FastRoute: A Step to Integrate Global Routing into Placement

Min Pan and Chris Chu
Department of Electrical and Computer Engineering
Iowa State University, Ames, IA 50011
Email: {panmin, cnchu}@iastate.edu *

ABSTRACT

Because of the increasing dominance of interconnect issues in advanced IC technology, placement has become a critical step in the IC design flow. To get accurate interconnect information during the placement process, it is desirable to incorporate global routing into it. However, previous global routers are computationally expensive. It is impractical to perform global routing repeatedly during placement.

In this paper, we present an extremely fast and high-quality global router called *FastRoute*. In traditional global routing approaches, congestion is not considered during Steiner tree construction. So they have to rely on the time-consuming maze routing technique to eliminate routing congestion. Different from traditional approaches, we proposed a congestion-driven Steiner tree topology generation technique and an edge shifting technique to determine the good Steiner tree topologies and Steiner node positions. Based on the congestion-driven Steiner trees, we only need to apply maze routing to a small percentage of the two-pin nets once to obtain high quality global routing solutions. We also proposed a new cost function based on logistic function to direct the maze routing.

Experimental results show that FastRoute generates less congested solutions in 132× and 64× faster runtimes than the state-of-the-art academic global routers *Labyrinth* [1] and *Chi Dispersion router* [2], respectively. It is even faster than the highly-efficient congestion estimator *FaDGloR* [3]. The promising results make it possible to incorporate global routing directly into placement process without much runtime penalty. This could dramatically improve the placement solution quality. We believe this work will fundamentally change the way the EDA community look at and make use of global routing in the whole design flow.

1. INTRODUCTION

Placement has become a critical step in VLSI design flow. The two major causes are both related to the increasing dominance of interconnect in nanometer-scale IC technologies. First, placement largely determines the performance of a circuit. As feature size in advanced VLSI technology continues to shrink, interconnect delay has become the determining factor of circuit performance. Placement decides the length and hence the delay of interconnect wires to a large extent. Many recent articles reported that interconnect delay can consume as much as 75% of clock cycle in modern designs. Therefore, a good placement solution can substantially improve the performance of a circuit. Second, placement also determines the chip size. Because of the shrinking of device size, the chip area is no longer determined by total cell area, but by the limited routing resources. Extra "white space" is commonly added to provide enough wire tracks to resolve routing congestion. It is typical that more than half of the modern chip is occupied by white space. A good placement needs to allocate white space appropriately to use the chip area effectively.

Because it is very difficult to incorporate circuit delay or routing congestion directly into the placement objective function, timing-driven and congestion-driven placement algorithms typically employ iterative improvement approaches [4] [5] [6]. First, a placement solution is produced. Next, timing/congestion information are estimated based on the current placement. Then the estimated information are fed back to direct the placer to generate a better placement. This process iterates until there is no significant improvement on timing or congestion objective. To estimate timing, interconnect delay is obtained from very rough interconnect models such as half-perimeter of the bounding box or star-model. Due to lack of routing information, it is impossible to get accurate interconnect topology, wirelength, and possible buffer positions and sizes. Hence, interconnect delay cannot be estimated accurately. To estimate routing congestion, previous works proposed generic estimators which aim at predicting the behavior of all routers consistently. However, as we point out in Section 2, routing solutions generated by different routers are very different. Therefore, it is not possible for an estimator to accurately predict congestion of all routers.

In order to get accurate interconnect information during the placement process, it is desirable to incorporate global routing into it. Global routing allocates the routing demand globally over the chip area. It generates interconnect information very close to the final routing implementation and can be used for accurate estimation of interconnect topology, wirelength, delay, congestion, buffering solution, etc. However, due to the high runtime of the traditional global routers, it is impractical to perform global routing repeatedly during placement.

In this work, we develop an extremely fast high-quality global router called *FastRoute*. Experimental results show that *FastRoute* can generate less congested global routing solutions with 132× and 64× speedup over the state-of-the-art academic global routers *Labyrinth* [1] and *Chi Dispersion* router [2], respectively. Very surprisingly, *FastRoute* is even faster than the highly-efficient congestion estimation algorithm *FaDGloR* [3].

The runtime of *FastRoute* is only 1/934 and 1/2229 of the runtime of state-of-the-art academic placers *Capo9.1* [7] and *Dragon3.01* [8], respectively. The promising runtime makes it possible to incorporate global routing directly into the placement process without much runtime penalty. This could dramatically improve the placement solution quality because accurate interconnect information becomes available during the placement process. Note that although we emphasis on the application in placement, we can apply our global router in any early design stage which has the pin

*Acknowledgements: This work was partially supported by the SRC under Task ID 1206 and NSF under grant CCF-0540998.

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

locations fixed, e.g., floorplanning and trial placement in physical synthesis loop. We believe that this work will fundamentally change the way the EDA community look at and make use of global routing in the whole design flow.

FastRoute can achieve superior quality and speed because of the following techniques.

- A congestion-driven Steiner tree topology construction method to distribute routing demand according to the congestion map.
- An edge shifting technique to move the horizontal or vertical tree edges in a Steiner tree from congested regions to less congested regions without changing wirelength of the tree.
- A new cost function based on logistic function to direct the maze routing to find less congested paths.

Traditional global routing approaches do not explore the flexibility of tree structures to resolve routing congestion. They just employ spanning tree or Steiner tree algorithms to construct trees for multi-pin nets. Then the tree structure of each net is fixed and broken into into a set of two-pin nets. After that, they rely on the maze routing to route the two-pin nets to resolve the routing congestion. The global routing runtime is dominated by maze routing. Different from them, we shift the burden of maze routing to Steiner tree construction. We focus on determining good Steiner tree topology and Steiner nodes locations according to congestion information. As a result, the maze routing has a good initial solution to work with and less effort is needed.

The remainder of the paper is organized as follows. In Section 2, we review the previous work in global routing, timing estimation and congestion estimation, as well as the problems with current estimation techniques. In Section 3, we describe the flow of *FastRoute* and explain its underlying idea. Next in Section 4, 5 and 6, we present the three major steps of *FastRoute* in detail. In Section 7, we perform extensive experiments and show the comparison results. Finally, the paper concludes with a summary of results and directions of future work.

2. PREVIOUS WORK AND SOME DISCUSSION

2.1 Global Router

The grid graph model is widely used in global routing [1] [2] [9]. In this model, the chip area is partitioned into rectangular regions called global bins and all the pins in a global bin are assumed to be at the center of the bin. Each global bin corresponds to a node in grid graph. The boundaries of global bins are called global edges, which correpond to the edges in grid graph. The capacity of an edge represents the number of routing tracks for the corresponding boundary. These notions are illustrated in Figure 1. The major optimization objective in global routing is to minimize the total overflow on all the edges in the grid graph. The overflow on a global edge e is defined as how much the routing demand d_e exceeds the edge capacity c_e. If $d_e > c_e$, $overflow_e = d_e - c_e$; otherwise $overflow_e = 0$.

Most academic and industrial global routers [1] [2] first decompose every multi-pin net into a set of two-pin nets by spanning tree or Steiner tree algorithms. After the decomposition, each two-pin net is routed by maze routing. To further improve the solution quality, those routers utilize rip-up and reroute technique. Albrecht [9] proposed a new multi-commodity flow approximation algorithm to solve the global routing problem. The approximation algorithm uses fractional flows. Hence, it is necessary to perform randomized rounding, followed by traditional rip-up and reroute to complete the process.

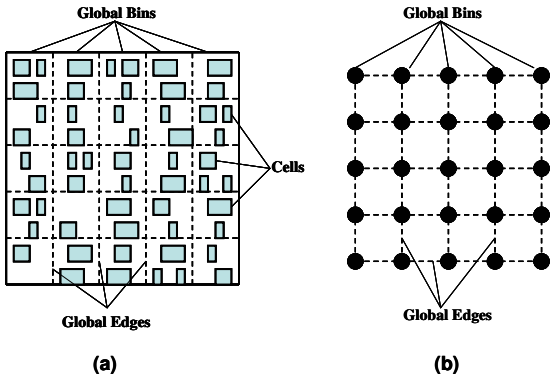

Figure 1: (a) Global bins. (b) Corresponding grid graph.

2.2 Timing Estimation

Previous timing-driven placement algorithms generally employ iterative approaches. For a given placement, the critical path information are obtained by timing analysis. Then the timing information obtained is fed back to the placement engine to generate a new placement solution favoring the critical path. The common methods are adding weight to the critical nets/paths in the objective function [5] [10], adding constraints to the critical nets/paths [4], or adding penalty for the critical nets/paths to the simulated annealing cost function [11] [12]. The basic assumption here is that the timing information obtained are accurate and can be used to direct the placement process.

As interconnect delay becomes the dominant part of circuit delay, accurate interconnect information is needed for timing analysis. However, since there is no routing information, it is impossible to get accurate estimate for interconnect. Early works neglect the interconnect delay in timing analysis. Many recent works [5] [11] [12] employ the half-perimeter of the bounding box to estimate the interconnect length. For multi-pin nets, they first lump all sinks of a net together and assume the lumped sink is driven by the driver through a single wire. The wire length is estimated by the half-perimeter of the bounding box. Hence, they can compute the wire capacitance and resistance using this length. In [10] [13], authors used a star-model to approximate a multi-pin net. An star point is put at the center of gravity of all pins of the net. However, considering the real implementation, multi-pin nets are typically routed as Steiner trees, and global nets are inserted with buffers to minimize the delay. Hence, both half-perimeter of the bounding box model and star-model is far from accurate for interconnect timing estimation.

2.3 Congestion Estimation

Fast congestion estimation is essential for congestion reduction techniques at different stages of the design flow. Post-placement congestion estimation methods try to predict the routing congestion for a given placement. In recent years, a number of probabilistic methods for congestion estimation have been proposed. Lou et al. [14] break mult-pin nets into two-pin wires. Probabilistic usages are then assigned to tiles according to the probability that a two-pin wire will be routed through the tile. Based on the observation that detours are rare, each detour-free path connecting the two pins is assigned an equal probability. Westra et al. [15] and Kahng et al. [16] observed that routes with one or two bends are more likely to occur than multi-bend routes. Consequently, probabilities for the occurence of L-shapes and Z-shapes are emperically derived from industrial designs and are used to improve upon [14]. In [3], Westra et al. presented two congestion estimation tools. The first one, called *pce*, is an implementation of a probabilistic method which is very fast in comparison with other probabilistic methods. The second one, called *FaDGloR*, is new and based on degenerate global routing techniques. Experiments show that *FaDGloR*

is about as fast as *pce*. They concluded that global routing based methods are probably more worthwhile than probabilistic methods in congestion estimation. However, unlike the normal global routers, *FaDGloR* does not generating the feasible global routing solutions that minimize overflow.

We notice that for the same circuit, different routers can give very different routing solutions, hence very different congestion distribution. So we have a basic question - is it possible for a generic congestion estimator to accurately predict the routing congestion for all routers? To answer this question, we investigate the routing solutions generated by two global routers - *Labyrinth* and *Chi Dispersion* router, also the routing solutions generated by *Labyrinth* but using different parameters. For a global edge in the grid graph model, if the routing demand on it is greater than its capacity, we say it is congested. Otherwise, it is uncongested. If a global edge is congested in one routing solution and uncongested in the other, we call it a *congestion mismatch*. The total number of *congestion mismatches* gives the similarity of congestion distribution between two routing solutions. Note that *congestion mismatch* is similar to the "wrongly congested" and "wrongly uncongested" notions in [3]. There, congestion is defined as the ratio of routing demand and capacity. The "wrongly congested" happens if the estimated congestion c is greater than 1.1 but real congestion C is lower than 1.1; the "wrongly uncongested" happens if the estimated congestion c is lower than 0.9 but real congestion C is higher than 0.9. We notice that this metric is not proper. Assume that the estimator simply gives the congestion estimation of $c = 1.0$ over the whole grid graph. In this metric, both the number of "wrongly congested" and "wrongly uncongested" edges are 0. Hence, we propose the *congestion mismatch* as the metric.

We perform the experiments as follows. We use the benchmark circuits provided by the authors of [1]. For each circuit, we generate a routing solution using *Labyrinth* (70% shortest nets use pattern routing) and make it as the standard. Then, we also generate two other routing solutions using *Labyrinth* (50% shortest nets use pattern routing) and *Chi Dispersion* router. We find the number of congestion edges for all three routing solutions, as well as the number of *congestion mismatches* between the standard solution and each of the other two solutions. Table 1 shows the number of congestion edges and the number of *congestion mismatches*. *Lab (70%)* and *Lab (50%)* means the routing solutions generated by *Labyrinth* with 70% and 50% shortest nets pattern routed, respectively. And #Mismatch in *Lab (50%)* and *Chi Dispersion* columns are the number of *congestion mismatch* compared to (*Lab (70%)*).

Table 1: Comparison of number of congestion edges and Congestion Mismatch

	Lab (70%)	Lab (50%)		Chi Dispersion	
	#Con	#Con	#Mismatch	#Con	#Mismatch
ibm01	238	268	398	122	272
ibm02	368	390	580	46	400
ibm03	247	214	367	1	248
ibm04	588	596	662	273	539
ibm06	367	391	596	9	374
ibm07	568	643	887	122	580
ibm08	486	655	865	30	480
ibm09	377	399	638	12	383
ibm10	501	376	691	27	496

From the table we can see that the number of *congestion mismatches* is so significant that it is even more than the number of congestion edges in routing solutions in almost all cases. If we code the congested edge as 1 and uncongested edge as 0, the congestion of a routing solution can be represented as a binary pattern (congestion pattern). The number of *congestion mismatches* of two routing solutions is the Hamming distance [17] between their corresponding congestion patterns. Hamming distance satisfies the triangle inequality: $d_H(x, y) \leq d_H(x, z) + d_H(y, z)$. Assume we use a congestion estimator with the congestion pattern z to estimate the congestion, the numbers of the *congestion mismatches* over the two routing solutions with congestion patterns x and y are $d_H(x, z)$ and $d_H(y, z)$, respectively. From the triangle inequality, we know that the sum of $d_H(x, z)$ and $d_H(y, z)$ is at least $d_H(x, y)$, which is the number of *congestion mismatches* between the two routing solutions. Hence, at least one of $d_H(x, z)$ and $d_H(y, z)$ is bigger than $0.5d_H(x, y)$. Since $d_H(x, y)$ is more than the number of congested edges in either routing solutions, at least for one routing solution, the number of wrongly estimated edges is more than 50% of the number of congested edges in that solution. Therefore, it is impossible for an estimator to claim it can estimate both routing solutions accurately. In fact, the results also show that even using one global router to predict the behavior of another global router (or using one global router with a set of parameters to predict itself with a different set of parameters) is not possible. Therefore, the only possible way to predict congestion accurately is to use the same technique and parameters in both congestion estimation and global routing.

3. OUTLINE OF FASTROUTE

Our goal is to develop a very fast high-quality global router which can be used as both interconnect estimator and traditional routing tool. Hence, we care a lot about the runtime of the router. Maze routing is effective in directing routes away from congested region. However as pointed out by many works (e.g, [1]), maze routing is the major contributor of global routing runtime. If we want to achieve orders of magnitude faster runtime, a lot of maze routing has to be cut down.

As far as we know, previous global routers do not consider the effect of routing tree structures on reducing congestion. RSMT or minimum spanning tree is constructed for each net and broken into two-pin nets. Later, every two-pin net is routed independently without touching the original tree structure. In contrast, our approach focuses mainly on the Steiner tree structures to construct good Steiner trees for better congestion results. The routing demand is allocated by these Steiner trees according to congestion map to alleviate the burden of later maze routing phase.

The main flow of *FastRoute* includes three phases:

1. Congestion map generation.
2. Congestion-driven Steiner tree construction.
3. Routing two-pin nets using pattern routing and maze routing.

In the following sections, we will discuss the three phases in detail.

4. CONGESTION MAP GENERATION

In this section, we will describe how to generate the congestion map in the first phase.

We mentioned in Section 3 that we will construct the Steiner tree according to routing congestion. Hence, before the congestion-driven Steiner tree construction, we need congestion information. Since we are aiming at a very fast global routing algorithm, we need a very fast but fairly good congestion estimation technique.

First, we generate the Steiner minimal trees for all the nets using *FLUTE* [18]. *FLUTE* is a very fast and accurate rectilinear Steiner minimal tree (RSMT) algorithm. It generates optimal RSMT for nets up to degree 9 and is still very accurate for nets up to degree 100, and is much faster than other RSMT techniques. It is very suitable for our application. Second, after generating the Steiner trees, we break all Steiner trees into two-pin nets. For every two-pin net, we assign the demand to the global edges in the grid graph in the following way. If the two pins of a net have the same x coordinates or y coordinates, we assign demand 1.0 to each global edge on the straight line connecting the two pins. If the two pins

of a net have different x and y coordinates, we assume two possible L-shape (sometimes called 1-bend) routings for it - the upper L or lower L. For each edge on the two L-shape routings, we assign demand 0.5 to it. In this way, we get the very first congestion map. Finally, in order to make the congestion map more accurate, we perform a fast rip-up and reroute using L-shaped pattern routing. For each two-pin net, we first remove its routing demand from the congestion map which is added in the second stage. Then we perform routing based on the current congestion map by taking the L-shape which passes through a less congested region. After a full round of L-shaped pattern routing for all the two-pin nets, we get a solution and its corresponding congestion information. We use it as the congestion map for the following congestion-driven Steiner tree construction. Of course, we can use maze routing here, but it will consume a lot of runtime. Since we will change the Steiner tree structures later, it is not worthy to spend the time to perform maze routing in this phase.

5. CONGESTION-DRIVEN STEINER TREE CONSTRUCTION

In this section, we focus on the Steiner tree structures to alleviate routing congestion. This is the key phase in the whole flow of *FastRoute*. First, we discribe the two major techniques, *Congestion-driven Topology Generation* in Section 5.1, and *Edge Shifting* in Section 5.2. Then the flow for the congestion-driven Steiner tree construction phase (phase 2) is given in Section 5.3.

5.1 Congestion-driven Topology Generation

There is a lot of research on Steiner tree problem. Previous works in global routing apply RSMT algorithms to find Steiner trees to minimize routing tree length. However, our goal is to construct the Steiner tree in favor of congestion reduction.

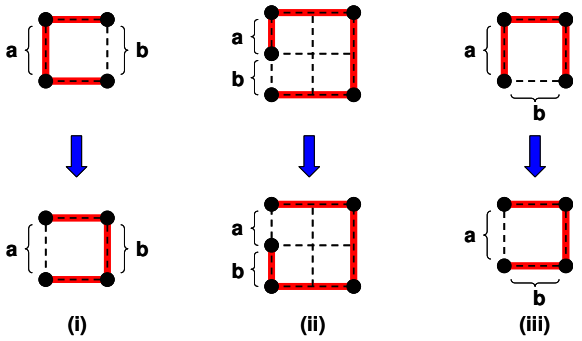

Figure 2: Three ways of reallocating routing demand.

Routing congestion happens when there is more routing demand than the capacity of global edges. Global routing essentially allocates routing demand over the global edges. The total routing demand of a net is its routing tree length. If a net routed with minimum wirelength uses a congested edge, we have no way to simply eliminate the routing demand on that edge. We have to reallocate it to some other global edges. Without loss of generality, assume a vertical global edge a is congested. There are three ways to reallocate some routing demand on a. (1) Reallocate the demand to another vertical global edge in the same row as a. For example, in Figure 2(i), global edge b is used instead of a. (2) Reallocate the demand to another vertical global edge not in the same row as a. For example, in Figure 2(ii), global edge b is used instead of a. (3) Reallocate the demand to a horizontal edge. For example, in Figure 2(iii), global edge b is used instead of a.

We observe that the widely used pattern routing and maze routing are applying the first way only. For example, in Figure 3, the route from X to Y (solid line) goes through a congested global edge a. To avoid congestion, we can take an alternative route (dashed line) to reallocate the demand to b from a. However, we also need to reallocate the demand from c to d, and from e to f at the same time. Notice that pattern routing and maze routing are not able to reduce the routing demand on any row of vertical global edges or any column of horizontal global edges. On the other hand, ways (2) and (3) can help. Way (2) could move the demand in a specific row (column) of global edges to another row (column) of global edges. Way (3) could transfer the demand from one direction to the other direction.

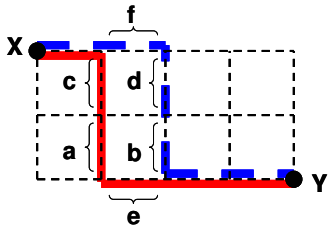

Figure 3: Pattern/maze routing example.

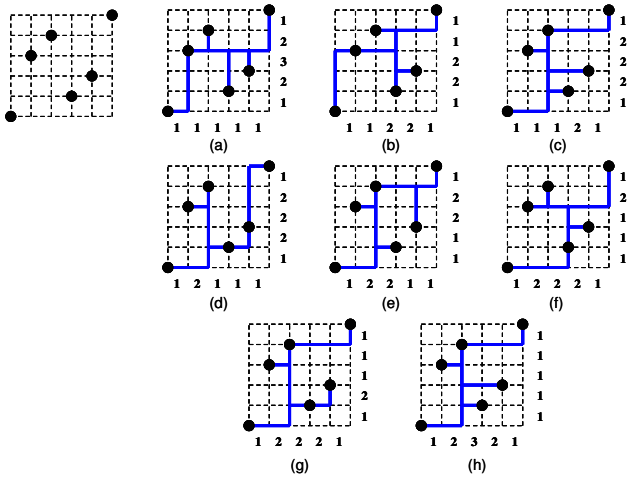

Figure 4: Different Steiner trees topologies.

One important observation we make is that Steiner tree topologies can supply a lot of flexibility in avoiding routing congestion by applying way (2) and (3). For a multi-pin net, there are many different Steiner tree topologies to connect all the pins in the net. Each topology corresponds to some specific routing demand distribution. We notice that different topologies can have very different routing demand in two directions and in different rows/columns of global edges. For example, in Figure 4, we show 8 minimal wirelength Steiner tree topologies for a 6-pin net. For each topology, we only show one of the possible embeddings on the routing grids. The number below each column of global edges is the routing demand over all the horizontal global edges in that column. The number right to each row of global edges is the routing demand over all the vertical global edges in that row. It is clear that although all these Steiner trees have the same wirelength, they have very different routing demand distribution, hence very different congestion results. Therefore, we can make use of this flexibility in topology and try to find good topology for each net in terms of congestion metric. For example, for the net shown in Figure 4, if it is congested in horizontal direction, we want to pick topology (a) which has less routing demand in horizontal direction. On the contrary, if it is congested in vertical direction, (h) would be the best choice. This applies way (3) of reallocating demand. In addition to transferring routing demand between two directions, way (2) of reallocating demand is also enabled by changing topology. Comparing topology (b) with (e), instead of having more routing demand in the 2nd row (from left) and 2nd column (from top)

of global edges as in (e), topology (b) have more routing demand in the 4th row and 4th column of global edges. So whether use topology (b) or (e) depends on the congestion of these rows and columns of global edges.

With this flexibility of topology in mind, our main idea is to construct good Steiner tree topology for each net according to the congestion map. We encourage to use the topology with less routing demand in the congested direction, and also less routing demand in the congested regions. To achieve this goal, we construct Steiner tree topologies as follows. First, we define the row/column region between two Hanan grid lines [19] for a net as the rectangular region between the two grid lines and the bounding rectangle of the net. As illustrated in Figure 5, the shaded region in (a) is the row region between the Hanan grid lines G_{H1} and G_{H2}, and the distance between G_{H1} and G_{H2} is $v2$. Similarly, the shaded region in (b) is the column region between the Hanan grid lines G_{V1} and G_{V2}, and the distance between G_{V1} and G_{V2} is $h2$. For each row/column region between two hanan grid lines of the original net, we compute its corresponding "average congestion" (we will discribe how to compute it in detail later). Then, the distance between the corresponding two hanan grid lines is scaled proportional to the "average congestion". We use these scaled distances instead of their original distances to measure the routing tree wirelength. Hence, we transform the congestion-driven Steiner tree problem into a RSMT problem in scaled wirelength measure. Finally, we apply *FLUTE* to find the RSMT topology in terms of this scaled wirelength. This topology with minimal scaled wirelength leads to the best congestion result. In this way, we maintain a balance between wirelength and congestion when constructing the Steiner tree rather than just minimize wirelength.

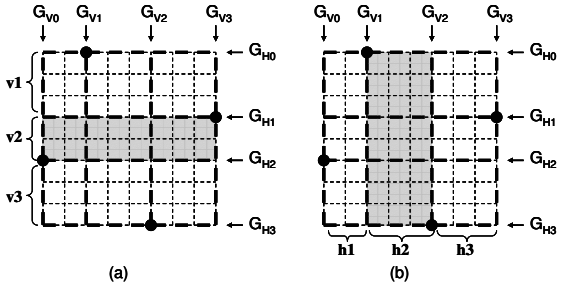

Figure 5: (a) The row region between G_{V1} and G_{V2}. (b) The column region between G_{H1} and G_{H2}.

So far we have presented the general flow to find a good topology. Now we describe what the "average congestion" for a row/column region is and how to compute it. For a row/column region between two Hanan grid lines, if it is congested in vertical/horizontal global edges, we discourage to use the segments across the region in the direction perpendicular to the two Hanan grid lines. Hence, we scale up the distance between the two Hanan grid lines. The scaling factor we use is the "average congestion". For a row/column region, it is defined as the ratio between the total demand and total capacity on all vertical/horizontal global edges in the corresponding row/column region. "Average congestion" indicates on average how congested a row/column region is. For example, in Figure 5 (a), "average congestion" for the shaded row region is computed as the total demand divided by the total capacity on all vertical global edges in the region. Note that we are not just considering the global edges on Hanan grid, but all the global edges in this region because all these global edges are possibly used by our Steiner trees. In this technique, we only try to control the frequency to use different segments between Hanan grid lines in the topology but not the exact position of these segments in the Steiner tree. In fact, it is not necessary to specify the position of segments here. After we fix the Steiner tree topology in this phase, the segments still have a lot of flexibility to change locations. Hence, what we want is the "average" congestion for a row/column region instead of congestion on some specific global edges.

Finally, we want to point out that this congestion-driven Steiner tree construction technique has great impact on the routing solution quality. It explores the solution space out of the scope of pattern routing and maze routing.

5.2 Edge Shifting

In Section 5.1, we present the congestion-driven Steiner tree topology construction technique. The topology only specify the connections between the pins and Steiner nodes for the net. After fixing the topology, there is still flexibility left for congestion optimization. For example, in Figure 6, we focus on the bold edge in the Steiner tree. With different congestion scenarios, the edge should be shifted to different positions to avoid congestion.

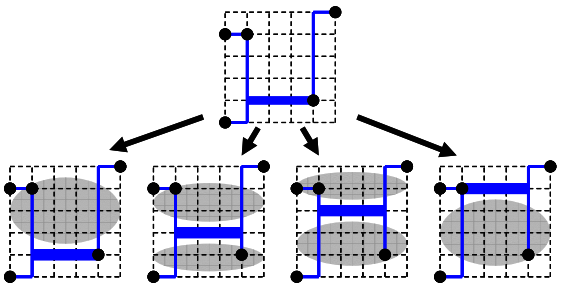

Figure 6: Edge Shifting for less Congestion
Bottom four cases: shaded regions are congested.

If possible, we want to move tree edges out of congested regions without increasing Steiner tree wirelength. The reason is that the total wirelength is related to the overall congestion. If the total wirelength is more, it is very likely to have more overall congestion. We observe that if the two pins of a horizontal or vertical tree edge are both Steiner nodes, we can shift this edge freely within a "safe range" without increasing the Steiner tree length. In order to find the "safe range", for a horizontal/vertical edge between a pair of Steiner nodes S_1 and S_2, we define the "sliding range" as the range of y/x coordinates so that S_1 and S_2 will not pass any node (including pins and Steiner nodes) in the tree when shifting the tree edge S_1-S_2. As illustrated in Figure 7, the "sliding range" of (a) a horizontal edge, or (b) a vertical edge S_1-S_2 is R_{12}. We only consider shifting edge S_1-S_2 when both S_1 and S_2 have degree 3. A Steiner node can only have degree 3 or 4. For any degree 4 Steiner node, we can break it into two connected degree 3 Steiner nodes. The way to get this "sliding range" is as follows. We consider the two neighbors for S_1/S_2 which are not S_2/S_1. If S_1-S_2 is horizontal, the range for safely sliding S_1-S_2 is between the y coordinates of two neighbor nodes (R_1 and R_2 in Figure 7(a)). Otherwise, the range for safely sliding S_1-S_2 is between the x coordinates of two neighbor nodes (R_1 and R_2 in Figure 7(b)). The "sliding range" of S_1-S_2 is the common part of R_1 and R_2, which is R_{12} in Figure 7. In R_{12}, the edge S_1-S_2 can be shifted freely without changing tree length.

We want to point out that the "sliding range" may not always be the "safe range". Sometimes, it is just part of the "safe range". For example, in Figure 8(a), the "sliding range" for edge S_1-S_2 is R_{12}. Hence, S_1-S_2 can be shifted at most to the same y grid as Steiner node S_3. But we notice that S_1-S_2 can be shifted higher than S_3 without changing the Steiner tree length. The only problem here is that the tree topology needs to be changed. This happens when two Steiner nodes S_2 and S_3 overlap with each other (as illustrated in Figure 8(b)). In this case, we will exchange the two Steiner nodes S_2 and S_3 to enable further shifting, which is shown in Figure 8(c). Notice that by exchanging S_2 and S_3, we change the *topology*1 into *topology*2. In Figure 8(c), the "sliding range" for *topology*2 is R_{13}. The full "safe range" is R_{123}, which is the sum of R_{12} and R_{13}. Therefore, now we can explore the full "safe range" R_{123} for S_1-S_2.

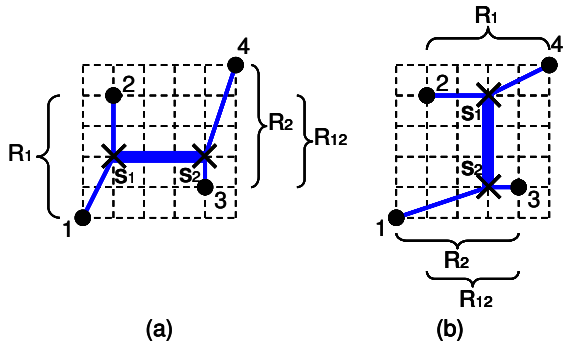

Figure 7: "Sliding range" for edge S_1-S_2.

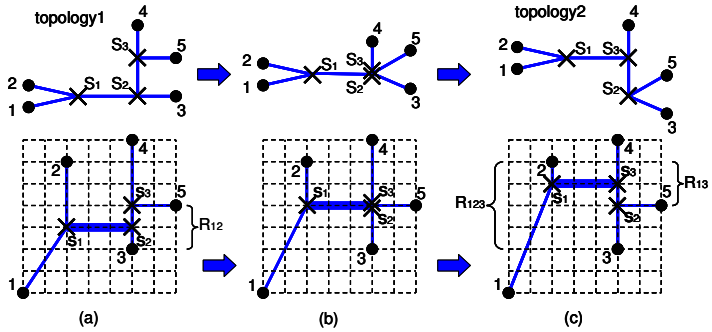

Figure 8: Modification of tree topology during edge shifting.

After we find the "safe range" for an edge S_1-S_2, we need to decide the best position for it within the "safe range". The criterion for the best position is that the total demand of all the global edges on the Steiner tree is minimized. We define this total demand as the cost of the tree. Hence, for every possible position, we can evaluate this cost for the tree. Note that we only need to evaluate the demand on the global edges affected by shifting S_1-S_2, because other global edges will not be affected. So the edges need to be considered are all edges E adjacent to S_1 and S_2. Note that some edge $e \in E$, could be a diagonal edge (e.g., edge 1-S_1 in Figure 7(a)). We do not know which global edges this tree edge will use. In this case, we consider the two possible L-shape route for it and pick the one that results in smaller cost. The reason is that for these diagonal edges, later routing stages will try to minimize the total demand of global edges on their routing path. Of course, instead of considering two L-shape routes, we can consider Z-shape route or even maze route. It is a tradeoff between accuracy and runtime. Since the Steiner tree structures keep on changing at this stage, it is not necessary to consider the route too accurately.

The way to shift one tree edge is described above. For each Steiner tree, the algorithm to perform edge shifting is as follows. We find all the horizontal and vertical tree edges between two Steiner nodes in the Steiner tree. Next, we compute the "safe range" R_{12} for each tree edge S_1-S_2 (including the expanded range by exchanging Steiner nodes). Then the cost of the tree is evaluated for every possible position of S_1-S_2 within the "safe range". Finally, the tree edge S_1-S_2 and its position with the minimal cost is chosen and S_1-S_2 is shifted to that position. We iteratively apply this process until we cannot find a tree edge for shifting to further reduce the cost.

After *Edge Shifting*, the positions of Steiner nodes are fixed and the only flexibility left is how to route each tree edges.

5.3 Phase 2 Flow

Section 5.1 and 5.2 give the details of the techniques to construct good Steiner tree structure for a net. In this part, we present the flow of phase 2 to generate Steiner trees for all the nets.

We go through every net in the order in the netlist file. For each net N, we first remove its routing demand from the congestion map. Second, the Steiner tree topoloy for N is constructed as in Section 5.1. Then, we apply edge shifting technique in Section 5.2 to further reduce the congestion. After Steiner tree structures are fixed, we route all the tree edges using L-shaped pattern routing. Finally, we add new routing demand by N to the congestion map.

6. PATTERN ROUTING AND MAZE ROUTING

After the congestion-driven Steiner tree construction phase, we find good Steiner tree structures for the nets. Then all routing trees are broken into tree edges (two-pin nets). In the routing phase, we route all two-pin nets by pattern routing and maze routing.

We first apply pure pattern routing to route all the two-pin nets once. Pattern routing is to use predefined patterns to route two-pin nets. The most commonly used are L-shaped (1-bend) or Z-shaped (2-bends) patterns. Pattern routing has much better runtime complexity over maze routing. The effect of pattern routing is investigated extensively in [1]. Here, we use the Z-shaped pattern. It has more flexibility than L-shaped pattern and much faster than maze routing. In fact, in the congestion-driven Steiner tree construction phase, we already perform L-shaped routing when we update the congestion map after constructing Steiner tree for a net.

After the pattern routing, we apply rip-up and reroute using maze routing, which is similar to other works. Many recent global routers [1] [9] have routing cost which increases abruptly when the demand on a global edge reaches the edge capacity (Figure 9(a)). In [2], the routing cost function for maze routing is discussed and a piece-wise cost function is proposed. A unit cost is assigned to a global edge until it reaches a certain percentage below capacity, and cost is increased linearly until it reaches a certain percentage above capacity (Figure 9(b)). Instead, we employ a logistic function [20] in equation (1) as our cost function (Figure 9(c)). h and k are function parameters.

$$cost = 1 + \frac{h}{1 + e^{-k(demand - capacity)}} \tag{1}$$

The reason for us to use such an function is that we want the cost to increase dramatically around the capacity but mildly in the under-capacity and over-capacity part. The idea behind this is to differentiate the slope of cost function in different parts. If demand on a global edge is much lower than capacity, we do not need to differentiate different demand values, e.g., if the capacity is 10, the difference in cost for demand 2 and 3 should be small. Similarly, if demand on a global edge is much higher than capacity, we do not need to charge very different cost for different demand values, either, e.g., demand 20 or 25 should not make significant difference when capacity is 10. However, if demand on a global edge is close to capacity, the change on demand make significant difference because the edge could become over capacity from within capacity, or from within capacity to over capacity. In this way, we focus more on the global edges with demand close to capacity.

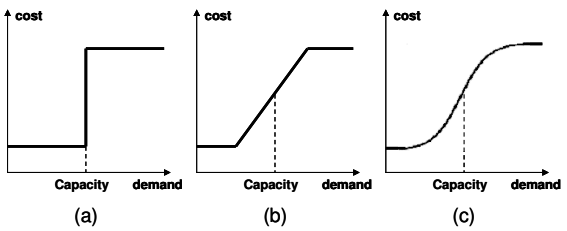

Figure 9: (a) Abrupt cost, (b) Linear cost, (c) Logistic cost.

For *FastRoute* default mode, we only do one round of maze routing and on average only 2.15% of nets are really routed by maze routing (others use pattern routing). This is the major reason that our algorithm is so fast. Moreover, we can get better total overflow than other global routers although we do much less maze routing.

We attribute this to the high-quality Steiner tree structures generated by the second phase. Maze router has a very good starting solution to work with.

7. EXPERIMENTAL RESULTS

In this section, we present our experimental results. All experiments were performed on a Linux workstation with Intel Pentium 4 3.0 GHz CPU and 2GB memory.

First, we compare *FastRoute* with two state-of-the-art academic global routers: *Labyrinth* [1] and *Chi Dispersion* router [2]. We use the same benchmarks as in [2] provided by the authors of [1]. For *Labyrinth*, 70% of the shortest connections are routed by pattern routing, which is the same as in [2]. We measure wirelength and total overflow in the manner suggested by the authors of both papers. The results are summarized in Table 2. The total overflow and wirelength of *FastRoute* is less than both *Labyrinth* and *Chi Dispersion* router. At the same time, *FastRoute* is 132× and 64× faster than *Labyrinth* and *Chi Dispersion* router, respectively. Because we cannot find a version to duplicate the results in [2], the runtime of *Chi Dispersion* router is scaled from the runtime in [2] based on the information from Standard Performance Evaluation Corporation (SPEC) [21]. In [2], it was claimed that runtime of *Chi Dispersion* router is roughly 2× faster than *Labyrinth*, which coincides with the scaled runtime we obtained. We also get a new version of *Chi Dispersion* router from the authors of [2], the total overflow on the same set of benchmark is 804, but the total runtime is 917 seconds which is close to the runtime of *Labyrinth*. We also have a beaver mode for *FastRoute*. It performs several rounds of rip-up and reroute to achieve lower overflow. It can cut down the total overflow by half with 2.2× runtime of the default mode.

Table 3: Effect of Congestion-driven Steiner tree topology construction, Edge shifting and Logistic cost function.

	FastRoute Overflow	w/o StTree Overflow	w/o Edgeshift Overflow	Linear Cost Overflow
ibm01	250	283	323	297
ibm02	39	114	57	108
ibm03	1	5	1	30
ibm04	567	672	666	606
ibm06	33	71	85	129
ibm07	18	178	54	174
ibm08	58	91	89	126
ibm09	28	74	89	113
ibm10	18	39	38	220
Total	1012	1527	1402	1803
Norm*	**1**	**1.51**	**1.39**	**1.78**

(*) Normalized to FastRoute total overflow.

Second, we investigate the effect of three main techniques in *FastRoute*: congestion-driven Steiner tree construction, edge shifting and logistic cost function for maze routing. We disable the three techniques from *FastRoute* one by one and compare the final total overflow with *FastRoute*. For the logistic cost function, we substitute it with a linear cost function proposed in [2] and tried to tune the parameters in the linear cost function to get results as good as possible. From Table 3, the total overflow are increased by 51%, 39% and 78% without the three techniques, respectively. It is clear that all of them contribute to the high quality of *FastRoute*.

Third, we show the runtime breakdown for *FastRoute* default mode. As shown in Table 4, the three phases in *FastRoute*: congestion map generation, congestion-driven Steiner tree topology construction, and two-pin nets routing account for 14.4%, 27.5% and 58.1% of the total runtime, respectively. In addition, maze routing in two-pin nets routing is still the most time-consuming part (48% of total runtime) although on average only 2.15% two-pin nets are routed using maze routing. Consider that *Labyrinth* apply maze routing on 30% of the nets and do many rounds of rip-up and reroute. That is why *FastRoute* can be two orders faster.

Table 4: Runtime breakdown for FastRoute.

	Cong Map	Steiner Tree	Route two-pin nets	
			Pattern Route	Maze Route
ibm01	14.3%	23.8%	4.8%	57.1%
ibm02	12.5%	25.0%	7.1%	55.4%
ibm03	13.6%	27.3%	11.4%	47.7%
ibm04	14.0%	22.0%	12.0%	52.0%
ibm06	13.2%	30.8%	8.8%	47.3%
ibm07	16.3%	28.8%	12.5%	42.3%
ibm08	17.2%	36.9%	11.5%	34.4%
ibm09	14.1%	24.6%	12.0%	49.3%
ibm10	14.1%	28.1%	11.1%	46.7%
avg	**14.4%**	**27.5%**	**10.1%**	**48.0%**

Table 5: FastRoute and FaDGloR Runtime Comparison.

	FastRoute	FaDGloR[1]	FastRoute(-rsmt)	FaDGloR(-rsmt)
ibm01	0.21	0.71	0.20	0.17
ibm02	0.56	2.18	0.52	0.45
ibm03	0.43	1.36	0.41	0.46
ibm04	0.50	1.54	0.47	0.48
ibm06	0.91	2.27	0.86	0.74
ibm07	1.05	3.08	0.99	1.01
ibm08	1.16	4.35	1.07	1.13
ibm09	1.39	4.31	1.32	1.86
ibm10	1.98	5.86	1.88	2.49
Total	8.19	25.66	7.72	8.79
Norm	**1**	**3.13**[2]	**1**	**1.14**[3]

The unit for all runtime (except Normalized) is second.
1. This runtime include file I/O and result checking time, 2. normalized to full FastRoute runtime, 3. normalized to FastRoute(-rmst) runtime.

Fourth, we compare the runtime of *FastRoute* and an efficient congestion estimator *FaDGloR*. In [3], the authors claimed *FaDGloR* is as fast as probabilistic congestion estimators. *FaDGloR* reports two runtime, "total runtime" (the total runtime including Steiner tree construction, decomposition, routing, file I/O, and result checking) and "route time" (the actual routing time for all two-pin nets). Hence, we do two type of comparison here. First, we compare the *FastRoute* total runtime with the "total runtime" of *FaDGloR*. Table 5 shows that *FastRoute* is about 3.13× faster than *FaDGloR*. Since we should exclude the file I/O and result checking parts from *FaDGloR* "total runtime", the real speedup should be around 3×. Considering their Steiner tree construction algorithm is much slower than FLUTE, we perform a second comparison. We report the *FastRoute* runtime excluding the Steiner tree construction in phase 1 (FastRoute(-rsmt)) and compare it with the "route time" of *FaDGloR* (FaDGloR(-rsmt)). *FastRoute* is still 14% faster than *FaDGloR* for the routing time and has better scalability. But note that *FastRoute* generates high-quality global routing solutions while *FaDGloR* only gives congestion estimation.

Fifth, we also run state-of-the-art placers *Capo9.1* [7] and *Dragon 3.01* [8] on the placement benchmarks from which the global routing benchmarks are generated. Table 6 show that *FastRoute* runtime is only about 1/934 and 1/2229 of the runtime of *Capo9.1* and *Dragon3.01*. This means we can run *FastRoute* hundreds of times inside placers without much runtime penalty.

8. CONCLUSIONS

In this paper, we develop an extremely fast high-quality global router - *FastRoute*. It generates less congested routing solutions

Table 2: Comparion of FastRoute, Labyrinth and Chi Dispersion router.

	FastRoute			FastRoute (Beaver mode)			Labyrinth Predictable router			*Chi* Dispersion router		
	Overflow	Wirelen	Time(s)	Overflow	Wirelen	Time(s)	Overflow	Wirelen	Time(s)	Overflow	Wirelen	Time(s)
ibm01	250	67128	0.21	159	68436	0.72	242	76228	16.99	189	66005	8.63
ibm02	39	179995	0.56	3	180139	1.16	214	202235	26.53	64	178892	26.27
ibm03	1	151023	0.43	1	151023	0.43	117	191500	37.92	10	152392	24.71
ibm04	567	172593	0.50	300	175219	2.30	786	198181	80.95	465	173241	32.94
ibm06	33	285882	0.91	7	287870	1.71	130	339379	72.06	35	289276	53.33
ibm07	18	376835	1.05	2	379989	1.99	407	450855	168.41	309	378994	79.61
ibm08	58	412915	1.16	17	414909	3.17	352	466556	154.82	74	415285	72.94
ibm09	28	426471	1.39	22	428803	2.75	310	481841	229.59	52	427556	86.67
ibm10	18	599433	1.98	1	600321	3.80	288	680113	296.70	73	599937	139.61
Total	1012	2672275	8.19	512	2686709	18.03	2846	3086888	1083.97	1271	2681578	524.71
Norm*	**1**	**1**	**1**	**0.506**	**1.005**	**2.201**	**2.812**	**1.155**	**132**	**1.256**	**1.003**	**64**

(*) Normalized to FastRoute results.

Table 6: Runtime comparison with Placers.

	FastRoute Time(s)	Capo Time(s)	Dragon Time(s)
ibm01	0.21	126	778
ibm02	0.56	280	663
ibm03	0.43	338	633
ibm04	0.50	456	1234
ibm06	0.91	666	1392
ibm07	1.05	1145	1904
ibm08	1.16	1277	4163
ibm09	1.39	1329	3953
ibm10	1.98	2035	3537
Total	8.19	7652	18257
Norm	**1**	**934**	**2229**

and is 132× and 64× faster than *Labyrinth* and *Chi Dispersion* router. This makes it possible to integrate the global router into placement to get accurate interconnect information to direct the placement process. Our future work will focus on two aspects. First, we will further improve the runtime and quality of *FastRoute*. Second, we will integrate *FastRoute* into placement framework to develop placement algorithms that generate better solutions in terms of timing, routability, etc.

Acknowledgment

The authors would like to thank Prof. Patrick Madden from SUNY Binghamton for help with *Chi Dispersion* router, and Dr. Jurjen Westra from Eindhoven University of Technology for help with *FaDGloR*.

9. REFERENCES

[1] R. Kastner, E. Bozogzadeh, and M. Sarrafzadeh. Predictable routing. In *Proc. IEEE/ACM Intl. Conf. Computer-Aided Design*, pp. 110-113, 2000.

[2] R. T. Hadsell and P. H. Madden. Improved global routing through congestion estimation. In *Proc. ACM/IEEE Design Automation Conf.*, pp. 28-31, 2003.

[3] J. Westra and P. Groeneveld. Is probabilistic congestion estimation worthwhile? In *Proc. Intl. Workshop on System-Level Interconnect Prediction*, pp. 99-106, 2005.

[4] B. Halpin, C. Y. R. Chen and N. Sehgal. Timing driven placement using physical net constraints. In *Proc. ACM/IEEE Design Automation Conf.*, pp. 780-783, 2001.

[5] H. Eisenmann and F. M. Johannes. Generic global placement and floorplanning. In *Proc. ACM/IEEE Design Automation Conf.*, pp. 269-274, 1998.

[6] U. Brenner, A. Rohe. An effective congestion-driven placement framework. In *IEEE Trans. on Computer-Aided Design, vol. 22(4)*, pp. 387-394, 2003.

[7] A. E. Caldwell, A. B. Kahng, and I. L. Markov. Can recursive bisection produce routable placements? In *Proc. ACM/IEEE Design Automation Conf.*, pp. 477-482, 2000.

[8] M. Wang, X. Yang, and M. Sarrafzadeh. Dragon2000: Standard-cell placement tool for large industry circuits. In *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*, pp. 260-263, 2000.

[9] C.Albrecht. Global routing by new approximation algorithms for multicommodity flow. In *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 20:622-631, 2001.

[10] B. M. Riess, and G. G. Ettelt. SPEED: fast and efficient timing driven placement. *Proc. Intl. Symp. on Circuits and Systems*, pp. 377-380, 1995.

[11] W. Swartz and C. Sechen. Timing Driven Placement for Large Standard Cell Circuits. In *Proc. ACM/IEEE Design Automation Conf.*, pp. 211-215, 1995.

[12] X. Yang, B.-K. Choi, and M. Sarrafzadeh. Timing-driven placement using design hierarchy guided constraint generation. In *Proc. IEEE/ACM Intl. Conf. Computer-Aided Design*, pp. 177-184, 2002.

[13] G. Stem, B. M. Riess, B. Rohfleisch and F. M. Johannes. Timing driven placement in interaction with netlist transformations. *Proc. Intl. Symp. on Physical Design*, pp. 36-41, 1997.

[14] J. Lou, S. Krishnamoorthy, and H. Sheng. Estimating routing congestion using probabilistic analysis. In *Proc. Intl. Symp. on Physical Design*, pp. 112-117, 2001.

[15] J. Westra, C. Bartels, and P. Groeneveld. Probabilistic congestion prediction. In *Proc. Intl. Symp. on Physical Design*, pp. 204-209, 2004.

[16] A. B. Kahng and X. Xu. Accurate pseudo-constructive wirelength and congestion estimation. In *Proc. Intl. Workshop on System-Level Interconnect Prediction(SLIP)*, pp. 61-68, 2003.

[17] R. W. Hamming. Error-detecting and error-correcting codes. *Bell System Technical Journal*, 29(2):147-160, 1950.

[18] C. Chu, Y. Wong. Fast and Accurate Rectilinear Steiner Minimal Tree Algorithm for VLSI Design. In *Proc. Intl. Symp. on Physical Design*, pages 28-35, 2005.

[19] M. Hanan. On Steiner's problem with rectilinear distance. *SIAM Journal of Applied Mathematics*, 14:255-265, 1966.

[20] D. von Seggern. CRC Standard Curves and Surfaces. Boca Raton, FL: CRC Press, 1993.

[21] http://www.spec.org/