routing space from a different point of view.

In this framework, the global route of a net is decomposed and represented with the sets of two entities:

- **Horizontal Trunk:** A horizontal trunk is a continuous horizontal wire of a routing tree. It can connect to multiple cells and vertical segments. It can also have a zero length.



(a) Initial trunk decomposition (b) Effect of moving segment v_1

(c) Effect of moving trunk t_2 (d) Effect of moving segment v_3

Figure 1: Trunk decomposition of global route and effect of segment/trunk movement.

- **Vertical Segment:** A vertical segment is a continuous vertical wire of a routing tree. It can connect to multiple horizontal trunks and cells. If the vertical segment is not connected to any cell, it is called a *free* segment.

Fig. 1 (a) shows global route of a five-pin net. The global route is decomposed into two sets: a set of vertical segments $\{v_1, v_2\}$ and a set of horizontal trunks: $\{t_1, t_2, t_3\}$. These sets together exactly define the global route of a net. It should be noted that the trunk decomposition shown in Fig. 1 (a) is only one of many such decomposition possible for a given global route of the net.

The main feature of trunk decomposition is that it allows us to view the routing solution space differently. We describe a set of perturbation operators, which can be used to optimize the routing configuration of a net in this framework. Although we describe the perturbation operators for the vertical segments, similar moves can also be defined for the horizontal trunks. The perturbation set includes:

- **Displacement:** Each free vertical segment can be displaced in horizontal direction, with possible contraction or elongation of the connected trunks.

- **Break:** We can "break" a vertical segment along its length and create two vertical segments by inserting a zero length trunk at their meeting point. This operation is useful in creating free vertical segments from connected ones.

- **Merge:** Two or more overlapping segments passing through same column or channel can be merged together to create a single vertical segment.

- **Split:** A vertical segment can be "split" by replicating a part of it and possibly introducing a zero length trunk at one of the ends connecting the two segments. As a result of this move, there will be two (or possibly more) overlapping segments in the same routing channel.

We can always insert zero length trunk where a vertical segment connects a cell so that we can make the vertical segment free. By applying one of these operators, we can change the routing configuration of a net. Fig. 1 (b) shows the effect of displacing segment v_1 left by one column after inserting zero length trunk at connection with cell c_2. By doing so, trunk t_1 gets contracted, while trunks t_2 and t_4 get elongated. This move illustrates the fact that even though we displace a vertical segment, there are implications in the horizontal routing configuration as well. Fig. 1 (c) shows the effect of displacing trunk t_2, after introduces additional vertical segment v_3 and trunk t_2 gets merged with trunk t_4. Finally Fig. 1 (d) shows the effect of moving vertical segment v_3 which again requires insertion of an additional trunk. Fig. 1 demonstrates the fact that with the perturbation operators we can essentially change the routing topology of a net. Similarly, Fig. 2 (b) shows application of break and split operator on segment v_1.

Some observations about the trunk decomposition and the perturbation set are in order:

- The atomic objects in a trunk decomposition are individual wiring segments rather than entire nets.

- Individual moves or perturbation operators are quite simple and can be evaluated quickly. In contrast, rerouting an entire net often employs an expensive maze running while the ultimate perturbation of the solution resulting from the reroute may be small.

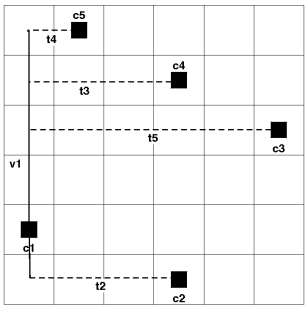
- Moves can result in perturbation of topological structure of Steiner trees.

- All the operators described above are reversible.

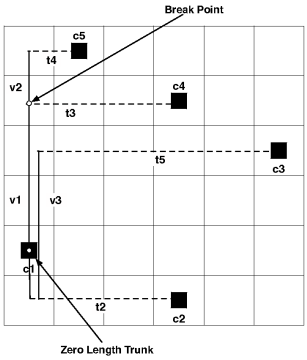
A desirable property of any set of perturbation operators is the ability to cover the entire search space, i.e. all pairs of configurations are mutually reachable by some sequence of perturbations. The following theorem captures some of the generality of the perturbations.

THEOREM 2.1. *The operators defined in the move-set above are sufficient to transform any valid routing configuration of a net to any other valid configuration. (In this sense, the perturbation set is complete.)*

The proof of the theorem is straight forward and relies on the fact that all the operators are reversible. We give general idea of the proof here. For a given placement of pins of a net, let's define a canonical routing configuration Γ. Fig. 2 (a) shows the canonical configuration for the net in Fig. 1. In the canonical routing configuration, there is only one vertical segment aligned with the left most pin on the net. All the other pins connect to this segment by means of a horizontal trunk, if necessary. We can transform any routing configuration (γ_1) to the canonical configuration by moving all the vertical segments so that they are aligned with the left most pin and perform necessary merging at

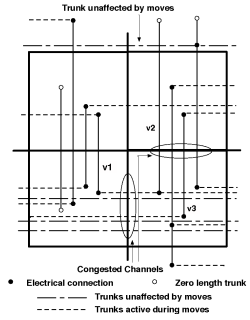


(a) Canonical form for routing shown in Fig. 1 (a)

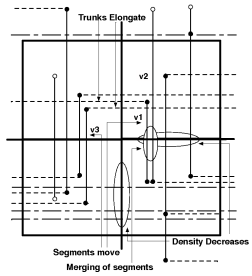


(b) Splitting vertical segment v_1. This can give configuration in Fig. 1 (d) after displacement of vertical segments.

Figure 2: Transformation from canonical form.



(a) Initial Configuration.



(b) As a result of displacement, v_1 and v_3 swap channels. So v_1 and v_2 can be merged and overflow in both congested channels can be improved in both directions.

Figure 3: Illustration of a routing instance in a 2×2 sub-grid. There are total of 12 routing edges in the figure, out of which 2 are congested.

that point to generate a single vertical segment. As the perturbation operators are reversible, we can revert back to configuration γ_1 from Γ. Similarly, for any other valid routing configuration γ_2, the same can be applied. So we have following transformation: $\gamma_1 \rightarrow \Gamma \rightarrow \gamma_2$. As γ_1 and γ_2 are arbitrarily chosen, we can say that any valid routing configuration can be transformed to any other valid routing configuration.

Fig. 2 shows how we can move from the canonical configuration (a) to the configuration shown in Fig. 1 (d). However, it should be noted that going through the canonical configuration may not be the best way to reach the other configuration.

3. SEGMENT PARTITIONING

In this section, we introduce an optimization technique that modifies the routing configuration of multiple nets simultaneously by employing some of the perturbation operators described in the previous section.

Let's consider the routing configuration shown in Fig. 3 (a), which is 2x2 routing subgrid. There are 12 routing edges shown in the figure, with two congested edges (one in each direction). Fig. 3 (b) shows the modified configuration in which segments v_1 and v_3 swap their positions. This results in merging on v_1 and v_2. While trunks elongate on the top half of the grid, the density in the horizontal density in lower half is reduced. This example illustrates the effect of simultaneous perturbation of multiple nets.

The example discussed above motivates the idea of developing a segment-shuffling technique to improve routing congestion. *Segment partitioning*, based on the idea of general partitioning, can shuffle the segments in neighboring channels efficiently. In the partitioning problem, we are given a set of entities and the goal is to partition this set into two disjoint subsets such that some cost function is optimized. Usually, there is also a balance constraint on the size of each subset. In the case of routing optimization, the idea is to choose segments from neighboring channels (by row or by column) and find new channel assignment of these segments so that some cost function is optimized.

We now describe the details of the partitioning algorithm. For the sake of simplicity, we only describe the algorithm with respect to vertical segment partitioning, however, it should be noted that the ideas are also valid for the horizontal trunk partitioning problem. Our partitioning method is based on the FM partitioning method [9]. FM partitioning is a method for graph partitioning to minimize cut-size. It starts with an initial partitioning of the graph and iteratively makes best "move" to improve current partitioning until all vertices are moved exactly once. During this process, it also keeps track of the best solution obtained. At the end all the moves made after the best solution was achieved are "undone." This process is called one pass of the algorithm, and additional passes are invoked until there is no improvement in the solution quality.

The segment partitioning method also works pass by pass. We choose certain number of vertical segments passing through neighboring routing channels. This assignment acts as the

```
Algorithm Segment Partition Pass:
        Input: A, B : Initial Segment Partitions
        Output: A, B: Modified Segment Partitions
1       unlockAllSegments
2       moveNum ← 1
3       bestMoveNum ← 0
4       bestCost ← ∞
5       Loop
6           a ← minCostSegmentA2B
7           cost_a ← costA2B(a)
8           b ← minCostSegmentB2A
9           cost_b ← costB2A(b)
10          If cost_a < cost_b Then
11              moveA2B(a)
12              lockSegment(a)
13              If cost_a < bestCost Then
14                  bestCost ← cost_a
15                  bestMoveNum ← moveNum
16              End If
17          Else
18              moveB2A(b)
19              lockSegment(b)
20              If cost_b < bestCost Then
21                  bestCost ← cost_b
22                  bestMoveNum ← moveNum
23              End If
24          End If
25          moveNum ← moveNum + 1
26      Until All segments are locked
27      undoMoves(bestMoveNum)
```

Figure 4: Single pass of segment partitioning algorithm.

initial partitioning. In each pass, each vertical segment changes partition exactly once. The details of the each pass of partitioning is shown in Fig. 4. Each pass starts by unlocking all the segments taking part in partitioning (procedure *unlockAllSegments*). During the pass, we locate the best unlocked segment to move (procedures *minCostSegmentA2B* and *minCostSegmentB2A*). Once we find the best segment to move, we make the move (procedures *moveA2B* and *moveB2A*) and lock the segment for the rest of the pass (procedure *lockSegment*). During the pass, we also keep track of the sequence of moves leading up to the best solution. At the end, when all segments are moved once, we "undo" all the moves past the best solution (procedure *undoMoves*). This completes one pass. We repeat this procedure until there is no improvement in the solution quality.

3.1 Cost Objective

In this subsection we describe the cost objective optimized during segment partitioning. Usually, the total overflow of a global routing solution is used as a metric for the quality of the solution. We employ lexicographic combination of the maximum overflow (OF_{max}), the total overflow (OF) and the routed wirelength (WL) as the cost objective:

$$< OF_{max}, OF, WL >$$

Having described the segment partitioning algorithm for vertical segments, it is important to emphasize that the cost computation involves contribution for *both* directions. The move of a segment from one partition to the other is an application of *displacement* operator from the perturbation set described in the previous section. So it has implication not only in the vertical direction, but also in the horizontal direction, since the connected trunks may get elongated or contracted by this displacement. Similarly, for partitioning of horizontal trunks, there are cost implications in the orthogonal direction. Also, when segments move in the routing channels, they can potentially overlap with other segments of the same nets in the some routing channels, this helps reduce the overflow.

It should also be noted that although we use the cost objective described here, the method is flexible enough to accommodate wide variety of cost components, e.g. via density.

3.2 Complexity Analysis

In this subsection, we analyze the complexity of one pass of the segment partitioning algorithm. The loop starting on line 5 of Fig. 4, goes over each vertical segment taking part in the partitioning process exactly once. So if there are n_{seg} number of segments taking part in the partitioning process, the number of iterations of the loop are $O(n_{seg})$. In order to determine best segment to move, a linear search method is used. If H is the average height of a segment, it takes $O(H)$ to compute the vertical component of the cost. If *degree* is the average number of trunks connecting a segment, it takes $O(degree)$ time to compute horizontal component of the cost. Hence, after cost computation and linear search, the time required to find the best vertical segment to move is: $O(n_{seg} \times (H + degree))$. So overall complexity of each pass of partitioning is:

$$O(n_{seg}^2 \times (H + degree))$$

4. SEGMENT MIGRATION

In this section, we describe the second optimization method called *segment migration*. This method is essentially a greedy local search method. The idea is very simple: choose a segment passing through a congested channel and find an optimal location for the segment in a window around its current location. If x is the current location of the vertical segment, and window size is δ, we search for its best location in window $[x - \delta, x + \delta]$, with a caveat that the window span is always within the chip boundaries. It is easy to deduce that the complexity of this optimizer is less compared to segment migration. Moreover, it can cover parts of the solution space not covered by segment partitioning. But segment migration can change routing topology of only one net at a time. So both the optimizers are complementary in nature.

Segment migration can be applied to wiring segments in both the directions. When the cost of displacement of wiring segment in one direction is computed, it also includes the component of the cost in the orthogonal direction as well.

In the context of the perturbation set described in the Section 2, this method employs *displacement* as the main perturbation operator with *merge* as a post-processing operator. Although, it should be noted that the cost computation takes into account the effect of overlapping wiring segments in same channel. It can also use *break* and *split* pre-processing operators to further enhance solution space exploration.

5. EXPERIMENTS AND ANALYSIS

We have implemented segment partitioning and segment migration methods in C++. We have conducted experiments on 2.4 GHz Pentium 4 Linux workstation with 1 GB RAM. The standard IBM placement benchmarks are used

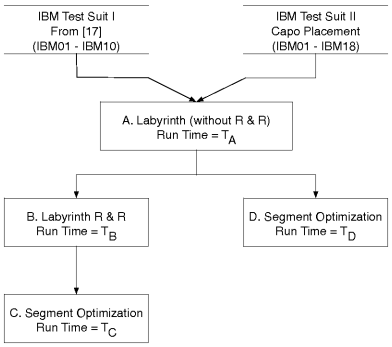


Figure 5: Experimental set up. There are four stages in experiments and two test suites. R & R stands for ripup and reroute.

Table 1: Characteristics of the benchmark circuits used for experiments.

Circuit	V_{cap}	H_{cap}	Grid
ibm01	12	14	64x64
ibm02	22	34	80x64
ibm03	20	30	80x64
ibm04	20	23	96x64
ibm05	42	63	128x64
ibm06	20	33	128x64
ibm07	21	36	192x64
ibm08	21	32	192x64
ibm09	14	28	256x64
ibm10	27	40	256x64
ibm11	22	32	128x128
ibm12	33	43	128x128
ibm13	20	32	196x128
ibm14	29	39	256x128
ibm15	38	48	256x128
ibm16	48	56	256x128
ibm17	49	69	256x128
ibm18	45	55	256x128

as the test suite. The general experimental set up is shown in Fig. 5. The set up revolves around Labyrinth router [1]. Labyrinth is a ripup and reroute (R & R) method based router. It also has an option of using pattern routing for some nets. The first stage of Labyrinth generates initial routing for each net, using either maze routing or pattern routing. This stage is denoted as A in the experimental flow. Then it starts R & R phase, in which nets passing through overflowing edges are ripped up and rerouted. This stage is denoted as B in the experimental set up. In stage C of the flow, we apply segment optimization, i.e. segment partitioning and segment migration, on output of the stage B, while in stage D, we apply segment optimization on output of stage A.

In general, segment partitioning and segment migration, on their own produce good results. Segment partitioning can reduce the overflow quickly and it ultimately saturates. While segment migration is fast, it takes comparatively more invocations and its search space is slightly more global. After trying out with different order of applications of these two optimizers, it is observed that, usually, segment migration followed by segment partitioning yields better results. In the rest of the discussion, segment optimization stage includes migration followed by partitioning followed by two more passes of migration.

Now we shall describe the detailed results of experiments on test suite I and II.

Table 2: Change in solution quality of stage D and of stage C over output of Chi router. Runtime comparison is not included because the router was not available. For ibm05 circuit, no overflow result is reported by Chi.

Circuit	Chi Output		Stage D change		Stage C change	
	OF	WL	OF%	WL%	OF%	WL%
ibm01	189	64355	-40.74	10.22	-78.31	18.39
ibm02	66	175368	-40.91	9.20	-86.36	17.04
ibm03	7	149695	85.71	8.59	-85.71	23.32
ibm04	411	170440	0.73	8.02	-68.37	13.41
ibm06	16	184700	-100.00	70.02	-93.75	84.44
ibm07	251	373739	-94.82	16.36	-100.00	16.36
ibm08	71	410507	-42.25	8.60	-100.00	19.69
ibm09	35	420691	-54.29	9.04	-100.00	13.80
ibm10	116	589508	-96.55	8.55	-100.00	14.46
Average			-42.57	16.51	-90.28	24.55

5.1 Test Suite I

The test suite I is based on ISPD98 IBM placement benchmarks with the cell placements and the routing parameters given by [1]. It has first 10 IBM benchmarks, ibm01 to ibm10. The routing parameters of the test suite are given in the first part of Table 1. As recommended in [10], we run Labyrinth (both stages A and B) by routing 70% of smaller nets using pattern routing.

The first part of Table 3 shows the output of stages B, D and C from Fig. 5. The improvement by segment optimization over output of stage B of Labyrinth, for test suite I is shown in the second part of Table 3. The results of Stage D reduce the maximum overflow by more than 2 tracks on average with the maximum reduction of 5 tracks for circuit ibm08. The total overflow reduction of more than 88% is obtained on average with maximum reduction of 100% for ibm06 circuit. The wirelength output of this stage is always better than Labyrinth phase B. The maximum wirelength reduction of more than 7% is obtained for circuit ibm01. For runtime comparison, runtime of stage B of Labyrinth is taken as normalizing factor. So on average, stage D takes about 61% of time taken by R & R phase of Labyrinth. Application of segment optimization on the output of Labyrinth stage B, yields even better results, with complete overflow elimination for four of the circuits and average total overflow reduction of 97%. The average reduction in maximum overflow for stage C is more than 3 tracks with maximum reduction of 7 tracks for circuit ibm08. The wirelength results are always better compared to stage B output as well. For runtime comparison, we take combined runtime of Labyrinth stage A and stage B as normalizing factor. So the improvement in the solution quality, during stage C is obtained with 33% additional runtime over and above total runtime of Labyrinth router.

Table 2 shows the comparison of the results of stages D and C with Chi router [10]. Since we did not have access to the router itself, we do not provide runtime comparison here. Moreover, the maximum overflow results were not reported in [10], so that is also excluded from the table. Both stages D and C produce good results in terms of total overflow reduction. On average, the total overflow reduction of more than 42% is obtained at the end of stage C, while stage D shows more than 90% reduction in the same. However, these stages do rely on input from Labyrinth, which tends to have larger wirelength. And hence, the wirelength of Chi router is better than the wirelength at the end of stage C and stage

Table 3: Performance of stage B, D and C of the experimental set up on the test suite I.

Circuit	Labyrinth stage B				Segment Optimization stage D					Segment Optimization stage C				
	OFmax	OF	WL	Time(sec)	OFmax	OF	WL	Time(sec)		OFmax	OF	WL	Time(sec)	
								Part	Mig				Part	Mig
ibm01	3	432	76845	21.4	3	112	70932	11.78	28.78	2	41	76191	8.28	19.18
ibm02	6	670	206317	52.66	2	39	191504	16.83	20.26	1	9	205251	13.47	16.96
ibm03	3	215	185222	51.92	1	13	162555	12.58	17.02	1	1	184604	10.87	11.01
ibm04	5	889	194086	93.23	4	414	184107	23.37	42.45	3	130	193300	19.69	33.07
ibm06	5	515	341466	68.08	0	0	314036	8.91	5.96	1	1	340658	20.77	19.55
ibm07	3	259	435181	125.62	1	13	434889	23.02	24.36	0	0	434889	0.14	6.08
ibm08	7	773	492487	205.26	2	41	445804	35.59	42.25	0	0	491321	26.9	9.52
ibm09	3	412	479413	173.63	1	16	458708	25.06	26.16	0	0	478729	25.83	7.64
ibm10	3	407	675577	243.95	1	4	639936	38.37	36.77	0	0	674745	6.13	10.99

Circuit	Stage D change				Stage C change			
	OFmax	OF%	WL%	Time[a]	OFmax	OF%	WL%	Time[b]
ibm01	0	-74.07	-7.69	1.90	-1	-90.51	-0.85	1.11
ibm02	-4	-94.18	-0.07	0.70	-5	-98.66	-0.52	0.46
ibm03	-2	-93.95	-0.12	0.57	-2	-99.53	-0.33	0.36
ibm04	-1	-53.43	-0.05	0.71	-2	-85.38	-0.40	0.50
ibm06	-5	-100.00	-0.08	0.22	-4	-99.81	-0.24	0.32
ibm07	-2	-94.98	0.00	0.38	-3	-100.00	-0.07	0.02
ibm08	-5	-94.70	-0.09	0.38	-7	-100.00	-0.24	0.11
ibm09	-2	-96.12	-0.04	0.29	-3	-100.00	-0.14	0.10
ibm10	-2	-99.02	-0.05	0.31	-3	-100.00	-0.12	0.03
Average	-2.56	-88.94	-0.91	0.61	-3.33	-97.10	-0.32	0.33

[a]Time $= T_D/T_B$
[b]Time $= T_C/(T_A + T_B)$

D[1].

5.2 Test Suite II

In order to test the scalability of the approach, more experiments are needed with larger circuits in the IBM placement benchmarks. Hence, we generated additional set of benchmarks, called suite II for further experiments. The details of the routing parameters of these benchmarks are shown in Table 1. For first ten benchmarks, we have identical routing parameters with suite I. For additional eight benchmarks, we generated routing grids such that there are on average 5-6 cells per routing tile. The placements for these benchmarks are obtained using Capo (version 9.3) [5] placement tool. Again, the experimental set up is same as the one described in Fig. 5.

The first part of Table 4 shows the performance of stages B, D and C of the experimental set up. In this case, we compare the results of D and C stages with the result from stage B in the second part of the Table 4. The results of stage D show an average reduction of more than 3 tracks in maximum overflow with maximum reduction of 10 tracks for ibm14 circuit. The average reduction in total overflow of more than 75% is observed, with complete overflow elimination for four circuits. The runtime of stage D is less than 60% of runtime for stage B of Labyrinth. The results of stage C looks even better. For eight circuits, the total overflow is reduced to zero with average reduction of more than 86%. The runtime of this stage is compared against the combined runtime of stages A and B of Labyrinth. So stage C takes about 19% of additional runtime to obtain its results.

6. CONCLUSION

We have presented a set of complementary routing optimizers within the framework of trunk decomposition. Perturbation operators from a complete move-set are used for routing optimization. The application of these methods on two benchmarks suite shows promising improvements in total and maximum overflow in relatively quick time.

Ongoing work includes devising new methods for generating better initial routing results, improving runtime and further enhancement of the techniques with applications other interesting areas, such as design for manufacturability. The implementation of the techniques is right now in beta phase and will eventually be made available to the CAD community for further research.

[1]We intend to do further analysis with alternate initial router.

7. REFERENCES

[1] Labyrinth webpage. `http://www.ece.ucsb.edu/~kastner/labyrinth`.

[2] C. Albrecht. Provably good global routing by a new approximation algorithm for multicommodity flow. In *Proc. of Intl. Symp. on Physical Design*, pages 19–25, April 2000.

[3] E. Bozorgzadeh, R. Kastner, and M. Sarrafzadeh. Creating and exploiting flexibility in rectilinear steiner trees. *IEEE Tran. on CAD of Integrated Circuits And Systems*, 22(5):605–615, May 2003.

[4] M. Burstein and R. Pelavin. Hierarchical wire routing. *IEEE Tran. on CAD of Integrated Circuits And Systems*, 2(4):223–234, October 1983.

[5] A. E. Caldwell, A. B. Kahng, and I. L. Markov. Can recursive bisection alone produce routable placements? In *Proc. of Design Automation Conf.*, pages 693–698, June 2000.

[6] R. C. Carden and C.-K. Cheng. A global router using an efficient approximate multicommodity multiterminal flow algorithm. In *Proc. of Design Automation Conf.*, pages 316–321, 1991.

[7] J. Cong, J. Fung, M. Xie, and Y. Zhang. Mars - a multilevel full-chip gridless routing system. *IEEE Tran. on CAD of Integrated Circuits And Systems*, 24(3):382–394, March 2005.

[8] C. Ebling, L. McMurchie, S. A. Hauck, and S. Burns. Placement and routing tools for the triptych fpga. *IEEE Tran. on Very Large Scale Integration (VLSI) Systems*, 3(4):473–492, 1995.

Table 4: Performance of stage B, D and C of the experimental set up on the test suite II. The placements are generated using Capo.

Circuit	Labyrinth stage B				Segment Optimization stage D					Segment Optimization stage C				
	OFmax	OF	WL	Time(sec)	OFmax	OF	WL	Time(sec)		OFmax	OF	WL	Time(sec)	
								Part	Mig				Part	Mig
ibm01	6	1480	88852	43.39	4	1030	84406	14.62	29.86	4	658	86639	12.97	27.65
ibm02	10	3709	233340	107.27	4	769	221995	27.31	38.15	3	565	229668	22.52	31.82
ibm03	2	112	185802	22.4	1	21	166884	27.83	10.15	0	0	185323	7.51	3.18
ibm04	10	5892	222507	153.3	6	5731	213245	61.8	28.18	7	3940	216887	35.25	49.33
ibm05	1	1	490599	0.86	0	0	490566	0	1.3	0	0	490598	0	0.01
ibm06	7	1281	355202	70.74	2	80	329110	25.65	29.8	2	73	352790	26.27	26.64
ibm07	9	4800	573857	595.96	6	2943	541016	59	81.54	4	959	567298	47.22	54.88
ibm08	14	16891	565455	2428.72	10	17937	536457	76.54	140.47	12	11446	552314	83.24	123.18
ibm09	5	1231	492071	89.07	2	47	476867	32.64	36.12	2	29	490494	30.56	34.14
ibm10	1	3	607359	11.71	0	0	606458	0	4.3	0	0	607360	0	6.37
ibm11	1	1	564674	56.75	0	0	558249	0.16	7.9	0	0	564672	0	1.75
ibm12	3	258	844991	279.95	1	2	820791	48.66	40.48	0	0	844504	3.73	11.34
ibm13	2	75	929841	292.18	1	5	895788	47.38	45.1	0	0	929530	0.15	11.92
ibm14	12	1252	1410262	292.13	2	307	1390190	79.12	80.09	2	297	1407543	76.39	76.43
ibm15	8	551	1753251	443.8	2	57	1667958	93.55	92.49	1	29	1750834	95.54	92.22
ibm16	1	1	1653280	37.55	0	0	1647798	0.29	25	0	0	1653280	0	6.37
ibm17	1	29	2267710	1272.13	1	1	2217948	121.68	110.87	0	0	2267525	0.36	31.05
ibm18	6	867	1800413	765.55	2	59	1710815	110.65	107.7	1	1	1797724	102.56	99.61

Circuit	Stage D change				Stage C change			
	OFmax	OF%	WL%	Time[a]	OFmax	OF%	WL%	Time[b]
ibm01	-2	-30.41	-5.00	1.03	-2	-55.54	-2.49	0.86
ibm02	-6	-79.27	-4.86	0.61	-7	-84.77	-1.57	0.45
ibm03	-1	-81.25	**-10.18**	1.70	-2	-100.00	-0.26	0.34
ibm04	-4	-2.73	-4.16	0.59	-3	-33.13	-2.53	0.50
ibm05	-1	-100.00	-0.01	1.51	-1	-100.00	0.00	0.00
ibm06	-5	-93.75	-7.35	0.78	-5	-94.30	-0.68	0.42
ibm07	-3	-38.69	-5.72	0.24	-5	-80.02	-1.14	0.14
ibm08	-4	6.19	-5.13	0.09	-2	-32.24	-2.32	0.08
ibm09	-3	-96.18	-3.09	0.77	-3	-97.64	-0.32	0.25
ibm10	-1	-100.00	-0.15	0.37	-1	-100.00	0.00	0.03
ibm11	-1	-100.00	-1.14	0.14	-1	-100.00	0.00	0.01
ibm12	-2	-99.22	-2.86	0.32	-3	-100.00	-0.06	0.03
ibm13	-1	-93.33	-3.66	0.32	-2	-100.00	-0.03	0.02
ibm14	-10	-75.48	-1.42	0.54	-10	-76.28	-0.19	0.12
ibm15	-6	-89.66	-4.86	0.42	-7	-94.74	-0.14	0.11
ibm16	-1	-100.00	-0.33	0.67	-1	-100.00	0.00	0.00
ibm17	0	-96.55	-2.19	0.18	-1	-100.00	-0.01	0.01
ibm18	-4	-93.19	-4.98	0.29	-5	-99.88	-0.15	0.08
Average	-3.06	-75.75	-3.73	0.59	-3.39	-86.03	-0.66	0.19

[a]Time = T_D/T_B
[b]Time = $T_C/(T_A + T_B)$

[9] C. M. Fiduccia and R. M. Mattheyses. A linear time heuristic for improving network partitions. In *Proc. of Design Automation Conf.*, pages 175–181, 1982.

[10] R. T. Hadsell and P. H. Madden. Improved global routing through congestion estimation. In *Proc. of Design Automation Conf.*, pages 28–34, June 2003.

[11] H. Hou, J. Hu, and S. S. Sapatnekar. Non-hannan routing. *IEEE Tran. on CAD of Integrated Circuits And Systems*, 18(4):436–444, April 1999.

[12] J. Hu and S. S. Sapatnekar. A survey on multi-net global routing for integrated circuits. *Integration: The VLSI Journal*, 31(1):1–49, November 2001.

[13] D. Jariwala and J. Lillis. On interactions between routing and detailed placement. In *Proc. of Intl. Conf. on CAD*, pages 387–393, November 2004.

[14] R. Kastner, E. Bozorgzadeh, and M. Sarrafzadeh. Pattern routing: Use and theory for increasing predictability and avoiding coupling. *IEEE Tran. on CAD of Integrated Circuits And Systems*, 21(7):777–790, July 2002.

[15] C. Y. Lee. An algorithm for path connections and its applications. *IRE Tran. on Electronic Computers*, EC-10(3):346–365, 1961.

[16] R. Nair. A simple yet effective technique for global routing. *IEEE Tran. on CAD of Integrated Circuits And Systems*, 6(2):165–172, March 1987.

[17] P. Raghvan and C. D. Thompson. Multiterminal global routing: A deterministic approximation. *Algorithmica*, 6:73–82, 1991.