A Layout Advisor for Timing-Critical Bus Routing[1]

Wei Huang and Andrew B. Kahng
UCLA Computer Science Dept., Los Angeles, CA 90095-1596 USA

Abstract

We describe a "topology advisor" for routing of critical (multi-source) buses in building-block design. The tool accepts as input a block layout, a two-layer routing cost structure superposed over the block layout, terminal locations for a multi-source bus, and source-sink delay upper bound (linear or Elmore delay) constraints for all terminal pairs. The b best routing solutions (b a user parameter) that satisfy all constraints are returned. Efficient implementations of exhaustive search are used to guarantee optimal results when practical, and otherwise yield fast, high-quality results (if the problem is large or if the constraints are loose). Practical features include: (i) modeling of per-region and per-layer routing costs, (ii) routing to terminals located inside blocks, (iii) optional splitting of k-pin bus routes when the optimal routing passes through narrow channels, and (iv) heuristic speedups based on clustering and sampling.

1 Introduction

In today's deep-submicron technologies, global interconnect planning pervades the design process. A typical RTL-down chip implementation methodology will create a hierarchical physical floorplan and perform mixed hard-soft block placement that is heavily driven by budgeting (timing, area, power) and estimation. Implicit in chip planning is the *route planning* of all global nets (control and data flow, clock, and power/ground), subject to performance constraints and potentially large variations in local routing resources.

The route planning essentially determines all relevant interconnect characteristics needed to make correct design planning choices. The chip planner will use route planning results to (i) modify the chip floorplan (floorplan compaction, and pin assignment derived from the top-level route planning); (ii) determine new synthesis constraints (budgets for intra-block delay, block input/output boundary conditions); (iii) modify the netlist itself (driver sizing, repeater insertion, buffer clustering); (iv) determine placement directives for block layout (over-block routes will locally affect utilization factors within blocks); and (v) determine performance-driven routing directives for block layout (wire tapering, spacing, shielding, etc.). Route planning thus entails modeling of hierarchical and area pins, understanding of power-area-delay tradeoffs in both devices and interconnects, and the ability to perform and verify "intelligent" bus routing, timing- and signal integrity-driven routing, repeater insertion, tapering, shielding, interleaving, etc.

[1] This work was supported by a grant from Cadence Design Systems, Inc. Andrew B. Kahng is currently Visiting Scientist (on sabbatical leave from UCLA) at Cadence. Author contact information: Andrew B. Kahng, UCLA Computer Science Dept., 3713 Boelter Hall, Los Angeles, CA 90095-1596 USA. E-mail: abk@cs.ucla.edu, Tel. 310-206-7073, Fax 310-825-7578.

Our work centers on the timing- and resource-constrained (multi-source) bus routing aspect of full-chip route planning. We believe that this problem will become increasingly important in top-down reuse-centric ASIC methodologies. Previous literature on the multi-source delay-constrained routing problem is limited. Performance-driven multi-source routing is NP-complete [6] even without obstacles. Linear programming has been used to construct lower and upper bounded delay routing trees with a single source, under the linear delay model [12]; this work requires an input topology. For multi-source routing, [5] heuristically generates minimum-cost minimum-diameter Steiner arborescences; [4] wiresizes such routing trees via decomposition into source and loading subtrees. None of these algorithms for multiple-source net routing accept path delay bounds as constraints. In the presence of obstacles, Ganley and Cohoon [7] perform minimum-wirelength routing using exhaustive search for small nets and a k-Steinerization heuristic for large nets.

2 A Bus Routing Advisor

Our new "topology advisor" for intelligent bus routing can provide routing solutions for multi-source buses that are used to time-share scarce global interconnect resources. The tool is also aware of channel width requirements for wide buses. Input consists of a rectangular block layout (non-rectilinear blocks can be modeled as unions of rectangular blocks), a two-layer (per-region, per-layer) routing cost structure superposed over the block layout, terminal locations that can be within blocks, and source-sink delay (Elmore or wirelength) upper bounds between terminal pairs. The returned output consists of the best b routing solutions that satisfy delay constraints and are subgraphs of the *escape graph* [7]. Here, b is a user parameter and "best" is also user-defined (typically, we seek either the minimum weighted-cost tree satisfying delay bounds, or the tree that has maximum *delay slack* (defined as the smallest difference between delay upper bound and actual path delay, over all source-sink paths). The fundamental approach is exhaustive search, with various implementation details that help reduce runtimes. Practical extensions include optional splitting of k-bit wide bus routes when the optimal routing passes through narrow channels, and heuristic speedups based on clustering and Steiner point sampling. Our runtimes are on the order of a few CPU seconds on a Sun Ultra-1 (140MHz) for a 6-terminal net in a 20-block layout, assuming that the constraints are difficult (but possible) to satisfy. An overview of the approach is given in Figure 1. The remainder of this section discusses several key implementation details.

Topology Generation

To find an exact solution, we test all possible trees. To do this, we first must list all topologies. In a *full Steiner*

Procedure:

Input: A layout with B blocks and n pins
Upper-bounded delay constraints between each source-sink pin pair
User defined parameter b and "best"

Output: b best routing solutions

1. Generate all full-topologies for n pins.
2. Generate the escape graph from blocks and pins.
3. Compute shortest distances between all pairs of points in the escape graph.
4. Generate distance function and reduce constraints.
5. Prune list of feasible points for path between each pair of terminals.
6. For each topology
7. For each Steiner point s
8. Read in full, one_less and pairs list
9. Reduce the candidate list for s according to the *full*, *one_less* and *pairs* lists.[1]
10. If s has no remaining candidate points, delete the topology.
12. For each remaining topology
13. Check each combination of all candidates of each Steiner point.
14. Output the certain number of "best" trees depend on user defined cost function.

Figure 1: Overview of Procedure

topology, each Steiner point is adjacent to exactly three other nodes. There are $n-2$ Steiner points in a full topology over n terminals. By an early result of Gilbert and Pollak [8] stating that every topology has an underlying full topology, we need test only full topologies.

Given the full Steiner topologies over n terminals, the full topologies over $n+1$ terminals are obtained by adding the new Steiner point into every possible edge. There are $2n-3$ edges for a full Steiner topology over n terminals, so there are $2n-3$ ways to add the new Steiner point. The recurrence is therefore $f(n+1) = f(n) * (2n-3)$, e.g., $f(4) = 3$, $f(5) = 15$, $f(6) = 105$, where $f(n)$ is the number of full topologies for an n-terminal tree. We perform actual generation of full Steiner topologies according to this recurrence.

Escape Graph Generation and Costing

For a given block layout with obstacles, we use line-sweep to generate an *escape graph* as in Cohoon and Richards [3]. The escape graph is quite similar to the standard notion of a channel intersection graph; we assume that routing must be performed within resources (e.g., channels) corresponding to edges in this graph. Note also that our tool allows terminals inside of obstacles, e.g., for route planning in presynthesis floorplanning. Escape graph generation has complexity $O(max(n, m \log m))$ where n is the number of intersections of segments in the escape graph, and m is the number of obstacle boundary segments. In the worst case, n is $O(m^2)$.

We accommodate varying per-layer (h- and v-layer), per-region routing costs. These costs, along with the region structure, are specified via an auxiliary input file. Since we restrict the solution to be a subgraph of the escape graph, without loss of optimality we can simply "overlay" the routing cost structure onto the escape graph edges.[2] For example, to determine the routing cost of a given vertical edge, we determine the regions it crosses and sum the costs of the segment's intersections with all regions, i.e., its vertical length in each region multiplied by the per-unit vertical routing cost of for that region. Routing costs for horizontal edges are similarly computed.

All Pairs Min-Jog Shortest Paths

For each topology, our approach tests all assignments of candidate Steiner point locations to Steiner points of the topology. Thus, we must store all shortest paths between pairs of terminals and candidate Steiner point locations in the escape graph.[3] If via costs are not modeled, the program will unrealistically consider all monotone staircase paths to be equally good. To generate all pairs shortest paths with minimal jogging, we assign jogs a small cost; path reconstruction is enabled by assigning horizontal segments $1+\epsilon$ times their original cost, and vertical segments $1-\epsilon$ times their original cost, where ϵ is a small positive constant.

We use each point (including both Steiner points and terminals) as a source s and perform breadth-first search (BFS). For each point p, we use $p.left[s]$ ($p.right[s]$, $p.top[s]$, $p.bottom[s]$) to store the minimum cost of any path from s to p that arrives at p from the left (right, top, bottom). The BFS starts with the source s in *test_list*. After a point has been visited, we update the values of its children, and put the new point and any updated points with new values into the queue. When the queue is empty, for each point p the shortest path cost $apsp[s][p]$ from the source s is the minimum value among $p.left[s]$, $p.right[s]$, $p.top[s]$ and $p.bottom[s]$.

If we know that points s and p are connected by a path of length $apsp[s][p]$, we reconstruct this path as follows. The values of $p.left[s]$, $p.right[s]$, $p.top[s]$ and $p.bottom[s]$ are checked. If $p.left[s]$ is equal to $apsp[s][p]$, p's left neighbor $p1$ is added into the path, and $p1$ is checked against the value $v = apsp[s][p] - d[p][p1]$. For $p1.top$ and $p1.bottom$ the jog cost is considered, i.e., we check whether $p.left[s]$ or $p.right[s]$ is equal to v, and whether $p1.top$ or $p1.bottom$ is equal to $v - jog\_cost$. The path reconstruction ends when s is reached.

Steiner Candidate Reduction and Constraint Checking

A typical escape graph for 20 blocks will have well over 100 candidate Steiner points. For a six-terminal net, there are 105 full Steiner topologies, with four Steiner points in each topology. If we test all possibilities, there are 100^4 possibilities for each topology, implying about $105 * 10^8 \approx 10^{10}$ possible trees. Clearly, we must to reduce the number of candidate Steiner points.

For each pair of terminals, we generate a list of possible Steiner points between them. Under the linear delay model, if the shortest distance between terminal a and Steiner candidate s, plus the shortest distance between terminal b and s, is greater than the pathlength upper bound for a and b, then s cannot be a Steiner point on the a-b path in the tree.[4]

To improve the program speed, we precompute and store lists of all possible Steiner points between each terminal and all other terminals. We also precompute lists of all possible Steiner points between each terminal and *all except one* other terminals. The latter are generated by intersecting

[2] A simple extension of Hanan's 1968 result for rectilinear Steiner minimal trees is that the minimum-cost tree is a subgraph of a "Hanan grid" induced by terminal locations, block and routing region boundaries, as well as boundaries between different-cost regions. See, e.g., [10] for a review.

[3] This implies that our approach is correct only for *monotone* delay models, where increasing (decreasing) any given segment length will increase (decrease) signal delay.

[4] Under the Elmore model, delay between a and b (between b and a) using s as the only Steiner point between them must be less than the a-b (b-a) delay upper bound. If not, again s cannot be a Steiner point on the a-b path in the tree.

the lists of possible Steiner points between each pair of terminals.

When we generate a full topology, we record all Steiner points on the path between each pair of terminals, i.e., for each Steiner point s, we save a list of pairs of terminals whose path between them can contain s. We save this list in three parts, called *full*, *one_less* and *pairs*. If every path from terminal a to another terminal can contain s, then a is saved into the *full* list of s. If every path from terminal a to another terminal – except for the path to terminal b – can contain s, then (a, b) is saved into the *one_less* list of s. All other pairs of terminals for which the connecting path can contain s, but which are not yet included in the *full* or *one_less* lists, are saved into the *pairs* list of s. This preprocessing is done as the topologies are generated.

Finally, when we read in a topology, a list of candidate Steiner locations (points) for each Steiner node is generated from the above information.[5]

For any topology, if the candidate list of one of its Steiner points becomes empty, the topology can be removed from consideration. For each remaining topology, we have a list of candidate points for every Steiner point. We test all combinations of candidates for every Steiner point, calculating delays between each pair of terminals, and checking whether the input delay constraints are violated.[6] The b best valid trees are saved, and can be returned to the user in order of user parameter b, e.g., decreasing slack or increasing cost.

3 Experimental Results

The bus routing advisor has been implemented in C++ in the Sun UNIX environment; it compiles and runs under Solaris 2.5.1 and the Sun CC4.2 compiler, as well as the g++ 2.7.2 compiler. All runtimes that we report are for a Sun Ultra-1 (140 MHz).

Our main experiments address the scaling of runtimes with instance complexity as measured by number of blocks, number of terminals, and tightness of delay constraints. We do not report experiments involving per-region and per-layer routing costs, since these have little effect on runtime scaling. We also report experimental results only for linear delay constraints, although our program handles Elmore delay constraints as well. The random block layouts are generated by shrinking a random non-slicing floorplan. Terminals are placed randomly in blocks, at most one terminal per block. The *tightness* value (T) of constraints is varied as follows. We randomly choose a given number of pin pairs (if pair a-b is picked, b-a must be picked), and set the pathlength delay upper bound for that pair as T times the estimated pin-to-pin delay for that pair. Under the linear delay model, the estimated delay for pair a-b is the cost of the shortest path between them. Under the Elmore delay model, we place a Steiner point at the midpoint of the a-b path; in estimating the a-b delay, we then consider the subtree capacitance effect from a lower bound on the Steiner tree cost over all terminals except a and the midpoint. Every other pathlength upper bound constraint is set according to a default tightness equal to 3.0 times the cost of the the shortest path between the given terminal pair.

Figure 2 shows how the optimal multi-source routing solution changes as different path constraints are made tight. Table 1 reports average runtimes (RT) required to find the best 100 routing trees for random instances with prescribed numbers of blocks and terminals, and prescribed tightness of constraints. The table also reports treelength (TL) of the best solution normalized to the corresponding treelength for the same instance with "loosest" constraints. We report *multiple-source* nets having 5 and 6 pins, in 12- and 25- block layouts. More detailed results, including those for *single-source* nets are in [9]. Different numbers of tight pairs (/1, /2, ...) and tightness values ($T = 1.1$, $T = 1.2$, $T = 1.3$) and tightness values are tested.

Specific numbers shown in each table are the average (minimum, maximum) values taken over ten test cases. Runtime [RT] is in seconds, and treelength [TL] is normalized to the corresponding treelength for the loosest constraints (one tight pair with tightness 1.3 for 5-pin nets, and two tight pairs with tightness 1.3 for 6-pin nets).

Runtimes decrease rapidly as more terminal pairs have tight constraints, since more Steiner candidates are eliminated. Also, average treelength increases as the constraints become tighter. For 6-pin *multi-source nets* in 25-block layouts with tight constraints our method can find an optimal tree in a couple of seconds. But when the constraints are very loose, the same size problem can require up to 1500 seconds.

Block,Pin/	T = 1.1		T = 1.2		T = 1.3	
#tight pairs	RT	TL	RT	TL	RT	TL
12,5/1	4.13	1.03	5.19	1.00	5.85	1.00
12,5/2	1.37	1.05	1.79	1.02	2.22	1.01
12,5/3	1.07	1.06	1.27	1.02	1.55	1.01
12,5/4	0.97	1.11	1.06	1.05	1.32	1.03
25,5/1	15.14	1.03	17.78	1.00	20.25	1.00
25,5/2	5.96	1.03	8.35	1.01	10.36	1.00
25,5/3	5.51	1.03	6.59	1.01	7.77	1.00
25,5/4	5.49	1.03	6.12	1.01	6.96	1.00
12,6/1	49.70	1.02	130.91	1.00	267.48	1.00
12,6/2	24.21	1.02	48.72	1.01	90.16	1.00
12,6/3	2.72	1.05	10.74	1.01	41.68	1.00
12,6/4	1.86	1.07	6.50	1.02	24.83	1.01
25,6/1	125.74	1.02	900.08	1.00	1530.40	1.00
25,6/2	20.41	1.03	135.93	1.02	312.42	1.01
25,6/3	8.46	1.04	50.64	1.03	130.50	1.02
25,6/4	7.07	1.07	12.35	1.04	30.61	1.02

Table 1: Runtime [RT] and treelength [TL] normalized to corresponding treelength for 1/2 tight pair (for 5- / 6-pin nets) with tightness 1.3.

Steiner Point Sampling by K-Center Partitioning

Particularly under the Elmore delay model, large nets with loose constraints cause long runtimes, since very few topolo-

[5] There is a small difference for the case of the Elmore delay model, since off-path subtree capacitance is a major contributor to path delay. For each Steiner point, the *full* and *one_less* lists are still used. However, we also generate a *two_less* list, which contains $(a, (b, c))$ when the Steiner point s can be on paths between terminal a and all other points *except* terminals b and c. Suppose a is in the *full* list, and we want to test if s can be this Steiner point. A lower bound on the Steiner tree cost over s and all terminals but a is generated using all-pairs shortest paths distances (metric closure of the escape graph distances). By the 1981 result of Kou, Markowsky and Berman [11], the length of the Steiner minimal tree is at least $1/(2*(1-1/L))$ (L is the number of terminals of the tree) times the length of the minimum spanning tree (MST) in the metric closure of the escape graph. We use $1/(2*(1-1/L))$ times the length of the MST as a lower bound on the length of the subtree rooted at point s.

[6] We work only with the *metric closure* of pathlength upper bounds. In other words, we require that for any three points a, b and c, $constraint[a][c]$ is less than or equal to the sum of $constraint[a][b]$ and $constraint[b][c]$; this is because we can always reach c from a, via b. Taking the metric closure reduces large constraints and helps the Steiner candidate reduction.

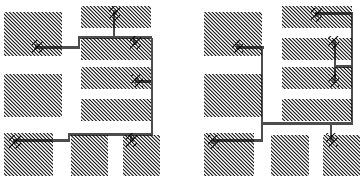

Figure 2: Different routing trees are found when different pin pairs have tight constraints.

gies or Steiner candidates can be eliminated. In such cases, we use *Steiner point sampling* to speed up the search. The K-center partitioning heuristic of Gonzalez [1] is used to form a geometric clustering of all candidate Steiner points. This heuristic randomly picks a "center", then iteratively picks as its next center the point with maximum distance to the nearest of all previously picked centers. When K centers have been chosen, each point p is clustered with the closest of the K centers. (The idea is that the K-center heuristic yields clusters that have reasonably small diameter.) The clustering is performed before topology checking.

During topology checking, if a Steiner point has a very large candidate list, the candidate points are clustered according to the pre-computed K-center clustering, i.e., into at most K groups. For each group, the point closest to the cluster center is chosen as the *representative point*. We then test only the representative points. Then, when a feasible tree is found, we test all combinations of points in the groups corresponding to the representative points used in this feasible tree. If there are too many combinations, we use local search as follows. Replace a given representative point in the tree, by each of the points in its respective group, and save the point *yielding the best results* as the new representative point of the group. Repeat this step for the other representative points, and their groups, cycling until no further routing cost reduction is possible. Figure 3 shows the cost/runtime tradeoff obtained using different values of K.

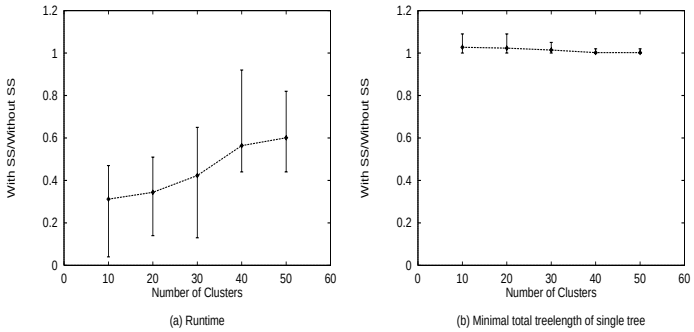

Figure 3: Using different numbers of clusters for Steiner sampling, versus no Steiner sampling. The x-axis gives the number of clusters, and the y-axis gives the relative cost ratio of using Steiner sampling versus no Steiner sampling. We tested ten randomly generated examples. The y-values on the curves are the average value, and the error bars represent the minimum and maximum values. (a) Comparison of runtimes. (b) Comparison of minimum treelength found.

In Tables 2 and Table 3, we compare K-center Steiner sampling with testing of all combinations of candidate Steiner points (A), versus K-center Steiner sampling with local search (L). We report *multi-source* nets having 5 and 6 pins, in 12- and 25- block layouts. More detailed results including *single-source* nets are in [9]. Different numbers of tight pairs and tightness values are tested. The numbers we show in the table are average (minimum, maximum) values over ten test cases. Runtime divided by optimal runtime [RT] and treelength divided by optimal treelength [TL] are also shown in the tables.

From the results, we see that when the constraints are loose, K-center Steiner sampling yields much shorter runtimes; using local search combined with K-center Steiner sampling yields further reductions. Both speedups work better for 6-pin nets than 5-pin nets; in the loosest case for 6-pin nets in 25-block layouts, both speedups together can yield up to 90% runtime improvement on average. Average treelength is 1-3% higher than optimal treelength for K-center Steiner sampling with testing of all combinations of candidate Steiner points; the average treelength is 3-6% higher than optimal treelength for K-center Steiner sampling with local search; and for one case we observed 39% greater than optimal treelength. When we have more tight pairs, neither speedup can reduce runtime significantly, and worst case treelength can be 15% greater than optimal treelength. However, runtimes without the speedups are quite fast.

Tight pairs		12-block layouts		
		T=1.1	T=1.2	T=1.3
1,A	RT	0.66 (0.29,0.97)	0.63 (0.25,0.99)	0.59 (0.20,0.94)
	TL	1.00 (1.00,1.01)	1.02 (1.00,1.10)	1.03 (1.00,1.13)
1,L	RT	0.56 (0.19,1.01)	0.49 (0.17, 1.01)	0.44 (0.12, 0.93)
	TL	1.03 (1.00, 1.11)	1.06 (1.00,1.12)	1.06 (1.00,1.13)
2,A	RT	0.86 (0.71,1.02)	0.74 (0.49,0.95)	0.67 (0.40,0.86)
	TL	1.01 (1.00,1.05)	1.03 (1.00,1.10)	1.01 (1.00,1.06)
2,L	RT	0.86 (0.64,1.06)	0.71 (0.47,0.93)	0.62 (0.40,0.93)
	TL	1.01 (1.00,1.05)	1.04 (1.00,1.11)	1.02 (1.00,1.06)
3,A	RT	0.93 (0.81,1.02)	0.87 (0.74,0.95)	0.81 (0.63,0.94)
	TL	1.01 (1.00,1.04)	1.02 (1.00,1.10)	1.02 (1.00,1.10)
3,L	RT	0.91 (0.76,1.00)	0.88 (0.75,1.13)	0.77 (0.63,0.97)
	TL	1.01 (1.00,1.04)	1.03 (1.00,1.11)	1.02 (1.00,1.10)
4,A	RT	0.91 (0.74,0.97)	0.90 (0.80,0.97)	0.83 (0.61,1.05)
	TL	1.00 (1.00,1.03)	1.03 (1.00,1.14)	1.02 (1.00,1.06)
4,L	RT	0.91 (0.71,0.97)	0.91 (0.82,1.01)	0.83 (0.63,1.01)
	TL	1.00 (1.00,1.03)	1.03 (1.00,1.14)	1.02 (1.00,1.06)
Tight pairs		25-block layouts		
		T=1.1	T=1.2	T=1.3
1,A	RT	0.80 (0.18,1.11)	0.79 (0.26,0.95)	0.74 (0.26,0.93)
	TL	1.02 (1.00,1.07)	1.02 (1.00,1.06)	1.04 (1.00,1.13)
1,L	RT	0.70 (0.06,0.97)	0.66 (0.06,0.94)	0.61 (0.06,0.92)
	TL	1.04 (1.00,1.12)	1.05 (1.00,1.16)	1.05 (1.00,1.13)
2,A	RT	0.96 (0.85,1.17)	0.87 (0.74,0.98)	0.81 (0.52,0.94)
	TL	1.01 (1.00,1.07)	1.03 (1.00,1.14)	1.05 (1.00,1.14)
2,L	RT	0.90 (0.67,1.01)	0.79 (0.24,1.01)	0.70 (0.18,0.94)
	TL	1.02 (1.00,1.07)	1.04 (1.00,1.14)	1.07 (1.00,1.14)
3,A	RT	0.94 (0.91,0.96)	0.91 (0.78,0.97)	0.91 (0.61,0.99)
	TL	1.01 (1.00,1.06)	1.04 (1.00,1.14)	1.05 (1.00,1.14)
3,L	RT	0.93 (0.87,0.97)	0.88 (0.42,1.10)	0.81 (0.29,0.97)
	TL	1.01 (1.00,1.06)	1.04 (1.00,1.14)	1.06 (1.00,1.14)
4,A	RT	1.02 (0.77,1.30)	0.94 (0.90,1.00)	0.94 (0.75,1.23)
	TL	1.02 (1.00,1.10)	1.02 (1.00,1.08)	1.04 (1.00,1.14)
4,L	RT	0.93 (0.78,0.99)	0.88 (0.48,0.99)	0.87 (0.33,1.22)
	TL	1.02 (1.00,1.10)	1.02 (1.00, 1.08)	1.05 (1.00,1.14)

Table 2: Runtime and treelength comparison for 5-pin *multi-source nets* in 12- and 25-block layouts with Steiner sampling.

Tight pairs		12-block layouts		
		T=1.1	T=1.2	T=1.3
2,A	RT	0.36 (0.18,0.69)	0.29 (0.07,0.61)	0.24 (0.09,0.40)
	TL	1.02 (1.00,1.07)	1.03 (1.00,1.19)	1.03 (1.00,1.09)
2,L	RT	0.27 (0.02,0.70)	0.19 (0.01,0.52)	0.09 (0.00, 0.27)
	TL	1.03 (1.00,1.12)	1.05 (1.00,1.19)	1.06 (1.00,1.14)
3,A	RT	0.69 (0.20,1.28)	0.47 (0.17,0.90)	0.35 (0.18,0.59)
	TL	1.01 (1.00,1.03)	1.03 (1.00,1.06)	1.03 (1.00,1.09)
3,L	RT	0.66 (0.02,1.28)	0.39 (0.02,0.90)	0.24 (0.01,0.54)
	TL	1.01 (1.00,1.03)	1.04 (1.00,1.11)	1.06 (1.00,1.13)
4,A	RT	0.87 (0.52,1.04)	0.68 (0.39,0.97)	0.51 (0.16,0.66)
	TL	1.03 (1.00,1.15)	1.04 (1.00,1.19)	1.03 (1.00,1.09)
4,L	RT	0.87 (0.55,1.04)	0.69 (0.39,0.99)	0.43 (0.03,0.66)
	TL	1.03 (1.00,1.15)	1.04 (1.00,1.19)	1.03 (1.00,1.09)
5,A	RT	0.94 (0.74,1.17)	0.76 (0.43,0.98)	0.61 (0.18,0.92)
	TL	1.03 (1.00,1.13)	1.02 (1.00,1.05)	1.03 (1.00,1.09)
5,L	RT	0.94 (0.73,1.21)	0.76 (0.43,0.98)	0.53 (0.04,0.90)
	TL	1.03 (1.00,1.13)	1.02 (1.00,1.05)	1.03 (1.00,1.09)
Tight pairs		25-block layouts		
		T=1.1	T=1.2	T=1.3
2,A	RT	0.47 (0.10,0.89)	0.34 (0.09,0.68)	0.26 (0.05,0.56)
	TL	1.01 (1.00,1.04)	1.02 (1.00,1.06)	1.02 (1.00,1.06)
2,L	RT	0.30 (0.02,0.93)	0.15 (0.00,0.58)	0.09 (0.00,0.50)
	TL	1.03 (1.00,1.06)	1.07 (1.00,1.21)	1.10 (1.00,1.21)
3,A	RT	0.69 (0.18,0.99)	0.54 (0.11,0.92)	0.34 (0.05,0.82)
	TL	1.03 (1.00,1.15)	1.02 (1.00,1.09)	1.02 (1.00,1.05)
3,L	RT	0.62 (0.07,0.91)	0.43 (0.01,0.90)	0.23 (0.00,0.78)
	TL	1.03 (1.00,1.15)	1.05 (1.00,1.13)	1.09 (1.01,1.20)
4,A	RT	0.88 (0.54,1.01)	0.74 (0.34,0.98)	0.42 (0.22,0.77)
	TL	1.03 (1.00,1.16)	1.02 (1.00,1.09)	1.02 (1.00,1.05)
4,L	RT	0.88 (0.53,1.00)	0.68 (0.02,1.00)	0.34 (0.01,0.77)
	TL	1.03 (1.00,1.16)	1.03 (1.00,1.10)	1.06 (1.00,1.15)
5,A	RT	0.96 (0.85,1.19)	0.75 (0.31,1.01)	0.60 (0.21,1.01)
	TL	1.03 (1.00,1.16)	1.03 (1.00,1.09)	1.04 (1.00,1.14)
5,L	RT	0.94 (0.86,1.01)	0.74 (0.18,1.03)	0.55 (0.13,0.99)
	TL	1.03 (1.00,1.16)	1.04 (1.00,1.11)	1.05 (1.00,1.14)

Table 3: Runtime and treelength comparison for 6-pin *multi-source nets* in 12- and 25-block layouts with Steiner sampling.

4 Extensions and Conclusions

We have developed efficient implementations of exhaustive search and heuristic constructions for timing critical bus routing. An exhaustive search algorithm is designed for timing critical multi-source small nets (5- or 6- pin), which is a typical size in practice. For loosely constrained small nets, we give some speedup methods.

Ongoing work seeks a number of other improvements. For example, the next version of our advisor will handle delay lower bounds and give more detailed geometric embedding advice to the actual bus router (e.g., layer assignment, and choice of via stagger pattern when the bus routing makes a bend). Other capabilities have already been implemented, notably the automatic splitting of wide buses to improve performance: when some channels are too narrow for a w-bit bus, we automatically split it into two $w/2$-bit buses and attempt to route them separately while observing delay constraints (Figure 4).[7]

[7] Details are as follows. In escape generation, we delete all segments with width less than $w/2$ tracks; then all trees are guaranteed to handle $w/2$ or more bits. All feasible trees are saved in a list sorted by cost, and the best tree is combined with itself, then with the the next-best tree, etc. For two $w/2$-bit trees to be compatible in routing a w-bit bus, each common channel must have width $\geq w$ tracks. We maintain a channel-usage list for each tree and check the intersection of these lists against a list of channels that have width $\geq w$ tracks.

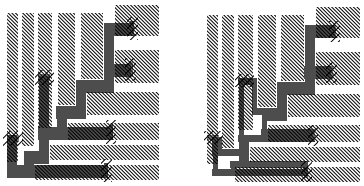

Figure 4: Routing of a wide bus when bus splitting is not allowed (left) and when it is allowed (right).

B	5-pin			6-pin		
	S	RT	TL	S	RT	TL
12	0.70	0.63	1.09	0.50	0.08	1.21
15	0.60	0.78	1.19	0.33	0.41	1.15
20	0.50	0.45	1.14	0.33	0.18	1.11
25	0.33	0.54	1.11	0.14	0.03	1.19

Table 4: Success rate (S), runtime (RT) and minimum treelength (TL) ratios (without WBS, divided by with WBS). B is the number of blocks in the layout examples. S gives the number of instances for which routing without WBS returned a feasible tree, divided by the number of instances for which routing with WBS returned a feasible tree. RT is the average of the analogous runtime ratios, TL is the average of the analogous minimum treelength ratios.

REFERENCES

[1] C.J. Alpert and A.B. Kahng, "Geometric Embeddings for Faster and Better Multi-Way Netlist Partitioning", in *Proc. ACM/IEEE Design Automation Conf.*, Dallas, 1993, pp. 743-748.

[2] K.D. Boese, A.B. Kahng, "Zero-Skew Clock Routing Trees With Minimum Wirelength", in *Proc. IEEE International Conf. on ASIC*, 1992, pp. 1.1.1-1.1.5.

[3] J.P. Cohoon, D.S. Richards, "Optimal Two-Terminal alpha-beta Wire Routing", in *Integration: the VLSI Journal*, 1988(6), pp. 35-57.

[4] J. Cong, L. He, "Optimal Wiresizing for Interconnects with Multiple Sources", in *Proc. IEEE Int'l Conf. on Computer-Aided Design*, 1995, pp. 568-574

[5] J. Cong, P.H. Madden, "Performance-Driven Routing with Multiple Sources", *Proc. IEEE ISCAS*, 1995, pp. 203-206

[6] M. Farach, S. Kannan, T. Warnow, "A Robust Model for Finding Optimal Evolutionary Tree", in *Algorithmica*, 1995, pp. 155-179

[7] J.L. Ganley, J.P. Cohoon, "Routing a Multi-Terminal Critical Net: Steiner Tree Construction in the Presence of Obstacles", in *Proc. of IEEE International Symposium on Circuits and Systems*, New York, 1994(1), pp. 113-116.

[8] E.N. Gilbert, H.O. Pollak, "Steiner Minimal Trees", in *SIAM J. Appl. Math.*, 16(1), 1968, pp. 1-29.

[9] W. Huang, "A Layout Advisor for Timing Critical Bus Routing", M.S. thesis, Computer Science Dept., UCLA, 1997.

[10] A. B. Kahng, G. Robins, *On Optimal Interconnections for VLSI*, Kluwer, 1994.

[11] L. Kou, G. Markowsky, L. Berman, "A Fast algorithm for Steiner trees", in *Axta Information*, 15(2), 1981, pp. 141-145.

[12] J. Oh, I. Pyo, M. Pedram, "Constructing Lower and Upper Bounded Delay Routing Trees Using Linear Programming", in *Proc. European Design and Test Conf.* 1996, pp. 244-249.

[13] N. Saitou, M. Nei, "The Neighbor-joining Method: A New Method for Reconstructing Phylogenetic Trees", in *Mol. Biol. Evol.*, 4(4), 1987, pp. 406-425.