Maze Routing Steiner Trees With Delay Versus Wire Length Tradeoff

Renato Hentschke, Jaganathan Narasimhan, Marcelo Johann, *Member, IEEE*, and Ricardo Reis, *Senior Member, IEEE*

***Abstract*—In this paper, we address the problem of generating good topologies of rectilinear Steiner trees using path search algorithms. Various techniques have been applied in order to achieve acceptable run times on a Maze Router that builds Steiner trees. A biasing technique proposed for wire length improvement, produces trees that are within 2% from optimal topologies in average. By introducing a sharing factor and a path-length factor we show how to trade-off wire length for delay. Experimental results show that our algorithm generates topologies with better delay compared to state of the art heuristics for Steiner trees, such as AHHK (from 26% to 40%) and P-Trees (from 1% to 30% and from 6% to 21% in the presence of blockages) while keeping the properties of a routing algorithm. An important motivation for this work lies in the fact that it can be used for estimation in the early stages as well as for actual routing, thereby improving the convergence and timing closure of the design significantly. We also provide some valuable theoretical background and insights on delay optimization and on how it relates to our maze router implementation.**

***Index Terms*—Delay, maze search, routing, Steiner trees.**

I. Introduction

WIRE LENGTH and wire delay are very important issues in the chip design and need to be considered early in synthesis and physical design algorithms. In traditional design flows, early wiring estimates obtained from physical design are often used in many synthesis iterations in order to achieve timing closure. Finally global and detailed routing are also performed targeting wire length (wl) reduction thereby improving delays to critical sinks. Convergence of the optimization process is affected by the quality of the wiring estimates, placement algorithms, routing algorithms and their ability to optimize delay to the critical elements of the circuit while keeping the overall design routable. Convergence is affected based upon how well estimates made in the early stages match values obtained in later steps.

Manuscript received February 16, 2007; revised August 24, 2007. First published May 19, 2009; current version published July 22, 2009. This work was supported in part by CNPq and IBM. There is an IBM patent pending on this work.

R. Hentschke, M. Johann, and R. Reis are with Universidade Federal do Rio Grande do Sul (UFRGS), Instituto de Informatica, CP 15064-CEP 91501, Porto Alegre, Brazil (e-mail: renato@inf.ufrgs.br; johann@inf.ufrgs.br; reis@inf.ufrgs.br).

J. Narasimhan is with IBM, International Business Machine, T. J. Watson Research Center, Yorktown Heights, NY 10598 USA (e-mail: jagan@us.ibm.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2009.2019798

The most commonly used algorithms for routing are based on maze routing [1]. They are well known and are characterized by: 1) having high flexibility and ability to handle blockages, constraints and congestion and 2) producing optimal point-to-point paths and good tree topologies.

Steiner trees can be used for early estimation because, as they account for the individual positions of each pin and represent actual connected sets of paths, they are better than simple half-perimeter measures. Steiner tree algorithms are also commonly used in Global Routing to actually define global routing solutions for each net. In particular, minimum length rectilinear Steiner trees (MRSTs) represent very good routes in terms of wire length. However, wiring topologies strongly affect delay to the critical sinks and improperly designed MRSTs could lead to very high delays [2]. Even small changes in topology may significantly affect delay to sensitive sinks. Additionally, most Steiner tree algorithms are not flexible enough to handle blockages, congestion, and actual routing constraints.

Somewhere in the design flow one has to either decompose the net into a set of point-to-point connections or to leave the Steiner solution aside and let a routing algorithm find its own solution for the whole set of pins. Both approaches have their drawbacks. By decomposing the net we stop seeing the original problem; we can get stuck with inflexible Steiner point specifications or lose the ability to recognize better solutions if anything changes at this step. On the other hand, if we just abandon the early Steiner solution and use a multi-pin router, its solution may end up completely different from what was previously considered. Hence, in both cases early estimates used in logic design may not reflect the actual routes produced finally.

In this work, we take a slightly different approach. We extended a path search routing algorithm by including dedicated techniques into it, some old (heuristic search, Hannan grid, etc.) and some completely new, presented in Sections V, VI-B, and VI-C, so that it acquired the ability to find very good Steiner trees. An important motivation for that lies in the fact that the same algorithm can be used for estimation in the early stages as well as for actual routing. It is expected that the adoption of the same algorithm will result in routes that better match previous estimations, but even in the scenario where they do not match because the final context have changed, the routing algorithm still has the same power to understand and tradeoff tree topologies for wire length or delay optimization.

Various techniques are applied in order to achieve acceptable run times on a Maze Router for Steiner trees. The main contribution of this paper is to show how to improve an industrial standard routing algorithm by a selection of techniques so that it can

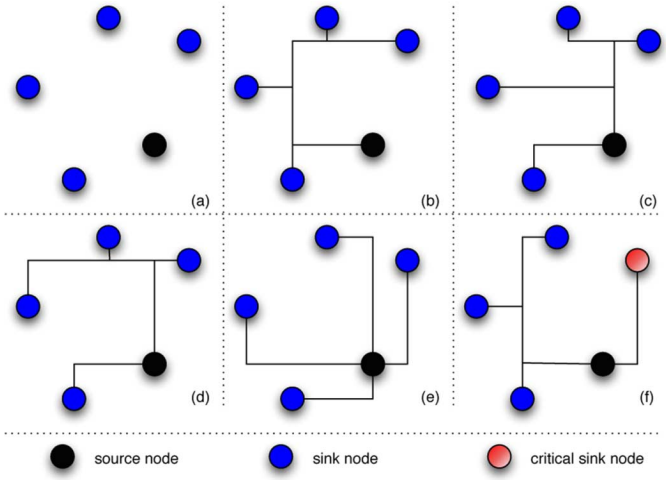


Fig. 1. Steiner topologies for delay optimization; (a) net; (b) minimum Steiner tree (MST); (c) minimum arborescence (MSA); (d) intermediate topology between (b) and (c) bounded radius Steiner tree (BRST); (e) star topology; (f) critical sink approach (CSA).

beat the quality of state-of-the-art heuristic Steiner tree methods and provide flexibility to tradeoff wire length and delay, while keeping the properties of a routing algorithm and exhibiting fast runtimes. The preliminary version of this paper was presented in ISPD 2007 [3].

II. Review of Basic Methods

A. Topologies for Delay and Wire Length Tradeoff

Delay and wire length are strongly related to Steiner tree topologies. Various kinds of rectilinear Steiner tree topologies are shown for a net in Fig. 1. Fig. 1(b) shows a minimum length Steiner tree (MRST), which minimizes wire length but may present high delay to nodes that accidentally get connected far from the driver. Fig. 1(c) shows a minimal Steiner arborescence (MSA) [4], [2] or shortest path tree (SPT) which is a tree with shortest paths from the source to any sink. Such a tree minimizes source to sink distances at the expense of total wire length but not necessarily minimize overall delay, affected by its increased wire length and capacitance sharing along the paths. Fig. 1(d) shows a bounded radius Steiner tree (BRST) [5]–[7] in which the maximum distance from the source to any sink is bounded, exhibiting a compromise between MRSTs and SPTs. The star tree topology shown in Fig. 1(e) has separate wires for each sink, resulting in optimal path length and minimum sharing, but possibly huge total wire length and therefore not optimal delay, depending on technology parameters and on the positions and number of pins.

It is clear that these topologies tradeoff wire length for delay. In real designs, there are two main strategies to control algorithms so that timing closure is achieved: slack satisfaction and critical path optimization. Although slack management is very precise, it is more complex, and many CAD tools rely on the optimization of critical paths, that are easily identified by incremental timing analysis. Optimization of critical paths surely reduce the worst logic delay and is an effective method of improving circuit speed. This way, we can relax the requirement of reducing the delay for all the sinks and concentrate on a few critical sinks that participate in critical paths. In this scenario, [8]–[10] proposed the *identified critical sink routing trees* in which a star tree topology is used for one identified critical sink while the rest of the sinks are connected by a minimum wire length Steiner tree (resulting in smaller Elmore delay to the critical sink). An example of such tree is given by Fig. 1(f).

B. Algorithms for Steiner Tree Construction

Performance-driven Steiner tree algorithms have been well studied and a variety of methods have been proposed. The first category of algorithms is based upon the minimum spanning tree and shortest path algorithms. Hou *et al.* [11] have shown techniques to find the shortest length Steiner tree under pin delay constraints. Alpert *et al.* [6], Boese *et al.* [10], and Cong *et al.* [5], have described several such algorithms. The spanning tree could be generated based upon different criteria. Boese *et al.* [10] proposed the use of an Elmore routing tree for the same purpose. The AHHK algorithm described by Alpert *et al.* [6], just like [5], builds a Steiner tree that trades off between shortest path and minimum spanning tree [see Fig. 1(d)]. These methods only see the graph containing net pins to build spanning trees, and usually employ a separate edge-overlapping procedure for conversion to rectilinear Steiner trees (RST). Lillis *et al.* [12] generate many topologies to minimize the Elmore delay for attending a required delay budget at each sink. The variety of topologies provides a wide range tradeoff between wire length and delay. Constructive algorithms have been proposed by Cong *et al.* [2], Hong. *et al.* [13], and Xu *et al.* [14]. The minimum Steiner arborescence [MSA—Fig. 1(c)] generated by Cong *et al.* [2] tends to have a high total wire length. The reviewed methods have limited control over the amount of sharing of the wires. In fact, the edge-overlaping procedure mentioned above tends to share most of the wires. For this reason, Boese *et al.* [10] proposed the SERT-C algorithm for individual critical sink routing, in which the wire to the critical sink is not shared and the rest of the tree is build as short as possible disregarding the delay of the sinks other than the critical one [see Fig. 1(f)].

In general, all the methods described so far do not account for blockages, congestion, etc. With the exception of [9] and [10], they also generally minimize the source to sink distance for all sinks in the net without discriminating between critical and noncritical sinks.

Algorithms based on path search on the other hand use intelligent methods that incorporate the desired properties into the search process to generate the tree. Commonly used path search algorithms include basic Dijkstra and the A* algorithm [18]. Dutt *et al.* [15] present an algorithm to perform incremental routing using Dijkstra algorithm to connect nodes to an existing tree in a restricted interval that satisfies the timing constraint. Hur *et al.* [16] present a method based on a multi-graph model for performance driven routing of two pin nets with wire sizing. Prasitjutrakal and Kubitz [17] have also proposed a basic timing-aware router. While this method uses Elmore delay to drive the A* search, their choice of the next target to be added to the tree is restrictive and can compromise the quality of the results in the presence of blockages. Furthermore, they do not differentiate (except for [12]) between the criticality of different sinks on the net.

III. Problem Definition

A net is a set V of points v located at positions (x_v, y_v) on a grid. These points need to be electrically connected in the circuit. Point s also called the driver, transmits a signal to all other sinks. All connections will be performed on the specified grid. Let subset K be the set of critical sinks in the net. For each critical sink $k \in K$ it is required that the delay of transmitting the signal from the driver s to k be minimized. Critical sinks can be identified by incremental timing analysis, that is available at the logic and placement levels of most CAD tools. For the rest of the sinks $i \in V - K$ delay is not considered and wire length should be minimized at most.

IV. Efficient Search Methods

A. A* Search

While implementing a simple path search router using Dijkstra or A* algorithms [18] is quite simple, based upon the use of subtle properties of these methods, it is possible to obtain a variety of results in terms of quality and CPU time depending on the problem formulation, search sequences, and other implementation options. With a good understanding of the A* algorithm and how it behaves in different situations, it is possible to get routing results whose quality is comparable to the best known algorithms for tree construction, while using little CPU time. In this section and in the next sections we will present some definitions and highlight some properties of the A* that support this flexibility.

A path search algorithm finds a path with minimum cost from a **source** node s to a **target** node t in a weighted graph. The A* algorithm uses a main data structure called **open list**. A node is said **open** when a reference to it is inserted into the open list. At all times, the open list must be kept sorted by the function f defined below. Initially the source node is open (i.e., inserted in the open list). The algorithm repeatedly selects the first node from the open list and **expands** it. The algorithm terminates when t is selected (or the open list is empty meaning that there is no path connecting s to t). When a node v is expanded, all of v's neighbors are then open. The node v is then **marked closed**. The open list is sorted in increasing values of $f(v) = g(v) + h(v)$, where $g(v)$ is the cost from the source to node v and $h(v) = k(v, t)$ an estimated cost of going from node v to t and is referred to as the heuristic estimator function.

If $h(v) = 0$ for every node then A* behaves like the Dijkstra algorithm, expanding nodes closest to the source first. If h is underestimated, e.g., $h(v)$ is always smaller than the actual cost from v to t, then $h(v)$ is called an admissible heuristic. The heuristic function helps to reduce the number of nodes expanded, during the search for the optimal path. If the heuristic function is admissible the A* algorithm guarantees that an optimal path will be found if such a path exists. In this case the algorithm is said to be admissible. Computing an underestimated h function is usually quite straightforward and fast. If h is also consistent, e.g., $k(v1, t) \leq k(v1, v2) + k(v2, t)$ for nodes $v1$ and $v2$, then the algorithm also knows the optimal path from s to v for each expanded node v, and never expands the same node twice [18]. The higher the heuristic estimator is (e.g., the closer the heuristic estimator is to the actual distance), the less nodes are expanded by A*. In routing applications, Manhattan distance (ManhD) is commonly used as heuristic estimator, being consistent and admissible (proof is omitted).

Let C^* be the optimum cost of reaching the target t from s. Algorithm A* expands each and every vertex v with $f(v) < C^*$ and no node with $f(v) > C^*$ [18]. We say that a **tie** happens whenever two nodes have the same value of $f(n)$ in the open list. A **critical tie** is a tie with the additional constraint that $f = C^*$. Simple ties are not a concern, given that all nodes with $f(v) < C^*$ must be expanded to ensure admissibility. Yet critical ties are very significant, specially for routing. The use of Manhattan distance as estimator and no additional costs for congestion or obstacles in initially empty areas make all estimates perfect, so all nodes inside the box bounded by the source and the target have $f(n) = C^*$. To get the most efficiency from A*, a mechanism is needed to avoid expanding all nodes with critical ties. In [18], Hart *et al.* point out that critical ties can be arbitrarily broken but always in favor of the target. Additionally, vertices that are closer to the target can be chosen to break intermediate critical ties for efficiency purposes. So, if two nodes have the same value of $f(n)$, theoretically the one with higher $g(n)$ should be expanded first. In routing this causes the effect of depth first searches (DFS) from s to t in empty areas. Now, in a regular and uniform grid, each expanded node that is not aligned to the target in x or y will open two neighbors with the same value of $f = C^*$ and the same value of g, what we will refer to as a **depth tie**. While depth priority can be used to get efficiency, depth ties represent a true degree of freedom for selecting between alternate paths every time this choice happens. A separate mechanism must be implemented to do that, and this will be addressed in Section V.

There is a final concern regarding ties. The routing grid cannot be assumed to be regular or to have the same step size in x and y. In a Hannan grid, two neighbors of a node may exhibit different values of g, and in this case we would lose the ability to recognize at this point that these are alternate paths that we must choose from. To cope with this, instead of using g (depth) as the critical tie breaking criterion, the number of expansion steps can be employed. For each vertex v a value $cs(v)$ is stored that indicates how many expansions were needed to reach v is from the source. We pick that vertex v with the highest $cs(v)$, and the definition of a depth tie is adapted accordingly.

In summary, a maze router using the A* algorithm can be made very efficient for routing. When running in empty and not congested areas whose costs are uniform and known, with the perfect Manhattan distance estimator and critical tie breaking based on $cs(v)$ the algorithm expands only the nodes that lie in the optimal path, most like in a DFS. A Hannan grid has fewer nodes and still preserves the degree of freedom regarding the choice of nodes that exhibit stepped depth ties. Additional speedup methods are addressed on Section VII. For global routing with congestion information, variable costs will slow down the search and change the occurrence of all types of ties. In extreme situations, bidirectional search and dynamic estimation methods such as LCS* [19] can be applied.

B. A* With Multiple Sources and Targets

Given a set of pins of the same net, a maze router will start from a particular pin and grow the tree inserting one pin at a time with the path search method. However, to properly generate Steiner trees, the classical approach for path search should be extended to multiple sources and multiple targets. All the intermediate points of the current tree are considered sources for the next search. The path-search algorithm itself selects the target pin among all marked targets. The details of the algorithm are given in algorithm 1.

Algorithm 1 A* Mult

Input: A graph $G = (V, E)$, a set S of sources and a set T of targets.
Output: A shortest-path with cost C^* from some $s \in S$ to some $t \in T$ such that ($\forall s_1 \in S \forall t_1 \in T$ the cost C of the shortest-path from s_1 to $t_1 \leq C^*$). New sources are created in S and t is removed from T.

1: Create an open list L and insert a reference $R(s)$ into it.
2: **for** every $s \in S$
3: $g(s) = 0$ ($s \in S$)
4: $ct(s) = ct$ such that $\forall t \in T(\mathrm{ManhD}(s, ct) \leq \mathrm{ManhD}(s, t))$
5: $h(s) = \mathrm{ManhD}(s, ct(s))$
6: Mark the predecessor pointer $p(s)$ as invalid
7: Create a reference $R(s)$ with $g(s)$, $h(s)$, $ct(s)$, $p(s)$
8: Insert reference $R(S)$ into L
9: **end for**
10: **while** (L is not empty)
11: Remove the first reference r from L
12: Get the vertex v from r
13: If v is closed than **continue** to the next iteration.
14: Set $G.\mathrm{pred}(v) = r.p(v)$
15: Mark v as closed
16: **if** ($v \in T$) **then**
17: Set $t = v$ as the reached target
18: **break**
19: **end if**
20: **for** each vertex u that is adjacent to v in G and is not closed
21: $g(u) = g(v) + c(v, u)$
22: Set $ct(u)$ as in step 4
23: $h(u) = \mathrm{ManhD}(u, ct(u))$
24: $p(u) = v$
25: Create $R(u)$ with $g(u)$, $h(u)$, $ct(u)$, $p(u)$
26: Insert $R(u)$ into L
27: **end for**
28: **end while**
29: **if** a target is found **then**
30: Retrace back the path from the reached target t until some $s \in S$ is reached. Use $G.\mathrm{pred}(i)$ (i are intermediate nodes in the path); Insert all i in S. Move t from T to S.
31: **else**
32: Report that no $t \in T$ is reachable from S
33: **end if**

Adding the capability of dealing with multiple sources and targets to the classical A* is straightforward. The algorithm 1 presents the steps of the algorithm in detail. All the sources are initially open (see steps 2–8) and the priority-queue of the open nodes will automatically select the most promising node to start with. Multiple targets can be handled simply by stopping the algorithm whenever any target is reached (see step 18). The heuristic function $h(v)$ will point to the target t that is the closest to v. The concept of chosen target $ct(v)$ is introduced and stored in the references r that are stored on the open list. Every open node has an associated chosen target that is the closest target to it measured by Manhattan distance.

An important property of A* is that f monotonically increases during the search (monotonicity property). This property was already demonstrated [18] if h consistency holds.

Theorem 1 demonstrates the consistency of the multiple target heuristic estimator.

Theorem 1: For two nodes n_1, n_2, each with a different assigned target $ct(n_1)$ and $ct(n_2)$, respectively, $k(n_1, ct(n_1)) \leq k(n_1, n_2) + k(n_2, ct(n_2))$.

Proof: At node n_1, we know that the closest target is $ct(n_1)$, so $k(n_1, ct(n_1)) \leq k(n_1, ct(n_2))$. By the consistency property demonstrated in [18] for single target searches, $k(n_1, ct(n_2)) \leq k(n_1, n_2) + k(n_2, ct(n_2))$. Joining the equations, we conclude the proof. □

In the presence of delay critical sinks, we first route critical sinks in order of criticality and than we route the noncritical ones. When routing a critical node, regular nodes are not considered targets, but they are used for biasing calculation (see Section V). The algorithm for generating a Steiner tree with priority to the critical nodes is given in Algorithm 2.

Algorithm 2 A* Steiner with critical nodes

Input: Graph G, the net driver s and a set T of sinks with an associated criticality.
Output: A Steiner tree connecting s to all targets in T.

1: $\mathrm{S} = \mathrm{s}$
2: **while** (T is not empty)
3: Call A* (S, T_c) such that T_c contains only the nodes from T with highest criticality
4: $T = T - t$ where t is the A* chosen target
5: $S = S \bigcup V_{st}$ where V_{st} is the set of all vertices in the path P returned by A*, including t
6: **return** S
7: **end while**

V. Wire Length Optimization

As already stated, often there are situations when two or more vertices (for instance v and n) have the same value of f as well as cs ($f(v) = f(n)$ and $cs(v) = cs(n)$). Uniformity of the grid will lead to such situations. The choice of the appropriate v will determine whether the upper-left L shaped wire or the lower-right L shaped wire will be selected, as illustrated by Fig. 2. These cases are degrees of freedom provided by the A* search. Either choice is valid and will lead to an admissible (shortest)

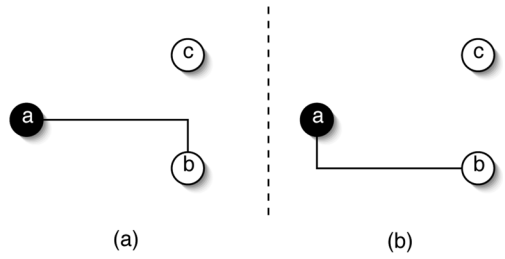


Fig. 2. (a) Illustration of a routing situation favorable to wire length minimization and (b) a favorable situation for the isolation of the paths.

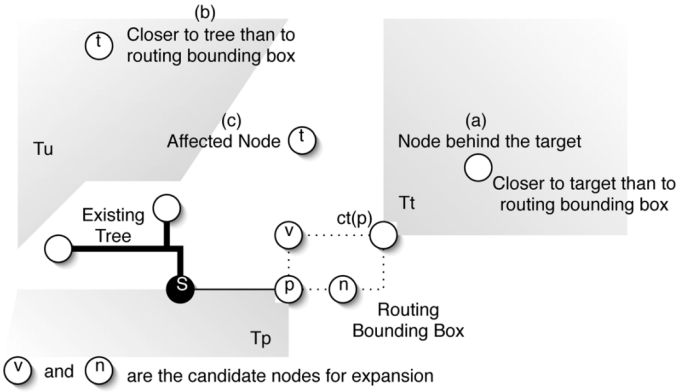


Fig. 3. Illustration of the biasing technique and the affected target. (a) Shows one target that is excluded because it is behind the target. (b) Shows one target that is closer to the existing tree than to the routing bounding box. (c) Example of a target that will affect the biasing point. In this situation, the path will go into the direction of node v.

path. Instead of letting the coding style implicitly decide on the choice we introduce a **biasing** technique to direct the search according to the needs of future connections to unrouted sinks. Clearly, the choice (a) in Fig. 2 is best for wire length minimization. The biasing technique will attempt to select the wire accordingly.

Biasing uses a reference point called **biasing point** that is used to choose the vertex for expansion. The biasing point is calculated by first determining a set of affected targets. Let T_p be the set of vertices in the quadrant with origin p (p is the predecessor of the candidate node v) that is diagonally opposite to t and T_t be the set of vertices in the quadrant with origin t that is diagonally opposite to p. The biasing point will not be affected by vertices in $T_p \cup T_t$, since p or t will be closer to these vertices than any other vertex in the routing box bounded by p and t, as shown in Fig. 3, situation (a).

Likewise, any vertex in the set of vertices (denoted by T_u) that are closer to the tree than the routing box bounded by v and t will not affect the biasing point either, as shown in Fig. 3 situation (b). Situation (c) represents one node that will affect the biasing calculation since it is neither behind the source/target nor closer to the tree than the routing box.

Therefore, $T_{\text{affected}} = T - (T_p \cup T_t \cup T_u)$. If (x_t, y_t) is the coordinate of a target $t \in T_{\text{affected}}$ whose distance from the vertex v is d_t the bias point is computed as a weighted centroid of the affected nodes according to (1)

$$(x_c, y_c) = \left(\frac{\sum_{t \epsilon T_{\text{affected}}} \frac{x_t}{d_t}}{\sum_{t \epsilon T_{\text{affected}}} \frac{1}{d_t}}, \frac{\sum_{t \epsilon T_{\text{affected}}} \frac{y_t}{d_t}}{\sum_{t \epsilon T_{\text{affected}}} \frac{1}{d_t}} \right). \quad (1)$$

The algorithm that calculates the biasing value *b(v)* is shown bellow (Algorithm 3).

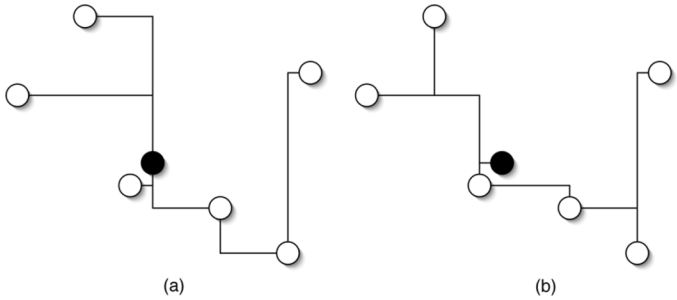


Fig. 4. Visualization of trees (a) without biasing and (b) with biasing.

Algorithm 3 Biasing Value

Input: The node to be expanded v, the parent n, the closest target t, the complete set of targets T, graph G.
Output: The value $b(v)$.

1: Compute the unaffected region R_p that is the opposite quadrant of p from t.
2: Compute the unaffected region R_p that is the opposite quadrant of t from n.
3: Let T_p and T_t be the set of targets in R_p and R_t.
4: Determine the set of vertices T_u that are close to the tree than to the rectangle defined by s and t.
5: Compute the centroid (X_c, Y_c) as explained before.
6: **return** $b(v) = |x_p x_c| + |y_p y_c|$

The bias value $b(v)$ is calculated by the distance from v to the biasing point. Vertices with the same $f(v)$ and $cs(v)$ in the open list are sorted by increasing values of $b(v)$ (the node with smaller $b(v)$ is selected). Fig. 4 illustrates the effect of the biasing technique on a 7-pins net. In this case, the pin placement favored the sharing of some wires and the biasing technique provided a 15% improvement in wire length.

Though biasing helps to reduce the total wire length, it could potentially result in sharing a path P from source to critical target with paths to other targets thereby increasing the capacitive load on P, which, combined with a significant value of wire resistance, can slow it down. To isolate critical paths and make them less likely to be shared we suggest the use of repulsive biasing for critical targets. Repulsive biasing sorts the open list in decreasing order of $b(v)$ and tends to route wires that connect the source to critical targets in such a way that ample space is available for routing noncritical wires.

We observe that the biasing technique is able to improve the wire length by breaking ties in the open list. The number of ties on a search is highly dependent on the modeling of the routing space. Cost functions that model congestion, for example, will

TABLE I
IMPACT OF THE BIASING TECHNIQUE IN AVERAGE FOR WL (MEASURE IN μM AND RUN TIME (S). IMP ROWS REPRESENT THE IMPROVEMENT ACHIEVED BY USING THE BIASING TECHNIQUE

	Random				Placed Circuit	
	3 pins		15 pins			
	wl	t (s)	wl	t (s)	wl	t (s)
	Considering the costs of Vias					
Off	314	1.38	915	7.24	3937	45.6
On	304	1.38	915	7.24	3937	45.6
Imp	3.0%	0%	0.7%	-2.8%	0.9%	0.3%
	Ignoring the vias					
Off	311	1.34	905	6.38	3899	42.0
On	301	1.23	881	6.58	3827	43.3
Imp	3.3%	-9.8%	2.6%	-3.1%	1.8%	-3.0%

reduce the amount of ties occurred in a search. Also, modeling of the vias will minimize the number of bends but also reduce the degrees of freedom for the biasing technique, since Z-shaped connections cost more than L shaped ones.

In order to evaluate the impact of the biasing technique in wire length and run time we performed experiments in two sets of benchmarks: random and placed circuit. For the random set we generated one thousand trees varying the number of pins from 3 to 15 in a space of 300 μm $\times$ 300 μm. The placed circuit set was extracted from the ibm02 circuit from ISPD 2004 placement benchmarks suite after full placement. This circuit has 19584 nets; 54% are 2-pin nets (those are being ignored) followed by 9%, 9%, 9%, 2%, 1.5%, 1.5%, 2%, 2%, 2.5% for 3, 4, 5, 6, 7, 8, 9, 10, and 11 pins; 7.5% of the nets are between 12 and 134 pins (an average of 4.1 pins per tree). In these experiment, we obtained an average of 1.1 depth-tie situations per tree, which means that biasing was applied approximately once per tree in average. The average results are presented in Table I. For both options, we study the impact of modeling the vias or not. Analyzing the table, we report average gains in the order of 1%–3% considering the vias and 2.5%–3% ignoring the vias, reinforcing the conclusion that the modeling of the vias will reduce the degree of freedom for the biasing technique and consequently increase wire length.

We observed that the average numbers are very small considering the visual impact (exemplified by Fig. 4) of the biasing technique. We then plotted a histogram of the biasing improvement for the placed circuit nets with 3 or more pins. The histogram, on Fig. 5, shows that in the big majority of cases the impact is 0%. Analyzing each case, we could observe that in fact the case where wire sharing does not help is the most common. However, for the ones that are affected by wire sharing, biasing can improve the wire length of a net by up to 20%. We also observed in the histogram the same reduction on the improvement of the biasing technique with the modeling of the vias.

The computation of whether a target is behind the source or the target (belongs to sets T_p or T_t) can clearly be performed in constant time, which makes it $O(t)$ for t targets. Determining whether the targets are closer to the existing tree can also be made in $O(t)$ if the distance of each target to the tree is precomputed in step 3 of Algorithm 1.

VI. DELAY OPTIMIZATION TECHNIQUES

This section presents an analysis of Steiner tree delays using the Elmore delay model [20]. While the model lacks accuracy

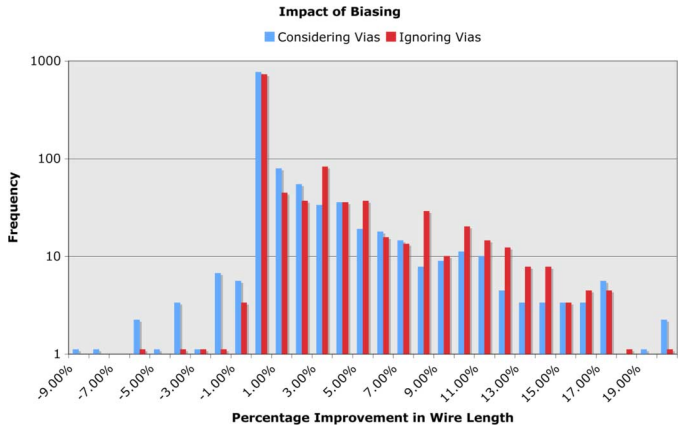


Fig. 5. Histogram showing the impact of biasing on nets taken from a placed circuit. The big majority of nets is not affected by biasing, but almost none is affected negatively while improvements can be in the order of 20%.

it has fidelity [10], which means that ranking trees by Elmore delay is reasonably accurate.

We observe the following three features of a particular wire are related to its Elmore delay.

- *Total Wire Length:* The driver output resistance is multiplied by the capacitance of the entire tree. The higher is the driver resistance the greater is the importance of reducing wire length of the net. Generally, critical nets consist of properly sized gates and are likely to have a small driver resistance.
- *Path Length Along the Tree to the Critical Sinks:* The wire resistance to the sink is proportional to the path length.
- *Amount of Wire Sharing:* Wire delay is proportional to the amount of capacitance it is driving. A wire that is shared by many paths has resistance elements that drive a greater downstream capacitance.

Section VI-A provides a theoretical background for the wire sharing impact on Elmore delay. We start with the simple case with one critical node k and one noncritical node t and then show how to add new sinks optimally to an existing Steiner tree. Section VI-B presents our algorithm for controlling the wire sharing on our Maze Router. Section VI-C presents an algorithm for controlling the path length to the critical sinks.

A. Theoretical Analysis of the Impact of Sharing

1) Three-Terminal Analysis With One Critical Sink: Fig. 6(a) shows a three-terminal net with source s, critical sink k, and a noncritical sink t. The source s is connected to the sinks k and t by the shortest paths p_k and p_t, of lengths d_{sk} and d_{st}, respectively. The length of wire shared by p_k and p_t is defined as d_{sp}. Elmore delay D_{sk} to the critical sink k is computed as follows. If D_R is the delay due to the source resistance R, and D_{sp} and D_{pk} are the delays due to the shared portion (p_{sp}) and the non-shared portion (p_{pk}), respectively, of the path p_{sk}, then D_{sk} is given by $D_{sk} = D_R + D_{sp} + D_{pk}$. The delay $D_R = R \times (d_{sk} + d_{st} - d_{sp}) \times c + R \times (c_t + c_k)$. We sum the product of the resistance and downstream capacitance along the path p_{sp} by integrating along p_{sp} to obtain the value of D_{sp} as

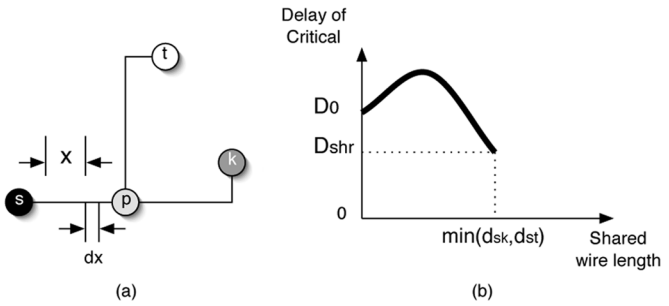


Fig. 6. Simple three-terminal example: (a) delay versus (b) shared wire length.

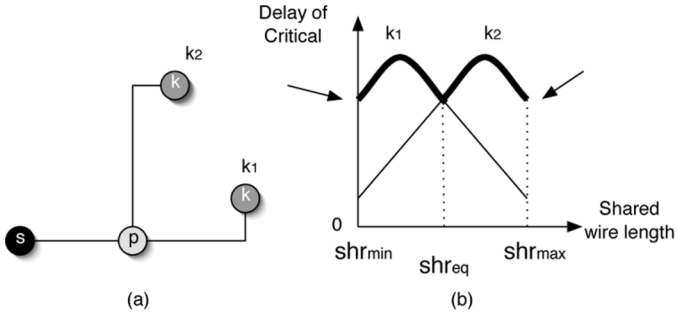


Fig. 7. Net with two critical sinks: (a) delay versus (b) shared wire length.

shown in (2) (where r stands for resistance per wire unit and c stands for capacitance per wire unit)

$$D_{sp} = \int_0^{d_{sp}} (d_{st} + d_{sk} - d_{sp} - x) rcdx + \int_0^{d_{sp}} (c_t + c_k) rdx = r[c(d_{st} + d_{sk}) + c_k + c_t] d_{sp} - \frac{3rcd_{sp}^2}{2}. \quad (2)$$

In a similar way, we compute the delay D_{pk} as $D_{pk} = rc(d_{sk} - d_{sp})^2/2 + rc_k(d_{sk}d_{sp})$. Since $D_{sk} = D_R + D_{sp} + D_{pk}$, the delay is of the form of (3), where the coefficients $A_1 = R(cd_{sk} + cd_{st} + c_k + c_t)$, $A_2 = rcd_{st} + rc_tRc$, and $A_3 = rc$. In the absence of congestion and blockages the values of d_{sk} and d_{st} are Manhattan distances for a given location of the source and sinks.

The plot between delay and the length of wire shared is characterized by (3) and shown in Fig. 6(b). This is an inverted bell shaped curve and is referred to as the delay curve. The values D_0 and D_{shr} are delays when there is no sharing and at maximum sharing, respectively. Clearly the minimum delay occurs only when the length of wire shared between paths from the source to the critical sink and the noncritical sink, is either 0 or the maximum possible. In other words the delay is dominated by either that due to *source resistance* or due to *wiring delay*

$$A_1 + A_2 d_{sp} - A_3 d_{sp}^2. \quad (3)$$

2) Three-Terminal Analysis With Two Critical Sinks: Fig. 7(a) shows a net with two critical sinks k_1 and k_2. Here, the maximum of the delays to both critical sinks must be minimized. Fig. 7(b) shows one possible set of delay curves for sinks of the net shown in Fig. 7(a). The best delay is one that minimizes the maximum of the delays to sinks k_1 and k_2 and is shown using a bold line in Fig. 7(b). In addition to computing delays with no sharing and maximum sharing, we may also have to compute delay when sharing is such that both delays are equal. Let D_{k1} and D_{k2} be the delays of the critical sinks k_1 and k_2, respectively. From (3), we see that $D_{k1} = A_1 + A_2 d_{shr} A_3 d_{shr}^2$ and $D_{k2} = B_1 + B_2 d_{shr} B_3 d_{shr}^2$.

The best delay for the design will occur when there is minimum/maximum sharing ($shr_{\min}/shr_{\max}$), or as in this case when both the delays are equal (shr_eq). Since A_1, A_2, A_3, B_1, B_2, and B_3 are known we locate the length of the path to be shared by solving for d_{shr} in $A_1 - B_1 + (A_2 - B_2)d_{shr} - (A_3 - B_3)d_{shr}^2$.

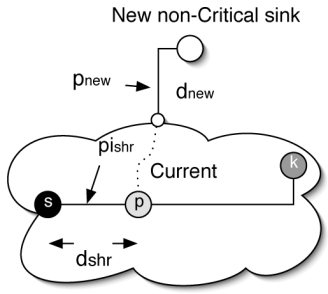


Fig. 8. Inserting a noncritical sink.

3) Incremental Tree Extension: We now consider incrementally extending a given Steiner tree with an additional sink. Such an extension can form the basis of a heuristic algorithm to perform complete wiring for a net by adding sinks to a tree that is initialized to be just the source vertex.

a) Adding a Noncritical Sink: We consider the problem of adding a noncritical sink to net that has an existing Steiner route and has exactly one critical sink. This algorithm is exact and has applications in heuristics for generating delay optimal Steiner trees. For the net in Fig. 8, let p_{st} be a shortest path from s to the new sink t. Let p_{new} be the portion of p_{st} added to the existing tree, and p_{shr} be the portion of p_{st} shared with p_{sk}. The Steiner tree to which the sink is being added is shown as the cloud. In the Elmore delay model the increase in delay due to the addition of sink t can be computed by summing up the delay increase D_R due to increased capacitance seen by the source resistance R and the increase in delay D_{shr} due to the additional wire capacitance seen by path p_{shr}. Clearly, $D_R = R(cd_{\text{new}} + c_t)$ and $d_{shr} = rcd_{shr}d_{\text{new}} + rd_{shr}c_t$.

In order to minimize $D_R + D_{shr}$, the delay d_{new} must be minimized in conjunction with the product $d_{shr} \times d_{\text{new}}$. Consider the addition of a sink t using a new wire starting at some point located between two branching points, v_1 and v_2, on the path p_{sk} of the existing Steiner tree. Assuming v_1 to be a virtual source and v_2 to be a virtual critical sink, we note that the curve between the delay at v_2 and the shared wire length is an inverted bell as shown in Fig. 6(b). The best delay at v_2 can only be obtained when the new wire starts either at v_1 or at v_2. We therefore observe that in order to obtain the best delay, sink t can only be connected to a sub tree that is rooted at a branching point (including source s and sink k) that is on the path p_{sk} of

the existing tree. This observation simplifies our search for the starting point of the new wire considerably.

Let $x_i, 0 < i < m$, be the set of branching points on the path p_{sk}, where x_0 and x_m correspond to the source s and to the sink t. Let p^i_{new} be the required addition to the sub tree rooted at the branching point x_i and p^i_{shr} the shared path. Let d^i_{new} and d^i_{shr} be the lengths of the new addition and the lengths of the shared portion of the path, respectively. If a chosen branching point minimizes d^i_{new} then the amount of shared resistance that must drive the extra capacitance could potentially become very high. On the other hand, choosing a branching point that minimizes d^i_{shr} could potentially make d^i_{new} very large. We find the desired branching point by locating that point x_i corresponding to the minimum increase in delay $D_i = R(cd^i_{\text{new}}+c_t)+rcd^i_{\text{new}}d^i_{shr}+rc_td^i_{shr}$. The optimum value x_{opt} is computed by evaluating delays obtained by connections using the shortest wire from the new sink to sub trees rooted at each possible value of x_i and choosing the one with the lowest delay. The distance from the new sink to all trees rooted at branching points can be computed in $O(g^2)$ time, where g is the number of grid points. In practice, it is fast since the number of branching points on path p_{sk} is usually small. If the generated tree is used for estimation purposes only then using a coarse grid could speed up algorithm.

The tree could contain several critical as well as noncritical sinks. We use the method outlined in the previous Section. The method is similar except that branching points along paths from the source to each critical sinks in the net must be analyzed. Since this method does not provide any new insight we omit the details.

b) Adding a Critical Sink: This method is similar to that described in Section VI-A3 with the difference that delays must be evaluated not only to the critical sinks in the tree but also to the new sink being added. Furthermore, the equal delay points must be analyzed as shown in Section VI-A3 for the new sink and each critical sink already in the tree. Again, for the sake of conciseness, we omit the details.

c) Incremental Steiner Tree Construction: Starting with an initially empty Steiner tree we add one sink at a time in our heuristic incremental construction of the Steiner tree. We first add all the critical sinks one by one and then follow by adding noncritical sinks. We use the methods described in Section VI-A3b to add critical sinks and that in Section VI-A3a to add noncritical sinks.

While this algorithm works in principle, we have found that similar wire sharing effects can be obtained by incorporating a sharing factor in the maze routing algorithm for Steiner tree construction. This modification to our heuristic Steiner tree construction simplifies the implementation considerably and also has desirable wire sharing properties. The sharing factor concept is described in Section VI-B.

B. Sharing Factor on Maze Search

From the theory established in the earlier sections it is clear that delay performance can be effectively managed by controlling the amount of sharing. On the other hand, we observe that optimizing the delay of trees with multiple sinks is complex, requiring a heuristic solution instead. We introduce the capability of wire length to delay trade-off using a parameter called

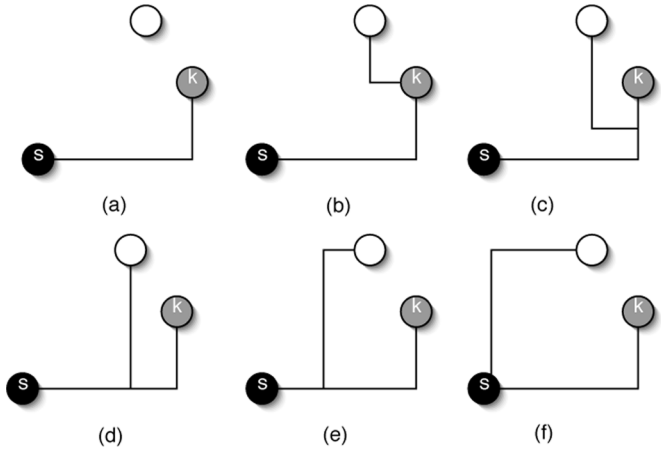


Fig. 9. Effect of sf on maze search.

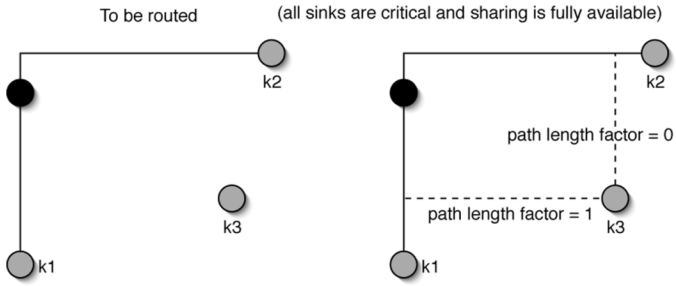


Fig. 10. Effect of path length factor on maze search.

the *sharing factor* (sf). The sharing factor has a value between 0 and 1 and is used to designate some parts of the tree as prohibited for connection, avoiding sharing of these wires. Critical wires connect the driver to critical sinks. From the formulation presented in the previous section, sharing a small amount in the beginning of this wire may be less harmful than sharing longer portions of a critical wire.

In step 1 of A* Mult algorithm we ascertain, for every node s_k on the tree whether or not it is part of a critical path. For every node n that is part of the tree, we store the distance d_n of the node from the source along the path of the tree. This distance can be obtained by the g from previous searches. If k is a critical sink, then all wires that are closer to s_0 than $\text{sf}*d_k$ are available for sharing. Clearly, if $\text{sf} = 1$ then all wires in the path from s_0 to s_k are available for sharing and if $\text{sf} = 0$ then no wire in the path is available for sharing. Fig. 9 shows the effect of different values of sf on the types of routes generated.

C. Path Length Factor on Maze Search

Path length is the term used for the distance of a critical sink k to the net driver s along the tree. If the path length is optimal for k then optimal wire resistance (disregarding possible wire sizing) is obtained for it. For example, consider Fig. 10, where k_1 and k_2 are connected to the driver with optimal path length. Observe that node k_3 can be connected into the path to k_1 with optimal path length as well, but connected to k_2 it would have a small detour.

The previous sections assumed that minimum path length to the critical sinks could be achieved easily by performing their

connections first. It is always true for a single critical sink but may not be for multiple critical sinks if sharing is available. In such cases, we propose the use of a path length factor (plf) in order to guarantee that the minimum path length for the critical sinks is achieved. The idea was based on [21] and extended to attend critical sinks only. For a node $n \in S$, the path length factor uses d_n (distance from the source along the tree). At each shared node, a number between 0 and 1 is multiplied by d_n. The product of this multiplication is set as initial g of the node (affecting step 3 of algorithm 1). The effect of initializing g with a constant value like 0 is that the obtained routes will not correspond to shortest wires from the source; the proposed algorithm will optimize overall path length instead. A tradeoff of the two goals can be obtained with the plf. If the path length factor is set to 0, then a MST-like tree will be generated. All intermediate values tradeoff wire length for path length.

VII. Run Time Improvement Techniques

A. Application Specific Grid

In general, the use of a coarser grid improves the speed of the algorithm at the expense of quality. During early estimation, speed is of essence. In this case, we use a Hanan grid [22] to model the space thereby reducing the grid to the smallest size that accommodates all the sinks. This leads to better memory usage as well as shorter runtime. During actual routing we use a finer grid thereby increasing the routability and improving wire length at the expense of CPU time.

Costs are also supported by a maze router and can be used at the expense of run time as well. Congestion cost can be modeled by a cost function. It is important to notice that those costs must remain static during the A* expansion in order to guarantee admissibility. Applications such detailed routing could take benefit of increasing costs dynamically during the expansion to cope with design rules.

Another feature of the maze routers is the ease to model multilayer routing. This is merely an extension of the maze router to a three dimensional scenario with via costs assigned appropriately. As already mentioned, the situation where f is unchanged during the whole search is the best for speeding up the algorithm. To achieve that, in the case where the space is modeled in 3-D, via costs must be included in the h function. If the Z coordinate of the node n being opened is the same as its target and in any of the situations described as follows, the value of twice the via cost can be included in the h function without harming the algorithm's admissibility:

- $n \cdot X \neq ct \cdot X$ and $n \cdot Y \neq n \cdot Y \neq ct \cdot Y$;
- $n \cdot X = ct \cdot X$ and $n \cdot Z$ is horizontal layer;
- $n \cdot Y = ct \cdot Y$ and $n \cdot Z$ is vertical layer.

This improvement was tested on Steiner trees extracted from ibm02 placement benchmark and resulted in 17% reduction of expanded nodes.

B. Simplified Heuristic Function (h) Calculation

Every time a node is open (steps 21–26), it has an associated *closest target* (ct) that is used to calculate its h value (as demonstrated in step 23). Consider the scenario where a node p, with associated closest target ct, is being expanded and is opening its neighbors represented by node n. We propose that, instead of looking at all of the targets, only ct be checked. The value f'_n denotes the f function of node n forcing it to point at ct without measuring if ct is actually the closest target from n. Consider two possibilities: $f'_n = f_p$ and $f'_n \neq f_p$. Theorem 2 explores the first possibility.

Theorem 2: If $f'_n = f_p$, then $f_n = f_p$ with the same ct.

Proof:

1) $f'_n = f_p$.
2) $f'_p \leq f_p$ (monotone property).
3) $f'_n \leq f_n$ (combination of 1 and 2).
4) $f'_n \geq f_n$ (this step is a requirement of A*—it must choose the target that leads to smaller f).
5) $f_n = f_p$ (combination of 3 and 4).
6) The target for f'_n can be used.

□

Considering that $f'_n \neq f_p$, the associated target ct for n_i must be recalculated because it might have changed. Since f_n will never be smaller than f_p this calculation could be postponed until the situation in which there are no more nodes in the open list with the current value of f. This mechanism is implemented as follows. An auxiliary list of open nodes is created, where no f value is computed. The A* search does not look at this structure unless the current f value changes upward. In this case, the auxiliary open list is flushed and an open list is built afresh.

C. Specialized Open List

The open list data structure must be a priority queue sorted by $f(v), cs(v)$ and $b(v)$ respectively. The insertion and removal operations in a standard STL priority queue implemented as a binary heap take $O(\log n)$ time. While this is not a strong performance limitation by itself, it demands any open node v to have calculated $f(v), cs(v)$, and $b(v)$ values. In our algorithm, we want to postpone the calculation of $b(v)$ to expansion time (see Section VII-D), for it is more expensive. Therefore, the binary heap turns out not to be the best data structure. We propose a specialized open list supported by theorems 3 and 4, which can be classified as a *bounded height priority queue*, a structure similar to what is used in bucket sorting.

Theorem 3: During an A* search (with a single source node), the difference of the smallest f to the largest f in the open list at all times is restricted to twice the maximum cost in the graph.

Proof: Consider the initial situation where the only open node is the source node s with f_s. After it is expanded, neighbors n are open and inserted in the open list. The one with largest f will cost $f_s + \Delta g + \Delta h$ where Δg is the highest cost edge connected to s that points in the opposite direction of the target ct. In the worst case, $\Delta h = \Delta g$ by the admissibility property and Δg is the highest cost in the graph (*HighestCost)*. In this case, the f range in the open list is given by $f_s + 2 \times \mathrm{HighestCost} - f_s = 2 \times \mathrm{HighestCost}$. As the search advances, the situation will be repeated for the expanded nodes p that follow. While p has $f_p = f_s$ the worst case f will be given by the same equation $f_p + 2 \times \mathrm{HighestCost}$. When a node k with a higher value for f_k is expanded, the largest f will be given by $f_k + 2 \times \mathrm{HighestCost}$, but the range is still given by $2 \times \mathrm{HighestCost}$ since the smallest f in the open list is now f_k.

TABLE II
WIRE LENGTH COMPARISON OF AMAZE TO OPTIMAL TREES (GEOSTEINERS) AND OTHER HEURISTICS

Alg.	#nodes:	8	14	20	26	49	100
GeoSteiners	Wire length	81.68	197.61	337.13	496.52	1289.23	3751.65
	CPU (s)	0.01	0.02	0.05	0.29	0.16	0.92
AMAZE WL	Wire length	101.60%	101.80%	102.10%	102.20%	102.70%	102.40%
	CPU (s)	0	0	0.02	0.01	0.06	0.4
Labyrinth	Wire length	111.10%	111.10%	111.70%	112.40%	112.60%	——
	CPU (s)	0.01	0.04	0.08	0.16	0.55	——
AHHK (c=0)	Wire length	101.30%	102.60%	102.40%	102.20%	102.60%	102.60%
	CPU (s)	0.01	0.01	0.01	0.02	0.07	0.53

Theorem 4: Given multiple sources, the f range is given by the maximum between the f range of the initially open nodes and $2 \times \mathrm{HighestCost}$.

According to the monotone property, the minimum f value $f_{\min}$ is given by the first node expanded in the search. Based on the maximum range for the f function (fr) from Theorems 3 and 4, we implement a circular array of fr positions, indexed by $(f - f_{\min}) \bmod (fr+1)$. At each node, there is another array indexed by cs and finally, at each node of this two dimensional array there is a linked list of references to nodes. This infrastructure provides constant time insertion. On access or removal the array must be traversed until a non-empty list is found. The time complexity of such operations will be $O(fr \times csr)$ in the worst case and $O(1)$ on average.

At the *expand* procedure, a linked list is located for the current value of f and cs. If this list contains only one node it is returned to A*Mult. When the list contains two or mode nodes, the biasing technique is applied to decide which should be expanded first. The advantages of such a structure is that the biasing point and $b(v)$ values are computed only if necessary, e.g., if A* reached both nodes and must decide which one to expand.

VIII. EXPERIMENTAL RESULTS

The algorithm developed here with all the previously described techniques will be called AMAZE from now on. The experimental results are divided into the following sections.

- *Wire Length Experiments:* to compare trees that AMAZE generates when optimizing WL alone to the optimal topologies, to heuristics for Steiner trees construction and to maze routers.
- *Delay Experiments:* to compare the trees generated by the AMAZE critical sink approach and AMAZE critical arborescence approach (path length factor is set to 1) to heuristics like AHHK [6] and P-Trees [12].
- *Wire Length Versus Delay Tradeoff Analysis.*
- *Analysis of the Impact of Blockages on the Generated Topologies.*

In all subsections, we applied randomly generated trees since we understand that, statistically, they cover all possible patterns. Placed circuits may exhibit an uneven distribution of the patterns that is sometimes consequence of that particular circuit or placement algorithm.

A. Wire Length Experiments

Initially we verified the wire length of our trees disregarding delay optimization. We compared AMAZE to three other algorithms: 1) the GeoSteiner software [23] that finds the optimal Steiner tree in an acceptable time; 2) the Labyrinth maze router [24]; and 3) the AHHK algorithm [6] configured for best wire length.

Table II presents the average results for 30 randomly generated trees ranging from 3 to 100 nodes. All values are normalized to the GeoSteiner solutions (optimal trees). In summary, we can observe that AMAZE generates near optimal trees (within 2% in average) while run time is better than even the AHHK heuristic. We can also observe that Labyrinth could not find good trees and requires significant more run time. The case with 100 nodes in a tree could not finish because of memory requirements. Such a result enforces the importance of the techniques described in this paper to the routing community.

B. Delay Experiments

We have conducted experiments with randomly generated nets, varying the numbers of total sinks (n_t) and critical sinks (n_k). For each (n_t, n_k) pair we averaged the results of 100 randomly generated nets. We compared three configurations of our algorithm to the P-Trees [12] and to a special version of AHHK [25]. The first configuration (AMAZE WL) is set to minimize WL only. As demonstrated in the previous section, our trees are close to optimal in terms of WL. The second and the third configuration (AMAZE) are set to demonstrate the effectiveness of the sharing factor while fixing the path length factor to 0 and 1, respectively.

In these tests we used our own implementation of the AHHK algorithm, configured to use 0, 0.25, 0.5, 0.75, and 1 as the control factor, as suggested in [25]. When implementing the edge overlapping procedure, however, we selected a method that results in less shared wires, as it produced tress with smaller delays. Therefore, this implementation, called DAHHK from now on, has better delay results and yields to higher wire lengths when compared to the results obtained from [26]. The same observation is valid for our previous publication [3] as well.

The P-Trees are configured to have required arrival times of 0 ps in the critical sinks, while the remaining sinks have no timing requirements, and are generated from the executable provided by the authors.

All these algorithms enable the user to choose the best tree considering the wire length and delay tradeoff. In all cases, we pick the best delay configuration, disregarding wire length. We claim that the most critical nets of the circuit must have minimum possible delay, and congestion will not be strongly affected since these nets represent a small portion of the wires.

In our experiments, we used wire resistance and capacitance values from an actual fabrication line and source resistance and

TABLE III
DELAY COMPARISON OF AMAZE TO DAHHK AND P-TREES IN A (300 μM $\times$ 300 μM) AREA

Alg	#nodes	3		5		7			9			11		
	#critical	1	all	1	all	1	3	5	1	3	5	1	3	5
AMAZE	WL (mm)	98	98	447	447	565	565	565	666	666	666	741	741	741
WL	Delay (ps)	45	54	85	109	120	161	173	165	222	235	194	260	274
AMAZE	WL (mm)	391	392	589	788	735	1078	1218	841	1280	1407	919	1320	1465
$plf = 0$	Delay (ps)	30	40	32	49	34	50	59	33	52	66	32	51	69
AMAZE	WL (mm)	391	391	589	786	735	1059	1200	841	1287	1400	919	1305	1454
$plf = 1$	Delay (ps)	30	40	32	48	34	49	57	33	51	64	32	51	67
DAHHK	WL (mm)	297	297	538	540	784	801	802	971	1019	1025	1202	1232	1234
	Delay (ps)	45	53	53	76	56	78	87	55	81	89	55	76	87
P-Trees	WL (mm)	391	391	642	730	863	1011	1051	1104	1262	1281	1314	1459	1504
	Delay (ps)	30	40	37	48	41	53	56	45	57	60	46	57	60
AMAZE	WL	-32%	-32%	-9%	-49%	6%	-32%	-50%	13%	-26%	-37%	24%	-6%	-18%
vs DAHHK	Delay	32%	26%	38%	39%	40%	37%	34%	40%	36%	28%	42%	32%	22%
AMAZE	WL	0%	0%	8%	-8%	15%	-5%	-14%	24%	-2%	-9%	30%	11%	3%
vs P-Trees	Delay	0%	0%	14%	1%	19%	8%	-2%	27%	11%	-7%	30%	11%	-12%

TABLE IV
DELAY COMPARISON OF AMAZE TO DAHHK AND P-TREES IN A (100 μM $\times$ 100 μM) AREA

Alg	# nodes	3		5		7		9		11	
	# critical	1	3	1	3	1	3	1	3	1	3
AMAZE	WL (mm)	98	98	152	152	188	188	221	222	244	244
WL	Delay (ps)	14	16	26	34	39	50	48	68	67	91
AMAZE	WL (mm)	126	125	200	275	243	344	276	395	313	441
plf = 1	Delay (ps)	10	12	10	15	11	15	10	16	11	18
DAHHK	WL (mm)	98	98	184	186	255	260	330	345	414	432
	Delay (ps)	14	16	17	24	18	25	16	25	19	27
P-Trees	WL (mm)	125	125	223	247	311	344	382	435	490	540
	Delay (ps)	10	12	12	15	13	16	13	17	14	18
AMAZE	WL	-28%	-28%	-9%	-48%	5%	-32%	16%	-14%	24%	-2%
vs DAHHK	Delay	30%	26%	37%	37%	40%	37%	40%	37%	42%	33%
AMAZE	WL	-1%	0%	10%	-11%	22%	0%	28%	9%	36%	18%
vs P-Trees	Delay	0%	0%	11%	3%	18%	5%	26%	8%	20%	-2%

sink capacitance values from libraries currently used in industrial designs. For the source resistance we chose a strong cell, since it will be driving a critical net. This is, in fact, a favorable scenario for the AMAZE algorithm. By increasing the driver resistance by 30$\times$ the AMAZE improvement over DAHHK went down by approximately 1%–8% (proportional to the size of the tree) while the improvement over P-Trees went down by approximately 0%–4%. With this increased driver resistance, the gate delay (driver resistance times total capacitance) was roughly around 10%–25% of the whole delay. In order to compute the Elmore delay, we used a 3ΠRC circuit to model the wires. The validity of the data obtained from the delay model was confirmed by a set of electrical simulations performed on sample trees.

Tables III and IV report the experimental results. In Table III, the trees were randomly generated in a window of 300 μm $\times$ 300 μm. In Table IV the window is reduced for 100 μm $\times$ 100 μm while we excluded 5 critical pins configurations. Additionally, in Table IV, only AMAZE with path length factor set to 1 is used.

The following facts can be observed from Table III.

- AMAZE best setting for delay is AMAZE with path length factor (plf) 1.
- AMAZE scales very well with the addition of a noncritical sink. Note that the delay of configurations with one critical sink is always close to 30 ps.
- AMAZE improvement is better (in delay) for cases with one critical sink. It does not scale well with the addition of other critical sinks. It delivers good results compared to P-Trees and DAHHK with up to three critical sinks.
- AMAZE delay is consistently better than DAHHK (ranging from 26% to 42% in average).
- AMAZE delay for 1 or 3 critical sinks is consistently better than P-Trees (ranging from 1% to 30% in average).
- AMAZE does better on the whole with larger trees.

In Table IV, we can observe similar advantages of AMAZE to P-Trees and DAHHK algorithms.

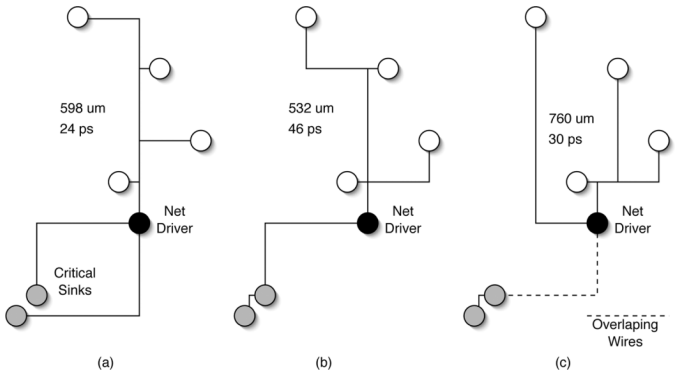


Fig. 11. Best tree for delay of: (a) AMAZE; (b) DAHHK; and (c) P-Trees.

Fig. 11 compares nets produced by the three algorithms. We observe that, in the AMAZE algorithm, the path to the critical sinks has minimum length and minimum sharing, while the rest of the tree is optimized for wire length. The best effort of the

TABLE V
STEINER DELAY COMPARISON WITH BLOCKAGES BETWEEN AMAZE AND P-TREES

Alg	# nodes	5		7		9	
	# critical	1	2	1	2	1	2
AMAZE $pfl = 1$	WL (μm)	595	709	752	935	851	1024
no block	Delay (ps)	33	39	35	43	41	53
AMAZE $pfl = 1$	WL (μm)	646	751	774	962	870	1047
with block	Delay (ps)	36	43	36	44	43	53
P-Trees	WL (μm)	672	692	904	1036	1163	1261
no block	Delay (ps)	36	43	43	53	45	54
P-Trees	WL (μm)	735	733	958	1046	1175	1274
with block	Delay (ps)	38	46	45	53	48	55
AMAZE	WL	-8.5%	-5.7%	-3.0%	-2.7%	-2.1%	-2.2%
Degradation	Delay	8.5%	10.2%	-3.1%	-3.6%	-0.5%	-4.0%
P-Trees	WL	-9.3%	-5.95%	-6.0%	-1.0%	-1.0%	-0.9%
Degradation	Delay	-5.2%	-6.8%	-6.5%	-1.06%	-5.49%	-3.13%
Impr. over	WL	12.1%	-2.37%	19.1%	8.0%	25.96%	17.82%
P-Trees	Delay	6.0%	5.8%	20.4%	16.6%	31.4%	21.91%

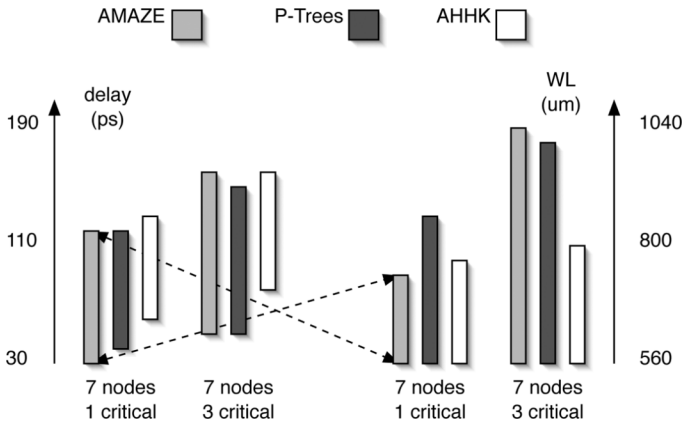


Fig. 12. Delay and wire length range of the studied algorithms. Within this range you can trade WL for delay. The worst case wire length leads to best delay vice versa.

DAHHK algorithm found the minimum path length to all sinks (building an arborescence), however it did not help much for the delay of the critical sinks, since the path is fully shared by both sinks. For the P-Trees algorithm, among 35 different topologies evaluated, the one that was best for minimizing delay to critical sinks has separated wires to the critical sinks. The P-Trees's drawback that impacted the delay is the fact that the overall wire length is too large (affecting the product of driver resistance per total capacitance). Another drawback is the generation of overlapped wires, that need to be somehow resolved in actual routing.

C. WL Versus Delay Tradeoff Analysis

We provide a comparison of the wire length and delay trade-off produced by the three studied algorithms (ours, DAHHK and P-Trees) in Fig. 12. The delay and wire length ranges are average numbers for 7 pin nets with one critical sink. The best delay generally leads to the worst wire length (wl). The algorithms will tradeoff wl for delay in this range.

The following facts can be observed from Fig. 12.

- In the case of one critical sink, AMAZE worst case wire length is better than the other studied algorithms, while delay obtained is the best.
- AMAZE and P-Trees have a wider range to tradeoff wire length and delay, since they can control the amount of sharing more effectively.
- The addition of a critical sink increases significantly the tradeoff range, since extra wire is inserted to isolate the wire.

D. Blockage Analysis

Since AMAZE is a maze router it naturally handles blockages. As a potential problem, we observe that our critical wire isolation approach (sharing factor) will be less effective if a blockage is in the middle of the way. In such case, AMAZE will select the possibility with minimum length and this might lead to a worse topology of the overall tree since the wires will not be shared. We enforce that AMAZE should be executed with varied sharing factors (path length factor can be set to 1) to avoid extremely long wires.

We compared our algorithm to the P-Trees. Since DAHHK trees cannot handle blockage directly we left it out of the comparison. Our experimental setup was as follows: We generated trees ranging from 5 to 9 pins, where 1 to 3 of the sinks were critical. For each configuration we generate 100 trees randomly with one blockage. Given the Hannan grid of the tree, sliced horizontally and vertically in the Hannan points, the blockage is constrained to interrupt at least one of the slices. After that, we run the AMAZE (with $\mathrm{plf} = 1$) and P-Trees algorithms for each of the trees picking the best tree for delay to the critical sinks. We then compare the obtained trees with and without the blockage. If the solution of both algorithms are identical to the non-blockage version, we discard the tree from the experiment set. We observed that half of the generated trees fall into this criteria.

Table V presents all the experimental results where the following facts can be observed: 1) AMAZE is consistently better than P-Trees in average for all tested configurations, from 5% to 32% in delay; 2) besides the better delay of critical sinks, AMAZE could deliver better wire length up to 26%; and 3) both algorithms suffer from a degradation for the blockage insertion. AMAZE degrades less in the cases with one critical sink and more than five pins.

To conclude with, we can observe that AMAZE has some important features that help in the blockage handling.

1) The critical sinks are routed first. This fact prevents that blockage detours made by wires to noncritical sinks eventually change the topology of the critical connection. This feature is not present in the P-Trees.
2) Each sink is routed separately, isolating the detour to the blocked branch only.
3) Even in an extreme case (a very long detour is needed), AMAZE will not change the strategy of isolating the critical wire. It might happen thought that wire length increases too much.

In these cases, the solution is to relax the isolation of critical wires by playing with the sharing factor.

IX. Application to Timing Driven Routing

The techniques presented in this paper called *sharing factor*, *path length factor*, and *biasing* provide means of controlling, respectively, the amount of sharing of critical wires, path length of critical wires and overall wire length. By understanding the impact of those parameters to the delay of critical elements in a circuit and playing with them it is possible to obtain a variety of tree topologies. For instance, an arborescence can be built simply by setting all sinks to critical and the path length factor to 1. A star tree can be built using the sharing factor 0. All intermediate values provide topologies that tradeoff sharing of critical wires, path length and wire length in such a way that an appropriate topology is found.

In our experiments, we target at obtaining a path with minimum delay. For that, our best setting is to fix the path length factor to one (minimum path length) and vary the sharing factor from 0 to 1 in steps of 0.25. This way, 5 different topologies are generated and we simply pick the best for delay (ignoring the delays to noncritical sinks).

The algorithm is also flexible to handle other applications. An arborescence can be generated by setting all sinks as critical and the path length factor to 1. By playing with intermediate values of this factor and providing multiple levels of criticality (as more detailed in Algorithm 2), delays to less critical sinks can be improved. This methodology can be applied to obtain the best tree that satisfies a certain slack.

As regarding speed, our algorithm also provides means of obtaining very fast routing combining existing techniques such as heuristic search and Hannan grid with some new ones such as an improved h function that predicts vias, an improved data structure for the open nodes with constant time insertion and $\log$ time access, an auxiliary open list to delay computation of f function for some *expensive* cases, and a method for fast expansion of nodes in the critical path. To evaluate run time, we performed 415 AMAZE runs on 5, 7, 9, and 15 pin nets and achieved respectively 0.61, 0.94, 1.24, and 2.45 s. We also performed the same experiment inserting one random blockage as described in the previous section obtaining, respectively, 0.69, 1.01, 1.35, and 2.7 s. There is a small degradation with the blockage since the heuristic based on Manhattan distance will not be exact in some cases, demanding some more nodes to be expanded in order to assure the minimal path. The hardware platform used was a Mac Dual G5 with 1 GB of RAM. Due to the speed of our results, we conclude that AMAZE can be used for higher level estimation such as placement and for routing as well.

X. Conclusion and Future Work

In this paper, we addressed the application of several techniques in a path search based algorithm called AMAZE so that it becomes able to compete with state-of-the-art methods for building Steiner trees. AMAZE has also the ability to tradeoff wire length for delay, minimizing the delay for critical sinks while reducing the overall wire length for the rest of the tree. Our biasing technique is the key to achieve wl reduction by maximizing wire sharing. On the other hand, path length and sharing factors were introduced to isolate critical paths so that delay to the identified critical sinks is minimized.

In terms of delay, AMAZE outperformed algorithms used in the industry and in the state-of-the-art academic research, such as AHHK (by 25%-40% in average) and P-Trees (by 1%–30% in average). We also observed that a wide range of topologies are available for the algorithm, which is able to find the appropriate tree according to the needs of the net being routed. On the blockage analysis, AMAZE maintained a good behavior and outperformed P-Trees from 5% to 32%. Congestion could also be incorporated in the algorithm by providing costs to the routing graph. Cost functions should be carefully used since they will reduce the occurrence of critical ties that eventually happen during the A* search, affecting the benefits obtained in this paper by exploring those ties.

Novel properties and data structures of path search that can be used to obtain better run time were demonstrated. Those optimizations include an open list with a bounded number of positions that can be read in constant time. By cleverly combining our contributions with heuristic search, the routing algorithm can take great advantage of the heuristic estimator (behaving like a direct DFS search) and have run times compatible with heuristic Steiner tree algorithms that are considered fast.

We believe that due to their speed and quality, our algorithms will find applications both in obtaining wiring estimates that are required during early design like placement, as well as during the actual routing. For example, applications such as global routing could potentially benefit from the added flexibility and tradeoff provided by our algorithm. Designs dominated by a small number of highly critical paths could benefit greatly by optimizing these paths for delay at the expense of some additional wire length.

Acknowledgment

The authors would like to thank students G. Flach, G. Santos, and G. Wilke for their help with reviewing the paper and student R. Fonseca for the implementation of the Elmore delay. They would also like to thank Dr. M. Hrkic and Dr. J. Lillis for providing them support with the P-Trees, and Dr. C. Alpert and Dr. S. Quay for their help with AHHK.

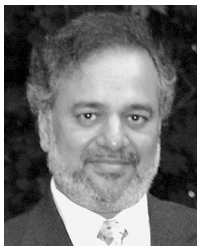
References

[1] C. Lee, "An algorithm for path connections and its applications," *IRE Trans. Electron. Comput.*, pp. 346–365, Sep. 1961.
[2] J. Cong, K.-S. Leung, and D. Zhou, "Performance-driven interconnect design based on distributed rc delay model," in *Proc. 30th Int. Conf. Des. Autom. (DAC)*, New York, 1993, pp. 606–611.
[3] R. F. Hentschke, J. Narasimham, M. O. Johann, and R. L. Reis, "Maze routing steiner trees with effective critical sink optimization," in *Proc. Int. Symp. Phys. Des. (ISPD)*, New York, 2007, pp. 135–142.

[4] S. K. Rao, P. Sadayappan, F. K. Hwang, and P. W. Shor, "The rectilinear steiner arborescence problem," *Algorithmica*, vol. 7, pp. 277–288, Dec. 1992.
[5] J. Cong, A. B. Kahng, G. Robins, M. Sarrafzadeh, and C. K. Wong, "Provably good performance driven routing," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 11, no. 6, pp. 739–752, Jun. 1992.
[6] C. J. Alpert, T. C. Hu, J. H. Huang, A. B. Kahng, and D. Karger, "Prim-dijkstra tradeoffs for improved performance-driven routing tree design," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 14, no. 7, pp. 890–896, Jul. 1995.
[7] J. Cong, C.-K. Koh, and P. Madden, "Interconnect layout optimization under higher order RLC model for mcm designs," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 20, no. 12, pp. 1455–1463, Dec. 2001.
[8] A. Khang and G. Robins, *On Optimal Interconnects for VLSI*. Boston, MA: Kluwer, 1995.
[9] M. Borah, R. M. Owens, and M. J. Irwin, "A fast algorithm for minimizing the Elmore delay to identified critical sinks," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 16, no. 7, pp. 753–759, Jul. 1997.
[10] K. D. Boese, A. B. Kahng, and B. A. McCoy, "Near optimal critical sink routing," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 14, no. 12, pp. 1417–1436, Dec. 1995.
[11] S. S. S. H. Hou and J. Hu, "Non-hanan routing," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 18, no. 4, pp. 436–444, Apr. 1999.
[12] J. Lillis, C.-K. Cheng, T.-T. Y. Lin, and C.-Y. Ho, "New performance driven routing techniques with explicit area/delay tradeoff and simultaneous wire sizing," in *Proc. 33rd Annu. Conf. Des. Autom. (DAC)*, New York, 1996, pp. 395–400.
[13] X. Hong, T. Xue, E. S. Kuh, C.-K. Cheng, and J. Huang, "Performance-driven steiner tree algorithm for global routing," in *Proc. 30th Int. Conf. Des. Autom. (DAC)*, New York, 1993, pp. 177–181.
[14] J. Xu, X. Hong, T. Jing, Y. Cai, and J. Gu, "An efficient hierarchical timing-driven steiner tree algorithm for global routing," in *Proc. Conf. Asia South Pac. Des. Autom./VLSI Des. (ASP-DAC)*, Washington, DC, 2002, p. 473.
[15] S. Dutt and H. Arslan, "Efficient timing-driven incremental routing for vlsi circuits using dfs and localized slack-satisfaction computations," in *Proc. Conf. Des., Autom. Test Eur. 3001 (DATE)*, Leuven, Belgium, 2006, pp. 768–773.
[16] H. S.-W., A. Jagannathan, and J. Lillis, "Timing-driven maze routing," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 19, no. 2, pp. 234–241, Feb. 2000.
[17] S. Prasitjutrakul and W. J. Kubitz, "A timing-driven global router for custom chip design," in *Proc. Int. Conf. Comput.-Aided Des. (ICCAD)*, 1990, pp. 48–51.
[18] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. SSC-4, no. 2, pp. 100–107, Feb. 1968.
[19] M. Johann and R. Reis, "Net by net routing with a new path search algorithm," presented at the 13th Symp. Integr. Circuits Syst. Des., Manaus, Brazil, 2000.
[20] W. C. Elmore, "The transient response of damped linear network with particular regard to wideband amplifiers," *J. Appl. Phys.*, vol. 19, pp. 55–63, 1948.
[21] A. Caldwell, Personal Contact 1997.
[22] M. Hannan, "On steiner problem with rectilinear distance," *SIAM J. Appl. Math.*, vol. 30, pp. 255–265, 1992.
[23] M. Zachariasen, "Rectilinear full steiner tree generation," *Networks*, vol. 33, pp. 125–133, 1999.
[24] R. Kastner and M. Sarrafzadeh, "Labyrinth," Jan. 2005 [Online]. Available: http://www.ece.ucsb.edu/~kastner/labyrinth
[25] C. J. Alpert, T. Chan, D. J.-H. Huang, I. Markov, and K. Yan, "Quadratic placement revisited," in *Proc. 34th Annu. Conf. Des. Autom. (DAC)*, New York, 1997, pp. 752–757.
[26] C. J. Alpert, A. B. Kahng, C. N. Sze, and Q. Wang, "Timing-driven steiner trees are (practically) free," in *Proc. 43rd Annu. Conf. Des. Autom. (DAC)*, New York, 2006, pp. 389–392.

Renato Hentschke received the B.S., M.S., and Ph.D. degrees in computer science from Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil.

He is currently with Intel Corporation, Hillsboro, OR. Prior to Intel, he worked for six months as a temporary Professor with State University of Rio Grande do Sul, Guaiba, RS, Brazil, and interned at IBM, Yorktown Heights, NY, for nine months. He holds one U.S. patent and is a coauthor of 20 papers in the field of computer-aided design of ICs. He is a reviewer of prestigious conferences such as DAC and ICCAD. His research interests include algorithms for physical design of VLSI circuits, such as placement, routing, and layout verification.



Jagannathan Narasimhan received the B.Tech. degree in mechanical engineering and the M.Tech. degree in industrial engineering and operations research from the Indian Institute of Technology Kharagpur, India, in 1977 and 1979, respectively, and the Masters degree in computer science from Texas Tech University, Lubbock, in 1982, and the Ph.D. degree in electrical engineering from the University of Maryland, College Park, in 1992.

He has worked for several years as an Operations Research Analyst for an airline and as an operating systems developer for Burroughs Corp. before starting his doctoral work, and for the last 16 years he has worked in the area of CAD for VLSI at IBM Research, Yorktown Heights, NY. He has published on topics that include various areas of CAD for VLSI, fault tolerant computing, etc. He holds several patents in related areas as well. His current research interests include combinatorial and parallel algorithms and their applications to synthesis and physical design.

Marcelo Johann (M'00) received the Bachelor's, Master's, and Ph.D. degrees in computer science from the Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil, in 1992, 1994, and 2000, respectively, having spent six months at the University of California at Los Angeles (UCLA).

He worked as a Professor with the Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil, from 2000 to 2002, and is a full-time Professor at UFRGS since 2003. He coauthored 6 book chapters and published 30 conference papers in topics related to computer aided design of ICs. His research interests include algorithms for placement and routing, combinatorial optimization, and his teaching activities include also operating systems, game programming, and computer music.

Ricardo Reis (M'81–SM'06) studied electrical engineering at Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil, in 1978. He received the Ph.D. degree from the Institut National Polytechnique de Grenoble, Grenoble, France, in 1983.

He has been a full Professor with the Federal University of Rio Grande do Sul, Porto Alegre, Brazil, since 1981. He is research level 1 of the CNPq (Brazilian National Science Foundation) and head of several research projects supported by Government Agencies and Industry. Since January 2008, he has been the Vice President of the IEEE Circuits and Systems representing Region 9 (Latin America). He has published more than 300 papers in journals and conferences proceedings. He has also authored and coauthored several books. His research interests include physical design, physical design automation, design methodologies, digital design, CAD, circuits tolerant to radiation, and microelectronics education.