Buffered Steiner Trees for Difficult Instances

C. J. Alpert[1], M. Hrkic[2], J. Hu[1], A. B. Kahng[3], J. Lillis[2], B. Liu[3],
S. T. Quay[1], S. S. Sapatnekar[4], A. J. Sullivan[1], P. Villarrubia[1]
1 IBM Corp., Austin, TX 78758
2 University of Illinois at Chicago, EECS Dept., Chicago, IL 60607
3 University of California at San Diego, CS Dept., San Diego, CA 92093
4 University of Minnesota, ECE Dept., 55455

Abstract

Buffer insertion has become an increasingly critical optimization in high performance design. The problem of finding a delay-optimal buffered Steiner tree has been an active area of research, and excellent solutions exist for most instances. However, current approaches fail to adequately solve a particular class of real-world "difficult" instances which are characterized by a large number of sinks, variations in sink criticalities, and varying polarity requirements. We propose a new Steiner tree construction called C-Tree for these instance types. When combined with van Ginneken style buffer insertion, C-Tree achieves higher quality solutions with fewer resources compared to traditional approaches.

1. Introduction

Interconnect's domination of system performance has made buffer insertion a critical step in modern VLSI design methodologies. The number of buffers needed to achieve timing closure continues to rise with decreasing feature size.

Several works have studied the problem of inserting buffers to reduce the delay on signal nets. van Ginneken's dynamic programming algorithm [13] has become a classic in the field. Given a fixed routing topology, his algorithm finds the optimal buffer placement on the topology under the Elmore delay model for a single buffer type and simple gate delay model. Together the enhancements to this work (e.g., [1][2][10][11][12]) make the van Ginneken style of buffer insertion quite potent as it can handle many constraints, buffer types, and delay models, while retaining optimality under many of these conditions.

The primary shortcoming with this approach is that the buffers must be inserted on the given Steiner topology. Thus, both Okamoto and Cong [12] and Lillis *et al.* [11] have combined buffer insertion with Steiner tree constructions, the former with A-Tree [6] and the latter with P-Tree [9].

This simultaneous approach is in some sense equivalent to the two-step approach of (1) constructing a Steiner tree, and (2) running van Ginneken style buffer insertion. An optimal solution can always be realized using the two-step approach if one uses the "right" Steiner tree (i.e., the tree resulting from ripping buffers out of the optimal solution) since the buffer insertion step is optimal. Of course, finding the "right" tree is difficult since the true objective cannot be directly optimized during the Steiner construction. However, if one tries to construct a "buffer-aware" Steiner tree, i.e., a tree with topology that anticipates good potential buffer locations, we believe the two-step approach can be as effective as the simultaneous approach.

For most nets, finding the right Steiner tree is easy (assuming no resource constraints). For two-pin nets a direct connection is optimal, and there are a manageable number of topologies for five sinks or less. This work focuses on the most difficult nets for which finding the appropriate Steiner topology is not at all obvious. These nets typically have more than 15 sinks, varying degrees of sink criticalities, and differing sink polarity constraints. Finding effective solutions for these nets is critical; a high-fanout net is more likely to be in a critical path because it is inherently slow.

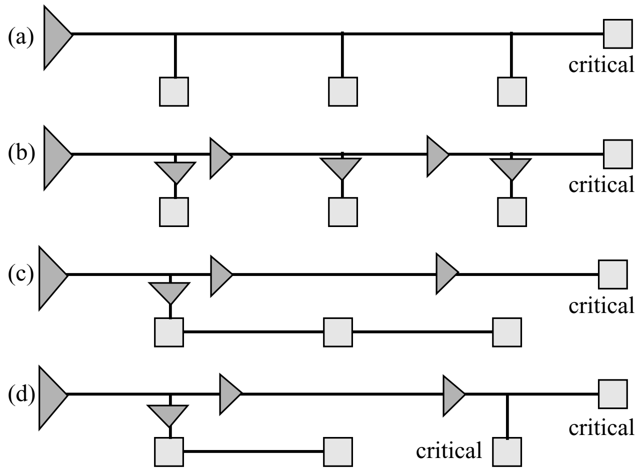

Figure 1 Example showing the minimum unbuffered delay tree (a) leads to a buffered tree (b) that is inferior to the best buffered tree (c) (since it needs fewer buffers). If two sinks are critical then a different optimal topology (d) would result.

Of course, a good heuristic for finding the right Steiner tree must take into account potential buffering. Figure 1(a) shows a 4-sink example where only one sink is critical. The (a) unbuffered tree has minimum wire length, yet (b) inserting buffers requires three buffers to decouple the three non-critical sinks; however, for a different topology (c) the buffered tree requires just one decoupling buffer. Thus, the tree in (c) uses fewer resources, and may also achieve a lower delay to the critical sink since the driver in (c) drives a smaller capacitive load than in (b).

One approach to finding this topology is to cluster non-critical sinks together and route each cluster individually. If there are multiple critical sinks (d), then a different topology which clusters critical sinks together will yield the best solution. One could find this tree by clustering sinks into a critical and non-critical cluster. The Steiner algorithm must be aware of opportunities to adjust the topology to allow for potential off-loading of non-critical sinks.

Finding the "right" Steiner tree becomes more difficult if one considers polarity constraints. During synthesis, fanout trees are built to repower and distribute a signal and/or its complement to a set of sinks without knowledge of the layout of the net. Once the net is placed, the tree may be grossly suboptimal. At this stage, the buffers and inverters can be ripped out of the tree and re-inserted while utilizing the new layout information. However, removing the buffers and inverters may leave sinks with opposing polarities.

Figure 2 shows a net with five sinks with positive ('+') polarity and five with negative ('-') polarity. The tree in (a) requires at least five inverters to satisfy polarity constraints, while the tree in (b) requires just one. The latter solution can be found by independently routing two clusters, one with all positive sinks and one with all negative sinks. Existing timing-driven Steiner tree constructions (e.g., [3][4][9]) cannot find this topology.

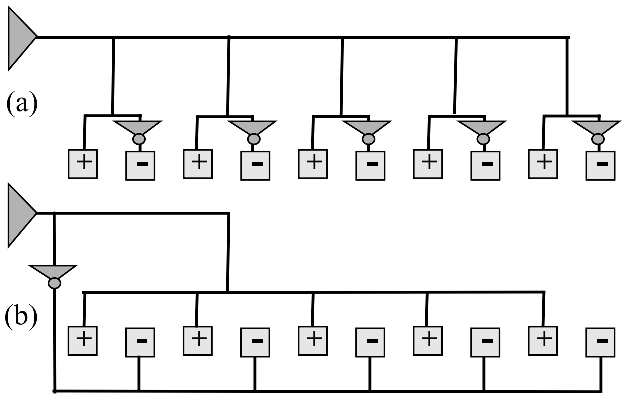

Figure 2 Example of how polarity constraints affect topology. The tree in (a) requires at least five inverters to satisfy polarity constraints while the tree in (b) requires just one.

The purpose of this work is to find a Steiner tree algorithm for particularly difficult instances which can be used in conjunction with van Ginneken style buffer insertion. Our proposed C-Tree heuristic first clusters sinks based on spatial, temporal, and polarity locality. Next, a sub-tree is then formed within each cluster, and finally, the trees are connected using a timing-driven Steiner tree at the top level. We show that this two-level approach is not only more efficient than the existing state-of-the art, but also generates higher quality solutions while using fewer buffers.

2. Preliminaries

We are given a net $N = \{s_0, s_1, \ldots, s_n\}$ with $n+1$ pins, where s_0 is the unique *source* and $s_1, \ldots, s_n$ are the sinks. Let $x(s)$ and $y(s)$ denote the 2-dimensional coordinates of s, and let $RAT(s)$, $cap(s)$, and $pol(s)$ denote the required arrival time, input capacitance, and polarity constraint for sink s. We assign $pol(s) = 1$ for a sink which requires the inversion of the signal from s_0 to s, and $pol(s) = 0$ for a "normal" sink that prohibits the inversion of the signal. A rectilinear Steiner tree $T(V, E)$ has a set of nodes $V = N \cup I$ where I is the set of intermediate 2-dimensional Steiner points and a set of horizontal and vertical edges E. Wire resistance and capacitance parasitics are given to permit interconnect delay calculation for a particular geometric topology.

For a given tree $T(V, E)$, a *buffered tree* $T_B(V_B, E_B)$ can be constructed from T if (i) there exists a set of nodes V' (corresponding to buffers) such that $V_B = V \cup V'$, (ii) each edge in E_B is either in E or is contained[1] within some edge in E and (iii) T_B is a rectilinear Steiner tree. Note that V and V' are not necessarily disjoint. Hence, buffers in T_B can only be inserted on the edges in T. Running van Ginneken style buffer insertion on T guarantees a buffered tree T_B. Let $nb(T_B)$ be the number of buffers inserted in T_B, i.e., $|V'|$.

Each Steiner tree (with or without buffers) has a unique path from s_0 to sink s_i. For each $v \in V'$, let $b(v)$ denote the particular buffer type (size, inverting, etc.), chosen from a buffer library B,inserted at v. Let $D(s_0, s_i, T_B)$ be the delay from s_0 to s_i within T_B. The delay can be computed using many ways; for this discussion, we adopt the Elmore model for wires and a switch-level linear model for gates. Our formulation is by no means restricted to these models (see e.g., [2]). The *slack* for a tree T_B is given by $slack(T_B) = min\{RAT(s_i) - D(s_0, s_i, T_B) \,\|\, 1 \le i \le n\}$.

The traditional buffer tree objective function is to maximize $slack(T_B)$. This can waste resources since additional buffers may be used to garner only a few extra picoseconds of performance. An alternative is to find the fewest buffers such that $slack(T_B) \ge 0$. However, a zero slack solution may not be achievable in this formulation, yet the designer still wishes to reduce the slack, even if a positive slack is not achievable. Instead of a single objective, one can generate a solution set that trades off maximizing the worst slack with the number of inserted buffers. This can be achieved with a van Ginneken style algorithm [10] or via simultaneous optimization [11]. Our problem statement is as follows:

Buffered Steiner Tree Problem: Given timing and polarity constraints and the topology for net N, a buffer library B, and the technology's interconnect parasitics, find a single Steiner tree T over N so that the family F of buffered trees constructed from T by applying van Ginneken style buffer insertion using B satisfies polarity constraints and is *dominant*. A family F is dominant if for every buffered tree T_B', there exists a tree T_B in F such that $slack(T_B) \ge slack(T_B')$ and $nb(T_B) \ge nb(T_B')$.

The formulation does not restrict the algorithm to a particular buffer resource or timing constraint, but rather allows the designer to choose a solution within the family

[1] Edge $((x_1, y_1)(x_2, y_2))$ is said to be *contained* within edge $((x_3, y_3), (x_4, y_4))$ if $min(x_3, x_4) \le x_1, x_2 \le max(x_3, x_4)$ and $min(y_3, y_4) \le y_1, y_2 \le max(y_3, y_4)$.

that is most appropriate for the particular design. Although not explicitly stated, there is actually a wire length component that can also be traded off. For example, if the routing resources are more tightly constrained than the area resources, one might want to reduce wire length for the price of additional buffers, while maintaining the same timing characteristics. To handle this constraint, one could used a cost function that combine the costs of buffering and wire resources which would allow simultaneous wire sizing within the buffer insertion optimization.

3. The C-Tree Algorithm

3.1 Overview

Our Steiner construction is called C-Tree, for "Clustered tree", emphasizing the clustering step, as opposed to the underlying timing-driven tree heuristic. The fundamental idea behind C-Tree is to construct the tree in two levels. C-Tree first clusters sinks with similar characteristics (criticality, polarity and distance). This step potentially isolates positive sinks from negative ones and non-critical sinks from critical ones. The algorithm then constructs low-level Steiner trees over each of these clusters. Finally, a top-level timing-driven Steiner tree over the set of clusters is computed. This tree is then merged with the low-level trees to yield a solution for the entire net.

```
Input:   $N = \{s_0, s_1, \ldots, s_n\}$ ≡Net to be routed
         $k$ ≡ Number of clusters
Output: $T$ ≡ Routing tree over $N$

1. $\{N_1, N_2, \ldots, N_k\} = Clustering(N - s_0)$. Set $N_0 = \{s_0\}$.
2. for $i = 1$ to $k$ do
3.     Find a tapping point $tp_i$ for cluster $N_i$.
4.     Add $tp_i$ to $N_i$ and label $tp_i$ as the source.
5.     Let $T_i = TimingDrivenSteiner(N_i)$.
6.     Set $RAT(tp_i) = slack(T_i)$, $cap(tp_i) = cap(T_i)$,
       and add $tp_i$ to $N_0$.
7. Compute $T_0 = TimingDrivenSteiner(N_0)$.
8. Combine all edges and nodes of $T_0, T_1, \ldots, T_k$ into tree $T$.
```

Figure 3 C-Tree Steiner Tree Algorithm (N, k).

Figure 3 presents C-Tree pseudocode. We assume that two generic subroutines, Clustering and TimingDrivenSteiner, are given (see Sections 3.2-3.4). However, one could implement these subroutines in a variety of ways to achieve similar clustering and routing functionality.

Step 1 invokes Clustering, which takes the sinks of a net as input and outputs a set of clusters $\{N_1, N_2, \ldots, N_k\}$. The net corresponding to the top-level tree N_0 is also initialized to contain the source. Step 2 iterates over the clusters, and in Step 3, a *tapping point* tp_i is computed for cluster N_i. The tapping point represents the source for the tree T_i over N_i and also the point where the top-level tree T_0 will connect to T_i. We choose tp_i to be a point on the bounding box of N_i closest to s_0. Step 4 then assigns tp_i to be the source for N_i. Step 5 invokes TimingDrivenSteiner on N_i to yield a tree T_i. Step 6 then propagates the RAT up T_i to yield an RAT constraint for tp_i. The capacitance of tp_i is assigned to be that of T_i After completing these operations for all the tapping points, N_0 consists of s_0 plus k tapping points which now serve as sinks. Step 7 computes the top-level Steiner tree for this instance, and Step 8 merges all the Steiner trees into a single solution.

Figure 4 shows an example. In (a), a clustering of the sinks is performed. In (b), the three tapping points are shown as black circles, and Steiner trees are computed for each cluster. Next (c), the top-level Steiner tree connecting the source to the tapping points is found, and finally, (d) the tapping points are removed and the Steiner trees are merged into a single tree. A clear advantage of this approach is that van Ginneken style buffer insertion can insert buffers to either drive, decouple, or reverse polarity of any particular cluster. Of course, C-Tree is sensitive to the actual clustering algorithm used, which we now describe.

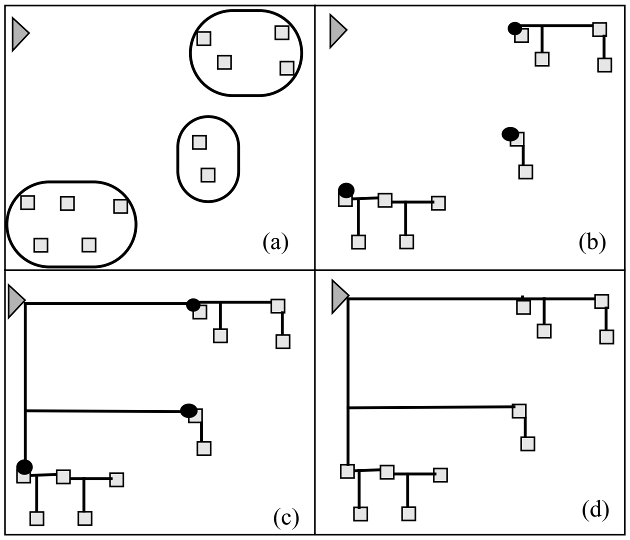

Figure 4 Example execution of the C-Tree algorithm.

3.2 Clustering Distance Metric

The key to clustering any data set is devising a dissimilarity or distance metric between pairs of points. The points in our instances have three properties: *spatial* (coordinates in the plane), *temporal* (required arrival times), and *polarity*. Our distance metric incorporates all of these elements; we first define individual spatial, temporal and polarity metrics, then combine them using scaling into a single distance metric.

The correct spatial and polarity metrics are straightforward. The spatial (Manhattan) distance $sDist(s_i, s_j)$ and polarity distance $pDist(s_i, s_j)$ for sinks s_i and s_j are given by $|x(s_i) - x(s_j)| + |y(s_i) - y(s_j)|$ and $|pol(s_i) - pol(s_j)|$, respectively. $pDist(s_i, s_j)$ is zero when the polarities for s_i and s_j are the same and one when they are opposing.

Finding a good temporal metric is trickier. First, RAT is not the only indicator of sink criticality. If s_i and s_j have the same RAT yet s_i is further from s_0 than s_j, then s_i is more critical since it is harder to achieve the same RAT over the longer distance. An estimate of the achievable delay to s_i can be used to adjust the RAT. Assuming an optimally buffered direct connection from s_0 to s_i, with sub-trees decoupled by buffers with negligible input capacitance, then the achievable delay is equivalent to the formula for optimal buffer insertion on a two-pin net. Let the *achievable delay* be denoted by $AD(s_i)$ using the formula from [1]. Let $AS(s_i) = RAT(s_i) - AD(s_i)$ be the potentially *achievable slack* for s_i. Now $AS(s_i)$ gives a better indicator of the criticality of s_i than $RAT(s_i)$.

Yet, $|AS(s_i) - AS(s_j)|$ is still not a good temporal metric. Assume that the achievable slacks for three sinks are $AS(s_1) = -1ns$, $AS(s_2) = 2ns$, and $AS(s_3) = 10ns$. Sink s_1 is most critical while s_2 and s_3 are both non-critical. Intuitively, s_2 is more similar to s_3 than to s_1, despite the 8 ns difference, because both s_2 and s_3 have high positive achievable slack. A temporal metric that looks at the differences in AS values cannot capture this behavior.

The criticality of s_i is given by $crit(s_i)$, where the more critical sink has $crit(s_i) = 1$ and $crit(s_i) \rightarrow 0$ as $AS(s_i) \rightarrow \infty$, i.e., the criticality of a sink is one if it is most critical and zero if it is totally uncritical; otherwise it lies somewhere in between. We define criticality as follows:

$$crit(s_i) = e^{\alpha(mAS - AS(s_i))/(aAS - mAS)} \text{ where}$$

$$mAS = min_{1 \le i \le n} AS(s_i)\,,\; aAS = \sum_{1 \le i \le n} AS(s_i)/n \quad (1)$$

Here mAS and aAS are the minimum and average AS values over all sinks, and $\alpha > 0$ is a user parameter.[2] Observe that $crit(s_i)$ is indeed one when $AS(s_i) = mAS$ and zero as $AS(s_i)$ goes to infinity. For a sink s_i with average achievable slack ($AS(s_i) = aAS$), then $crit(s_i)$ equals $e^{-\alpha} \cong 0.135$ when $\alpha = 2$. An average sink has criticality much closer to that of a sink with infinite AS as opposed to minimum AS. Now, temporal distance $tDist(s_i, s_j)$ can be defined as the difference in sink criticalities, or $|crit(s_i) - crit(s_j)|$.

Both temporal and polarity distances are on a zero to one scale, so spatial distance must be scaled before combining terms. Let $sDiam(N) = max\{sDist(s_i, s_j) \| 1 \le i, j \le n\}$ be the spatial diameter of sinks. The scaled distance between two sinks can be expressed as dividing $sDist$ by $sDiam(N)$. Our distance metric $dist(s_i, s_j)$ is a linear combination of the spatial, temporal, and polarity distances:

$$\beta\frac{sDist(s_i, s_j)}{sDiam(N)} + (1-\beta)tDist(s_i, s_j) + pDist(s_i, s_j)\,. \quad (2)$$

The parameter β lies between zero and one and trades off between spatial and temporal distance (we use $\beta = 0.65$). Note that the distance between two sinks with the same polarity is no more than the distance between two sinks with opposite polarity which ensures that polarity has precedence over spatial and temporal distance. This is key to avoiding the behavior shown in Figure 2(a).

[2] If all achievable slacks are exactly equal, i.e., $aAS = mAS$, then we define $crit(s_i) = 1$ for all sinks s_i.

3.3 Clustering

For clustering sinks, we adopt the K-Center heuristic [7] which seeks to minimize the maximum radius (distance to the cluster center) over all clusters. K-Center is just one of several potential clustering methods that could be used to achieve the purpose of grouping sinks with common characteristics. K-Center iteratively identifies points that are furthest away, which are called cluster seeds. The remaining points are clustered to their closest seed. For geometric instances, K-Center guarantees that the maximum diameter of any cluster is within a factor of two of the optimal solution [7]. The time complexity of K-Center is $O(nk)$.

3.4 Timing-Driven Steiner Tree Construction

The timing-driven Steiner tree method is implemented via the Prim-Dijkstra algorithm [3] which trades off between Prim's minimum spanning tree algorithm and Dijkstra's shortest path tree algorithm via a parameter c which lies between 0 and 1. Since Prim's algorithm yields minimum wire length (for a spanning tree) and Dijkstra's yields a tree with minimum radius, the trade-off is able to capture the desirable properties behind both extremes.

In our experiments, we run the Prim-Dijkstra algorithm for $c = 0.0, 0.25, 0.5, 0.75, 1.0$ followed by a post-processing algorithm that remove overlapping edges and generates a Steiner tree. Of the five constructions, the tree T which minimizes the slack at the tapping point is selected.

Certainly, other choices are just as reasonable. In fact, we speculate that P-Tree [11] would probably improve results slightly. We chose the Prim-Dijkstra algorithm because it is simple to implement, efficient and scalable, and because it outperformed the critical sink construction of [4] in separate experiments. P-Tree, while likely superior in terms of quality, is not as efficient, scalable, and easy to implement.

4. Experimental Results

We identified 8 nets on various industrial designs that the current production-level buffer insertion methodology had difficulty optimizing. The polarity characteristics and timing constraints for the nets are summarized in Table 1.

We compare C-Tree to the P-Tree [9] and Prim-Dijkstra [3] timing-driven tree constructions and also to BP-Tree (simultaneous buffering and routing) [11]. P-Tree was shown to yield better timing results than either SERT [4] or A-Tree [6]. P-Tree actually consists of two algorithms: P-TreeA seeks to minimize area, while P-TreeAT generates a family of solutions that trade off between area and timing. The Prim-Dijkstra algorithm is actually equivalent to "flat" C-Tree with each sink in its own cluster. For each tree, we ran van Ginneken style buffer insertion using a library of five non-inverting and two inverting buffers to generate a family of solutions. Like P-Tree, BP-Tree also has two modes which we suffix with either N (normal) or F (fast).

The results are summarized in Table 2. Comparisons for each net are shown in several rows. The first two rows contain results for P-TreeAT and P-TreeA, except for the three largest nets for which P-TreeAT ran out of memory (on a machine with 2Gb of RAM). The next row is for BP-TreeN except for the largest test case, for which BP-TreeF is reported since BP-TreeN ran out of memory. In general, BP-TreeF inserted about 3 times as many buffers as BP-TreeN. The next row is "flat" C-Tree or the Prim-Dijkstra algorithm. The remaining rows for each net are presented for C-Tree for a decreasing number of clusters to show the trade-off. For each algorithm, we present the following data:

Net Name	Sinks			RAT	
	+	-	Total	min	max
n873	10	10	20	730	6656
poi3	10	10	20	52	6707
n189	15	14	29	610	6650
n786	18	14	32	97	6704
n870	24	19	43	739	6589
big1	40	48	88	1974	159565
big2	38	41	79	104	65838
big3	34	29	63	1097	40675

Table 1 Polarity and temporal characteristics of the 8 nets.

- slack (to the most critical sink) in picoseconds (ps) and wire length of the tree before buffer insertion,
- the slack (ps) and the number of buffers used for three of the family of solutions generated. The Min Opt solution has the minimal buffering needed to fix polarity constraints. The Full Opt solution has the maximum slack, regardless of the number of buffers used, and Mid Opt reflects a solution in between. The three solutions give a reasonable view of the trade-off curve generated.
- the slack (ps) and wire length after a post-processing step on the Full Opt buffered solution. Once buffers are inserted, some wire length may be eliminated via simple re-routing. This step tries to reduce wire length without increasing slack from the Full Opt buffered tree.
- the total CPU time (s) for the entire process (tree construction, buffer insertion, and post-processing). Runtimes are for a Sun Sparc Ultra-60 with 2Gb of RAM.

We make several observations.

- For the Full Opt solution, C-Tree was able to find solutions with slacks at least as high as all the other approaches for at least one clustering (except for n873 for which C-Tree's slack was inferior by one ps). Sometimes the C-Tree slacks were significantly better (e.g., n870, and big1); most of the time the Full Opt slacks were fairly indistinguishable among the algorithms.
- The fewer clusters used by C-Tree, the fewer the number of buffers are needed to fix polarity constraints. With two clusters, one buffer is always sufficient. However, fewer clusters yields additional wire length. Indeed, two clusters yields almost double the wire length since two low-level trees are being routed over the same geometric space, one to the positive and one to the negative polarity sinks. When the number of clusters is small, the wire length does increase significantly.
- The post-processing step did not affect slack much at all, but occasionally reduced wire length (e.g., for big3).
- P-TreeAT and BP-TreeN are the most inefficient algorithms. P-TreeA is slightly more inefficient than the Prim-Dijkstra approach, but C-Tree is actually the fastest of the three constructions.
- For the larger nets, the other approaches require many more buffers than C-Tree to find a feasible solution. For example, P-Tree required 32, 27 and 27 buffers to satisfy constraints for big1, big2, and big3, respectively. C-Tree could generally find a solution with slack at least as high as P-Tree with 4, 6, and 9 buffers, respectively.
- There was not much differentiation in slack among the algorithms. The tree capacitances are mostly wire dominated causing most buffers to be used to improve delay instead of decoupling large loads. Nets with some very high capacitance sinks may prove more "difficult". Also, in several cases the highest slack solution was actually very close to the minimum sink RAT, which is an upper bound on slack at the source. For difficult instances, one may wish to alter the objective to capture the benefits of improving timing to the less critical sinks.

5. Conclusion

We identified a class of buffered Steiner tree instances for which existing algorithms are inadequate. These instances have several sinks and varying temporal and polarity constraints. We proposed the C-Tree algorithm which utilizes a distance metric that combines spatial, temporal, and polarity characteristics. Experiments show that C-Tree obtains results with slack equal to or better than previous approaches while using fewer buffers. C-Tree can also trade-off between buffering and wiring resources via the number of clusters. We hope that this work opens the door for additional research on these types of difficult instances.

References

[1] C. J. Alpert and A. Devgan, "Wire Segmenting for Improved Buffer Insertion", *IEEE/ACM DAC*, 1997, pp. 588-593.

[2] C. J. Alpert, A. Devgan, and S. T. Quay, "Buffer Insertion with Accurate Gate and Interconnect Delay Computation", *IEEE/ACM Design Automation Conf.*, 1999, pp. 479-484.

[3] C. J. Alpert, T. C. Hu, J. H. Huang, A. B. Kahng, D. Karger, "Prim-Dijkstra Tradeoffs for Improved Performance-Driven Routing Tree Design," *IEEE TCAD*, 14(7), 1995, 890-896.

[4] K.D. Boese, A. B. Kahng, B. A. McCoy, G. Robins, "Near-optimal Critical Sink Routing Tree Constructions", *IEEE Trans. on CAD*, 14(12), Dec. 1995, pp. 1417-1436.

[5] C. C. N. Chu and D. F. Wong, "Closed Form Solution to Simultaneous Buffer Insertion/Sizing and Wire Sizing", *International Symposium on Physical Design*, 1997, pp. 192-197.

[6] J. Cong, K. S. Leung, and D. Zhou, "Performance-Driven Interconnect Design Based on Distributed RC Delay Mode," *IEEE/ACM Design Automation Conf.*, 1993, pp. 606-611.

[7] T. F. Gonzalez, "Clustering to Minimize the Maximum Intercluster Distance", *Theoretical Comp. Sci.*, 38, 293-306, 1985.

[8] M. Lai and D. F. Wong, "Maze Routing with Buffer Insertion and Wiresizing", *IEEE/ACM DAC.*, 2000, pp. 374-378.

[9] 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", *33th IEEE/ACM DAC,* 1996, pp. 395-400.

[10] J. Lillis, C.-K. Cheng and T.-T. Y. Lin, "Optimal Wire Sizing and Buffer Insertion for Low Power and a Generalized Delay Model", *IEEE J. Solid-State Circuits*, 31(3), 1996, 437-447.

[11] J. Lillis, C.-K. Cheng and T.-T. Y. Lin, “Simultaneous Routing and Buffer Insertion for High Performance Interconnect”, *Sixth Great Lakes Symposium on VLSI*, 1996, pp. 148-153.

[12] T. Okamoto and J. Cong, “Buffered Steiner Tree Construction with Wire Sizing for Interconnect Layout Optimization”, *IEEE/ACM Int. Conf.Computer-Aided Design*, 1996, 44-49.

[13] L. P. P. P. van Ginneken, “Buffer Placement in Distributed RC-tree Networks for Minimal Elmore Delay”, *Intl. Symposium on Circuits and Systems*, 1990, pp. 865-868.

Net Name	Algorithm	# Clusts	Before Opt		Min Opt		Mid Opt		Full Opt		Post Process		CPU (s)
			slack (ps)	wire	bufs	slack (ps)	bufs	slack (ps)	bufs	slack (ps)	slack (ps)	wire	
n873	P-TreeAT	1	-788	4358	7	213	9	494	11	547	547	4293	2.6
	P-TreeA	1	-780	4321	7	204	9	494	11	547	547	4272	0.4
	BP-TreeN	1	----	----	7	151	9	541	10	566	----	----	62.1
	C-Tree	20	-769	4272	7	201	9	488	12	536	536	4272	0.2
	C-Tree	11	-822	4512	6	194	8	491	11	537	537	4301	0.3
	C-Tree	5	-993	5328	2	-92	5	520	9	528	539	5180	0.3
	C-Tree	2	-1036	5703	1	-17	4	529	7	546	546	5703	0.4
poi3	P-TreeAT	1	-727	6010	10	-418	12	38	13	40	40	6008	2.0
	P-TreeA	1	-727	6008	10	-418	12	36	13	38	38	6008	1.1
	BP-TreeN	1	----	----	7	-441	9	38	10	40	----	----	65.1
	C-Tree	20	-713	5852	8	36	9	43	9	43	43	6030	0.7
	C-Tree	11	-775	6550	5	36	6	43	6	43	43	6248	0.8
	C-Tree	4	-860	7501	2	18	3	25	4	31	31	6087	1.2
	C-Tree	2	-1155	10823	1	-544	3	16	5	26	26	10823	1.0
n189	P-TreeAT	1	-1235	4963	10	217	12	514	14	560	560	4953	33.8
	P-TreeA	1	-1229	4935	11	112	15	486	25	493	494	5033	2.3
	BP-TreeN	1	----	----	8	-98	10	419	12	472	----	----	511.4
	C-Tree	29	-1230	4937	9	200	12	491	15	510	510	4937	0.5
	C-Tree	16	-1271	5134	8	166	10	468	12	533	533	5112	0.5
	C-Tree	10	-1519	6314	5	-277	8	538	10	548	548	5576	0.6
	C-Tree	2	-1824	7772	1	-880	3	531	6	574	578	7582	0.6
n786	P-TreeAT	1	-816	4958	9	-496	11	56	13	82	83	4896	118.4
	P-TreeA	1	-807	4859	11	-494	13	58	15	82	82	4859	3.2
	BP-TreeN	1	----	----	9	-422	11	79	13	84	----	----	748.1
	C-Tree	32	-807	4859	13	-501	16	50	19	67	67	4859	0.9
	C-Tree	15	-847	5308	6	-505	8	51	10	82	82	4971	0.8
	C-Tree	7	-884	5718	3	-505	5	67	7	82	82	5294	0.7
	C-Tree	2	-1199	9252	1	-619	4	61	6	70	70	9255	1.3
n870	P-TreeAT	1	-2587	4136	18	8	19	84	19	84	122	4119	193.3
	P-TreeA	1	-2567	4089	17	49	18	98	19	99	99	4089	4.1
	BP-TreeN	1	----	----	13	97	17	288	21	295	----	----	860.5
	C-Tree	43	-2677	4061	18	-186	22	-104	26	-101	-101	4061	1.4
	C-Tree	17	-2677	4347	7	133	11	245	15	254	254	4297	1.3
	C-Tree	9	-2727	4464	6	132	8	241	11	258	258	4386	0.9
	C-Tree	2	-3749	7688	1	-1965	5	348	9	355	355	7688	1.5
big1	P-TreeA	1	-932	14734	32	830	40	1083	48	1106	1228	16368	14.9
	BP-TreeF	1	----	----	99	1381	98	1479	97	1555	----	----	308.5
	C-Tree	88	-162	15798	33	1267	35	1412	37	1416	1416	15798	5.3
	C-Tree	30	-844	23866	19	1090	21	1570	23	1595	1595	22230	7.0
	C-Tree	12	-1358	30021	6	236	9	1659	12	1682	1682	25550	3.7
	C-Tree	2	-982	25985	1	10	4	1660	7	1690	1692	25811	8.7
big2	P-TreeA	1	-1263	8899	27	-461	32	-71	38	-44	-44	8899	4.0
	BP-TreeN	1	----	----	20	-201	25	-29	29	-12	----	----	494.6
	C-Tree	79	-1258	9018	26	-303	29	-257	31	-255	-142	9226	3.7
	C-Tree	49	-1398	10672	21	-442	25	-129	29	-114	-112	9872	2.8
	C-Tree	28	-1682	13995	15	-704	22	-74	29	-68	-68	12340	3.2
	C-Tree	2	-1614	13199	1	-1118	7	-62	12	-51	-51	13199	3.1
big3	P-TreeA	1	-23	6907	27	867	31	1012	34	1021	1022	6907	1.9
	BP-TreeN	1	----	----	19	570	22	1048	25	1055	----	----	199.6
	C-Tree	63	0	6966	23	631	26	1024	28	1027	1027	6966	1.8
	C-Tree	38	-91	7987	18	871	21	979	23	981	988	7437	1.6
	C-Tree	21	-282	10300	11	652	14	1013	17	1021	1022	9422	1.5
	C-Tree	2	-264	9965	1	278	5	992	9	1028	1028	9962	0.9

Table 2 Summary of experimental results.