Buffered Steiner Trees for Difficult Instances

Charles J. Alpert, *Member, IEEE*, Gopal Gandham, Milos Hrkic, Jiang Hu, Andrew B. Kahng, John Lillis, Bao Liu, Stephen T. Quay, Sachin S. Sapatnekar, and A. J. Sullivan

***Abstract*—With the rapid scaling of integrated-circuit technology, buffer insertion has become an increasingly critical optimization technique in high-performance design. The problem of finding a buffered Steiner tree with optimal delay characteristics has been an active area of research and excellent solutions exist for most instances. However, there exists a class of real "difficult" instances, which are characterized by a large number of sinks (e.g., 20–100), large variations in sink criticalities, nonuniform sink distribution, and varying polarity requirements. Existing techniques are either inefficient, wasteful of buffering resources, or unable to find a high-quality solution. We propose C-tree, a two-level construction that first clusters sinks with common characteristics together, constructs low-level Steiner trees for each cluster, then performs a timing-driven Steiner construction on the top-level clustering. We show that this hierarchical approach can achieve higher quality solutions with fewer resources compared to traditional timing-driven Steiner trees.**

***Index Terms*—Buffer insertion, global routing, interconnect synthesis, Steiner tree.**

I. Introduction

IT IS NOW widely accepted that interconnect is becoming increasingly dominant over transistor and logic performance in the deep-submicromter regime. Buffer insertion is now a fundamental technology used in modern very large scale integration design methodologies (see [10] for a survey). Cong [9] illustrates that as gate delays decrease with increasing chip dimensions, the number of buffers required quickly rises. He expects that close to 800 000 buffers will be required for 50-nm technologies. It is critical to automate the entire interconnect optimization process to efficiently achieve timing closure.

Several works have studied the problem of inserting buffers to reduce the delay on signal nets. Closed-form solutions for two-pin nets have been proposed in [1], [6], [8], and [13]. van Ginneken's dynamic-programming algorithm [22] has become a classic in the field. Given a fixed Steiner-tree 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. Several extensions to this work have been proposed (e.g., [2], [3], [18], [20], and [21]). Together, these enhancements make the van Ginneken buffer-insertion framework very powerful as it can incorporate slew, noise, and capacitance constraints, a range of buffer and inverter types, and higher order gate and interconnect delay models, while retaining optimality under many of these variations. Most recently, research on buffer insertion has focused on accommodating various types of blockage constraints [12], [16], [17].

Clearly, the primary shortcoming with the van Ginneken-style of buffer insertion is that it is limited by the given Steiner topology. Thus, both Okamoto and Cong [21] and Lillis *et al.* [20] have combined buffer insertion with a Steiner-tree constructions, the former with A-tree [11] and the latter with P-tree [18]. Later, in [12], the work of [21] was extended to handle fixed buffer locations and wiring blockages.

Observe that the simultaneous approach is not necessarily any better than the two-step approach of first constructing a Steiner tree, then 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 buffer-insertion objective cannot be directly optimized. We believe that if one tries to construct a "buffer-aware" Steiner tree, i.e., a tree with topology that anticipates good potential buffer locations, the two-step approach can be as effective (and potentially more efficient) than the simultaneous approach.[1]

For the majority of the nets in a design, finding the right Steiner tree is easy (assuming no blockages or buffer resource constraints). For two-pin nets, a direct connection is optimal and there are a small number of possible topologies for five sinks or less. The purpose of our paper is to focus on the most difficult nets for which finding the appropriate Steiner topology is not at all obvious. These nets will typically have more than 15 sinks, varying degrees of sink criticalities, and differing sink polarity constraints. Optimizing these nets effectively is often critical, as large high-fan-out nets are more likely to be in a critical path because they are inherently slow.

Of course, a good heuristic for finding the right Steiner tree must take into account potential buffering. Consider the

Manuscript received April 12, 2001; revised July 13, 2001. This work was supported in part by a National Science Foundation CAREER Award and in part by the MARCO Gigascale Silicon Research Center. This paper was recommended by Guest Editor M. D. F. Wong.

C. J. Alpert, J. Hu, and S. T. Quay are with the IBM Corporation, Austin, TX 78758 USA.

G. Gandham and A. J. Sullivan are with the IBM Corporation, Hopewell Junction, NY 12533 USA.

M. Hrkic and J. Lillis are with the Electrical Engineering and Computer Science Department, University of Illinois, Chicago, IL 60607 USA.

A. B. Kahng and B. Liu are with the Department of Computer Science and Engineering, University of California at San Diego, La Jolla, CA 92093 USA.

S. S. Sapatnekar is with the Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN 55455 USA.

Publisher Item Identifier S 0278-0070(02)00092-1.

[1]None of the existing simultaneous tree and buffering approaches can handle the types of constraints that a van Ginneken-style framework can (such as slew constraints and higher order delay modeling). One could use the simultaneous approach (with its simpler assumptions and modeling) first to uncover the routing tree topology and then pass this result, with the buffers deleted, to the more sophisticated buffer-insertion algorithm that uses a fixed routing topology.

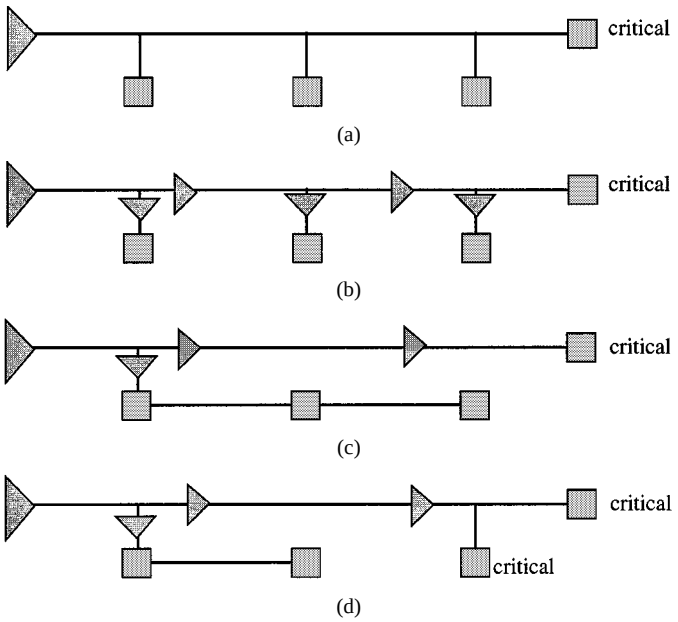


Fig. 1. Example where (a) the tree with less wire length yields (b) an inferior buffered tree than (c) the tree with more wire length. Tree in (b) requires three buffers to decouple the load, while the tree in (c) requires just one. If instead, two sinks are critical, then (d) the best buffered topology would group these critical sinks into the same subtree.

four-sink example in Fig. 1(a), where only one of the sinks is critical. The unbuffered tree in Fig. 1(a) has minimum wire length, yet inserting buffers in Fig. 1(b) would require three buffers to decouple the three noncritical sinks, while the buffered tree in Fig. 1(c) requires but one decoupling buffer. Thus, the tree in Fig. 1(c) uses fewer resources and further may actually result in a lower delay to the critical sink since the driver in Fig. 1(c) drives a smaller capacitive load than in Fig. 1(b). One can identify this topology by first clustering the noncritical sinks together and forcing the topology to route everything within a cluster as a separate subtree. If there are multiple critical sinks, as shown in Fig. 1(d), then a totally different topology which groups the critical sinks together in the same subtree likely yields the best solution. This tree would be identifiable if the critical sinks and noncritical sinks were clustered into two separate clusters and subtrees were constructed for each cluster. The Steiner algorithm must be aware of opportunities to manipulate the topology to allow potential offloading of noncritical sinks.

However, the crux of the problem with current buffer-tree technology is that it cannot adequately handle polarity constraints. During early synthesis, fan-out 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 is often grossly suboptimal. At this stage, one can rip out the fan-out tree and rebuild it using physical design information. However, ripping out the complete fan-out tree of buffers and inverters may leave sinks with opposing polarity requirements.

Fig. 2 shows a net with five sinks with normal polarity (indicated by a plus) and five with negative polarity (indicated by a minus). The tree in Fig. 2(a) requires a minimum of five inverters simply to ensure that polarity constraints are satisfied, while the tree in Fig. 2(b) requires just one. This solution can be identified by clustering the positive and negative sinks into two disjoint clusters and creating separate subtrees for the sinks in each cluster. Notice that it is fairly easy to reduce the wire length in Fig. 2(b) while preserving the topology, which actually yields a self-overlapping tree. Existing timing-driven Steiner-tree constructions (e.g., [5], [10], and [18]) cannot find this topology. In general, forming one tree connecting negative sinks and one connecting positive sinks will minimize the number of buffers, but waste wire length. Ideally, one would like to find a tree construction that balances both wiring and buffering resources.

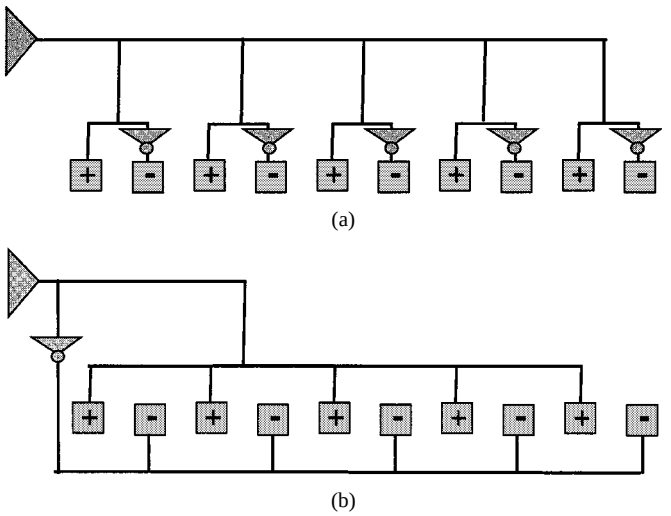


Fig. 2. Example of how polarity constraints affect topology. Tree in (a) requires at least five inverters to satisfy polarity constraints while (b) requires just one.

The purpose of this paper is to study Steiner-tree constructions for particularly difficult instances to optimize the buffered tree resulting from van Ginneken-style buffer insertion. We propose the clustered-tree (C-tree) heuristic that first clusters sinks based on spatial, temporal, and polarity locality. A subtree is then formed within each cluster and, finally, the trees are connected using a timing-driven Steiner 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.

The remainder of the paper is as follows. Section II presents notation and our problem formulation. Section III presents our proposed algorithm and Section IV presents experimental comparisons. We conclude in Section V.

II. Preliminaries

We are given a net $N = \{s_0, s_1, \ldots, s_n\}$ consisting of $n+1$ 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 two-dimensional (2-D) coordinates of pin s and let $\mathrm{RAT}(s)$ denote the required arrival time for a sink s. Each sink s has a capacitance $\mathrm{cap}(s)$ and a polarity constraint $\mathrm{pol}(s)$, where $\mathrm{pol}(s) = 0$ for a normal sink and $\mathrm{pol}(s) = 1$ for an inverted sink. The constraint $\mathrm{pol}(s) = 1$ requires the inversion of the signal from s_0 to s and $\mathrm{pol}(s) = 0$ 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-D Steiner points and a set of edges E such

that each edge in E is either horizontal or vertical. We also assume that wire resistance and capacitance parasitics are given to permit interconnect delay calculation for a particular geometric topology.

Given a Steiner tree $T(V, E)$, we say that a *buffered Steiner tree* $T_B(V_B, E_B)$ is constructed from T if: 1) there exists a set of nodes V' (corresponding to buffers) such that $V_B = V \cup V'$; 2) each edge in E_B is either in E or is contained[2] within some edge in E; and 3) T_B is a rectilinear Steiner tree. Consequently, one can obtain the original tree T by contracting T_B with respect to all nodes in V'/V. In other words, a buffered Steiner tree T_B, which can be constructed from T, must have the same wiring topology; buffers can only be inserted on the edges in T. Running a van Ginneken-style buffer-insertion algorithm on T is guaranteed to yield such a tree T_B. Let $\mathrm{cost}(T_B)$ be the cost of the wiring and buffering resources used by T_B. For example, $\mathrm{cost}(T_B)$ could be a linear combination of the total buffer area used in T_B and the wire length of T_B.

Each Steiner tree (with or without buffers) has a unique path from s_0 to a sink s_i. For each node $\nu \in V'$, let $b(\nu)$ denote the particular buffer type (size, inverting, etc.) chosen from a buffer library B that is located at ν. Let $\mathrm{Delay}(s_0, s_i, T)$ be the delay from s_0 to s_i within T. The delay can be computed using a variety of techniques. For the purposes of this discussion, we adopt the Elmore delay model [14] for wires and a switch-level linear model for gates. This formulation is by no means restricted to these models (see e.g., [3]). The slack for a tree T is given by $\mathrm{slack}(T) = \min\{\mathrm{RAT}(s_i) - \mathrm{Delay}(s_0, s_i, T) \parallel 1 \leq i \leq n\}$.

The obvious objective function for buffer insertion is to maximize $\mathrm{slack}(T_B)$ for a buffered tree T_B. This can clearly waste resources as several additional buffers may be used to garner only a few extra picoseconds of performance. Another alternative is to find the fewest buffers such that $\mathrm{slack}(T_B) \geq 0$. The problem with this formulation is often a zero slack solution is not achievable, yet it is still in the designer's interest to reduce the slack of critical nets, even if zero slack is not achievable. Instead of addressing either objective, one can generate a set of solutions that trade off maximizing the worst slack with the number of inserted buffers (or total buffer area). This can be done with a van Ginneken-style algorithm (such as [19]) or within a simultaneous optimization [20]. Thus, our problem statement is as follows.

Buffered Steiner-Tree Problem: Given a net N, a buffer library B and unit interconnect parasitics for the technology, find a single Steiner tree T over N so that the family F of buffered Steiner trees constructed from T by applying a van Ginneken-style algorithm using B satisfies polarity constraints and is *dominant*. We say a family F is dominant if for every buffered tree T'_B, there exists a tree T_B in F such that $\mathrm{slack}(T_B) \geq \mathrm{slack}(T'_B)$ and $\mathrm{cost}(T_B) \leq \mathrm{cost}(T'_B)$.

The problem is formulated in such a way that it might be possible that no optimal tree T exists because a dominant family may require multiple topologies. The purpose of this type of formulation is not to restrict the algorithm to a particular buffer resource or timing constraint, but rather to allow the designer (or a postprocessor) to find a solution within the family that is the most appropriate for the particular design.

[2]An edge connecting points (x_1, y_1) and (x_2, y_2) is *contained* within an edge connecting points (x_3, y_3) and (x_4, y_4), if $\min(x_3, x_4) \leq x_1, x_2 \leq \max(x_3, x_4)$ and $\min(y_3, y_4) \leq y_1, y_2 \leq \max(y_3, y_4)$.

C-Tree Steiner Algorithm (N, k)

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.

Fig. 3. High-level description of the C-tree framework.

III. C-Tree Algorithm

A. Overview

We call our Steiner construction C-tree, which emphasizes the clustering step, as opposed to the underlying timing-driven Steiner-tree heuristic. The fundamental idea behind C-tree is to construct the tree in two levels (though multilevel clustering may be used as well). C-tree first clusters sinks with similar characteristics (criticality, polarity, and distance). The purpose of this step is to potentially isolate positive sinks from negative ones and noncritical 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 is computed where each cluster is treated as a sink. The top-level tree is then merged with the low-level trees to yield a solution for the entire net.

Fig. 3 presents a more detailed description of the C-tree framework. We assume the existence of two generic subroutines, clustering and timing-driven Steiner, which are described later. However, one could plug in a variety of implementations to achieve the clustering and routing functionalities within the C-tree framework.

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 looks through 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 to be computed over N_i and also the point where the top-level tree T_0 will connect to T_i. Although there are several possible ways to compute the tapping point, we choose tp_i to be a point on the bounding box of N_i closest to s_0. If s_0 lies within the bounding box, the tapping point is instead s_0 itself. Once the tapping point is chosen it is added to N_i in Step 4 as the source node and then timing-driven steiner is called on N_i to yield a tree T_i in Step 5. Step 6 then propagates the required arrival time up the subtree computed for T_i to the tapping point. The capacitance for the subtree is also updated at the tapping point. After these operations have been done for all the tapping points, N_0 consists of s_0 plus the k tapping points which 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.

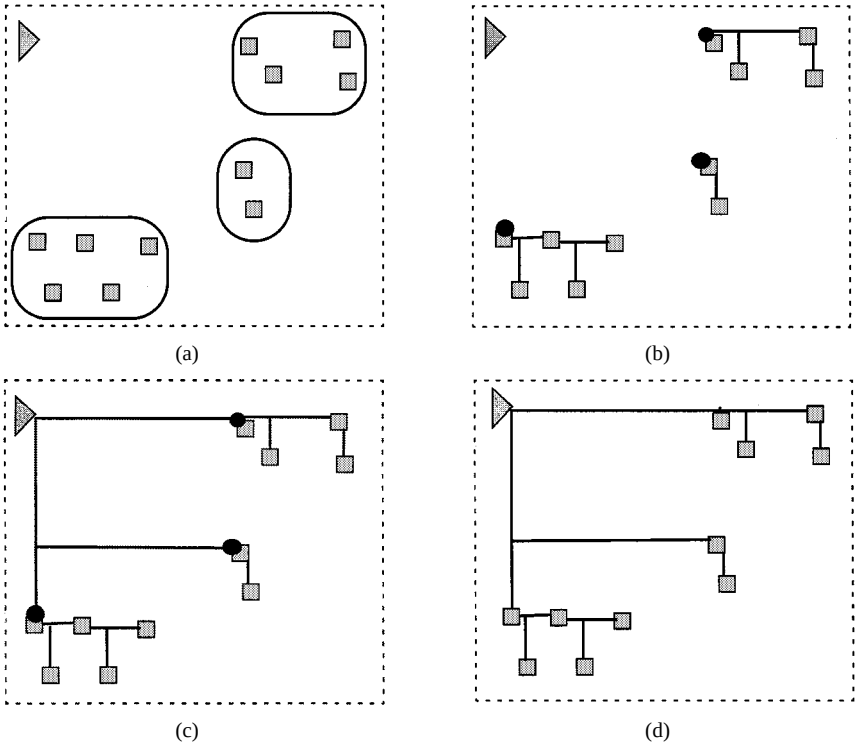


Fig. 4. Example execution of the two-level Steiner algorithm. (a) First, the sinks are clustered. (b) Then, a Steiner tree is built for each cluster. (c) A top-level Steiner tree connects the source to each cluster's tapping point. (d) The tree is then flattened.

Fig. 4 illustrates an example execution of the algorithm. In Fig. 4(a), a clustering of the sinks is performed. Note that in the example the clustering is geometric, but due to varying timing and polarity constraints, clusters certainly could overlap each other. In Fig. 4(b), the tapping point is shown for each cluster as a black circle and the Steiner trees are then computed for each cluster. In Fig. 4(c), the top-level Steiner tree which connects the source to the tapping points is computed and in Fig. 4(d), the tapping points are removed and the existing Steiner edges merged to yield a single tree for the entire net. The 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, the algorithm is sensitive to the actual clustering algorithm used, which we now describe.

B. Clustering-Distance Metric

The key to clustering any set of data is to devise a dissimilarity or distance metric between pairs of points. The sinks that we are clustering are characterized by three types of information: *spatial* (coordinates in the plane), *temporal* (required arrival times), and *spatial polarity*. We seek to define a distance metric that incorporates all of these elements. To do this, we first define spatial, temporal, and polarity metrics, then combine them using appropriate scaling into a single distance metric.

Appropriate spatial and polarity metrics are fairly straightforward. For two sinks s_i and s_j, let $s\mathrm{Dist}(s_i, s_j) = |x(s_i) - x(s_j)| + |y(s_i) - y(s_j)|$ denote the spatial (Manhattan) distance between two sinks and let $p\mathrm{Dist}(s_i, s_j) = |\mathrm{pol}(s_i) - \mathrm{pol}(s_j)|$ denote the polarity distance. The polarity distance has value zero when sinks have the same polarity and one otherwise.

Finding a temporal metric is trickier. First, RAT is not the only indicator of sink criticality. If two sinks s_i and s_j have the same RAT yet s_i is much further from the source than s_j, then s_i is more critical, since it will be much harder to achieve the RAT over the longer distance. An estimate of the achievable delay to s_i must be incorporated to reflect the distance from the source. If one assumes an optimally buffered direct connection from s_0 to s_i, with subtrees 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. We use the formula from [1] to denote $\mathrm{achDelay}(s_i)$, the potentially achievable delay from s_0 to s_i. Let $\mathrm{AS}(s_i) = \mathrm{RAT}(s_i) - \mathrm{achDelay}(s_i)$ be the potentially achievable slack for s_i. Now, $\mathrm{AS}(s_i)$ gives a better indicator of the criticality of s_i than $\mathrm{RAT}(s_i)$.

Yet a form such as $|\mathrm{AS}(s_i) - \mathrm{AS}(s_j)|$ still does not capture the desired behavior. For example, assume that the achievable slack values for three sinks are given by $\mathrm{AS}(s_1) = -1$ ns, $\mathrm{AS}(s_2) = 2$ ns, and $\mathrm{AS}(s_3) = 10$ ns. Sink s_1 is most critical while s_2 and s_3 are both noncritical. Thus, intuitively, s_2 is more similar to s_3 than to s_1, despite the 8-ns difference between s_2 and s_3. A temporal metric needs to capture that. Let $\mathrm{crit}(s_i)$ denote the criticality of s_i, where $\mathrm{crit}(s_i) = 1$ if s_i is the most critical sink and $\mathrm{crit}(s_i) \to 0$ as $\mathrm{AS}(s_i) \to \infty$. In other words, 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 zero and one. We propose the following measure of criticality:

$$\begin{aligned} \mathrm{crit}(s_i) &= e^{\alpha((m\mathrm{AS} - \mathrm{AS}(s_i))/(a\mathrm{AS} - m\mathrm{AS}))} \quad \text{where} \\ m\mathrm{AS} &= \min\{\mathrm{AS}(s_i) \,|\, 1 \le i \le n\} \quad \text{and} \\ a\mathrm{AS} &= \frac{\sum_{1 \le i \le n} \mathrm{AS}(s_i)}{n}. \end{aligned} \tag{1}$$

K-Center Algorithm (S, k)

Input: $S = \{s_1, \ldots, s_n\}$ ≡ Set of sinks
k ≡ Number of clusters

Output: $\{N_1, N_2, \ldots, N_k\}$ ≡ k-way clustering of S

1. Choose a random $s \in S$. Find $\hat{s} \in S$ such that $dist(s, \hat{s})$ is maximum. $W = \{\hat{s}\}$. Let $d = max\{dist(s, \hat{s}) | s \in S\}$. Set $N_1 = S$.
2. while $|W| < k$ do
3. Find $\hat{s} \in S/W$ such that $d = min\{dist(s, \hat{s}) | s \in W\}$ is maximized. $W = W \cup \{\hat{s}\}$.
4. Relabel seeds in W as $\{w_1, w_2, \ldots, w_{|W|}\}$ Let $\{N_1, N_2, \ldots, N_{|W|}\}$ be a $|W|$-way clustering where $N_i = \{w_i\}$ for $1 \le i \le |W|$.
5. for each $s \in S/W$ Find the cluster seed $w_i \in W$ such that $dist(s, w_i)$ is minimized. Add s to cluster N_i.
6. return $\{N_1, N_2, \ldots, N_k\}$.

Fig. 5. K-center clustering algorithm over a set of sinks S.

Here, $m\mathrm{AS}$ and $a\mathrm{AS}$ are the minimum and average AS values over all sinks and $\alpha > 0$ is a user parameter. One can see that, indeed, $\mathrm{crit}(s_i)$ is one when $\mathrm{AS}(s_i) = m\mathrm{AS}$ and zero when $\mathrm{AS}(s_i)$ goes to infinity. For a sink s_i with average achievable slack ($\mathrm{AS}(s_i) = a\mathrm{AS}$), then $\mathrm{crit}(s_i) = e^{-\alpha}$ is about 0.135 when α is set to two.[3] This average sink will have a criticality much closer to that of a sink with infinite AS as opposed to minimum AS. We can now define temporal distance as the difference in criticalities, i.e., $t\mathrm{Dist}(s_i, s_j) = |\mathrm{crit}(s_i) - \mathrm{crit}(s_j)|$.

If two sinks s_i and s_j are both extremely noncritical, but have different achievable slacks, their temporal distance will be practically zero. For example, assume that $m\mathrm{AS} = -1$ ns, $a\mathrm{AS} = 1$ ns, $\alpha = 2$, and the two sinks have achievable slacks of 7 and 9 ns. The respective criticalities are e^{-8} and e^{-12}, so $t\mathrm{Dist}(s_i, s_j) \approx 0.0004$.

Both temporal and polarity distances are on a zero to one scale, so we wish to scale spatial distance to make combining the terms easier for the complete distance metric. Let $s\mathrm{Diam}(N) = \max\{s\mathrm{Dist}(s_i, s_j) | 1 \le i, j \le n\}$ be the spatial diameter of the set of sinks. The scaled distance between two sinks can be expressed as $s\mathrm{Dist}(s_i, s_j)/(s\mathrm{Diam}(N))$. Our complete distance metric is a linear combination of the spatial, temporal, and polarity distances

$$\mathrm{dist}(s_i, s_j) = \beta \cdot \frac{s\mathrm{Dist}(s_i, s_j)}{s\mathrm{Diam}(N)} + (1 - \beta) \cdot t\mathrm{Dist}(s_i, s_j) + p\mathrm{Dist}(s_i, s_j). \quad (2)$$

The parameter β lies between zero and one and trades off between spatial and temporal distance. In our experiments, we use $\beta = 0.65$ based on empirical studies. Observe that the distance between two sinks with the same polarity will always be less than or equal to the distance between two sinks with opposite polarity. This occurs because two sinks with the same polarity have their distance bounded above by one, while two sinks with opposite polarity have their distance bounded below by one. This property ensures that polarity takes precedence over spatial and temporal distance in determining dissimilarity, which is important to avoiding the behavior shown in Fig. 2(a).

[3]In our experiments, we found using $\alpha = 2$ generates good results, which is used in Section IV.

C. Clustering

For clustering sinks,[4] we adopt the K-center heuristic [15], 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 (e.g., bottom-up matching and complete linkage) 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. Let $\mathrm{diam}(N_i) = \max_{p,q \in N_i}\{\mathrm{dist}(p, q)\}$ be the diameter of any set of points N_i. For geometric instances, K-center guarantees that the maximum diameter of any cluster is within a factor of two of the optimal solution [15].

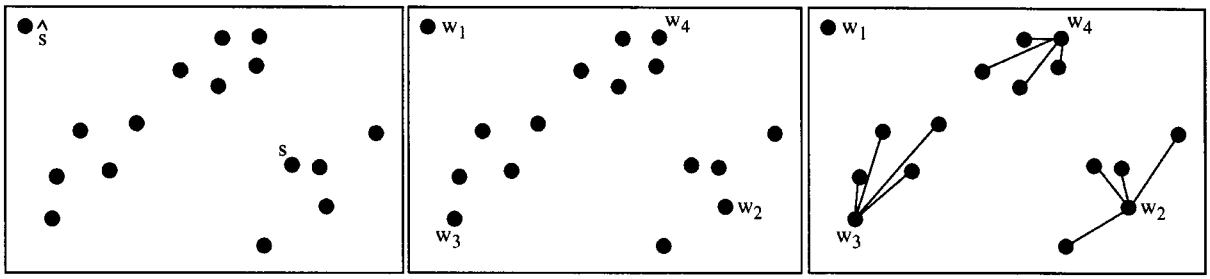
The complete description of the K-center algorithm is shown in Fig. 5. Step 1 picks a random sink s, then identifies the sink $\hat{s}$ furthest away from s, which will lie on the periphery of the data set. This step identifies $\hat{s}$ as the first cluster seed, which are all contained in the set W. Steps 2–5 iteratively find $|W|$-way clusterings for N until the ratio of the diameter of the largest current cluster to the diameter of n falls below the threshold D. Step 3 identifies the next seed which is furthest away from already identified seeds. Steps 4–5 then form a clustering by assigning each sink to the cluster corresponding to its closest seed. After the diameter threshold is reached in Step 2, Step 6 returns the final clustering. The procedure has $O(nk)$ time complexity.

Fig. 6 illustrates an example of the K-center algorithm applied to a 2-D data set with 16 points, where $k = 4$. In Fig. 6(a) a random point s is chosen and then the point $\hat{s}$ that is furthest from s is identified. In Fig. 6(b), this is relabeled as w_1, a cluster seed. The order that the four seeds were identified are indicated by the subscripts: w_2 is furthest from w_1, w_3 is furthest from both w_1 and w_2, and w_4 is the furthest point from w_1, w_2, and w_3. In Fig. 6(c), each point is mapped to its closest seed, revealing four clusters.

D. Timing-Driven Steiner-Tree Construction

For the timing-driven Steiner-tree construction, we adopt the Prim–Dijkstra tradeoff method from [5]. The algorithm

[4]One could modify the sink clustering algorithm to forbid the bounding box of a cluster to intersect the source node. We did not notice any appreciable change in results with this variation, but it may be worth more detailed investigation.



(a) (b) (c)

Fig. 6. 16-point example illustrating the K-center algorithm. (a) A random seed s is chosen and $s\hat{}$ furthest from s is identified. (b) This point is relabeled as $w1$ and the next three furthest points $w2$, $w3$, and $w4$ are found. (c) Each point is then clustered to its closest seed.

trades off between Prim's minimum spanning tree algorithm and Dijkstra's shortest path tree algorithm via a parameter c, which lies between zero and one. The justification behind this approach is that Prim's algorithm yields minimum wire length (for a spanning tree), while Dijkstra's results in minimum tree radius. A tradeoff captures the desirable properties behind both approaches.

Our implementation is as follows. We run the Prim–Dijkstra algorithm for $c = 0.0, 0.25, 0.5, 0.75, 1.0$ for the clusters and the top-level tree. After each spanning tree construction, we run a postprocessing algorithm to remove overlapping edges and generate a Steiner tree. Of the five constructions, the tree T that maximizes $\mathrm{slack}(T)$[5] is selected. A second postprocessing step is then invoked to reduce delay further. In this step, each sink is in turn visited, the connection from the sink to the existing tree is ripped up, and alternative connections to the tree are attempted. Any connection which either decreases wire length or improves slack is preserved.

Certainly, alternative timing-driven Steiner-tree algorithms could be used instead. In fact, we tried using the P-tree with area optimization (P-treeA) algorithm [20], which generates the tree with the best timing properties such that it has minimum wire length and obeys a given sink permutation. We found that P-treeA would sometimes yield trees with large radius and, hence, poor timing characteristics. P-tree with area and timing optimization (P-treeAT) overcomes this problem, but uses significantly more runtime. We chose the Prim–Dijkstra algorithm because it is simple to implement, it is efficient and scalable, and it outperformed the critical sink constructions of [7] in separate experiments.

IV. Experimental Results

For our experiments, we identified 12 difficult nets on various industrial designs. The polarity characteristics and timing constraints for the nets are summarized in Table I.[6]

We compare C-tree to both the P-tree [20] and Prim–Dijkstra [5] timing-driven Steiner constructions. P-tree was shown to yield better timing results than either the SERT [7] or A-tree [11] constructions. P-tree actually consists of two algorithms: P-treeA seeks to minimize area (or wire length when there is no wire sizing), while P-treeAT generates a family of solutions that trade off between area and timing. The Prim–Dijkstra algorithm is equivalent to "flat" C-tree when the number of clusters equals the number of sinks. For each tree generated, we run van Ginneken-style buffer insertion using a library of five noninverting and two inverting buffers to generate a family of solutions. We also compare to a buffered P-tree (BP-tree), which simultaneously inserts buffers and performs the Steiner routing. Like P-tree, BP-tree also has two modes, which we suffix with either normal (N) or fast (F).

TABLE I
Polarity and Temporal Characteristics of the 12 Nets Used for Experimentation

Net Name	Sinks			RAT	
	+	-	Total	min	max
mcu	8	10	18	6195	6596
n107	7	10	17	1902	2560
n313	9	10	19	1233	6704
n869	11	10	21	1054	6390
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

A. Algorithm Comparisons

The results are summarized in Tables II and III. The results are split into two tables, since the data could not fit into a single table. 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 2-GB machine). The next row contains results for BP-tree in normal mode except for the largest net (also because it ran out of memory). The first C-tree row uses the number of clusters equal to the number of sinks, giving the results for "flat" C-tree. Results are also presented for C-tree for a decreasing number of clusters to show the tradeoff for using a different number of clusters. For each algorithm, we present the following in the two tables.

1) The slack to the most critical sink (in picoseconds) and wire length of the tree before the buffer-insertion optimization step.

[5]Note that for the clusters, no driver exists. We choose a mid-level buffer from the technology to use as a phantom driver for the slack calculation.

[6]The polarity constraints were actually randomly assigned for these test cases, yet they represent the difficulties we have seen for real instances.

TABLE II
ALGORITHM COMPARISONS FOR THE FIRST SIX NETS

Net Name	Algorithm	# Clusts	Before Opt		Min Opt		Mid Opt		Full Opt		Post Process		CPU
			slack	wire	bufs	slack	bufs	slack	bufs	slack	slack	wire	
mcu	P-TreeAT	1	5948	3758	4	5877	8	5994	11	5999	5999	3758	1.1
	P-TreeA	1	5910	3298	5	5697	8	5778	11	5782	5810	4453	0.4
	BP-TreeN	1	---	---	5	5961	7	5976	9	5988	---	---	61.5
	C-Tree	18	5943	3743	6	5995	9	6013	11	6014	6014	3743	0.2
	C-Tree	10	5940	3635	4	5887	7	6015	10	6018	6018	3576	0.2
	C-Tree	5	5884	5174	2	5863	6	6028	10	6032	6032	5084	0.3
	C-Tree	2	5881	5380	1	5865	5	6028	8	6033	6034	5277	0.3
n107	P-TreeAT	1	1678	1091	5	1825	8	1835	11	1837	1837	1091	1.9
	P-TreeA	1	1678	1086	5	1825	8	1833	11	1835	1835	1098	0.2
	BP-TreeN	1	---	---	5	1831	7	1848	9	1864	---	---	
	C-Tree	17	1678	1086	5	1825	7	1831	8	1832	1832	1091	0.1
	C-Tree	11	1665	1265	5	1824	10	1871	11	1872	1872	1141	0.2
	C-Tree	4	1604	2065	2	1808	4	1863	5	1865	1866	1900	0.2
	C-Tree	2	1625	1781	1	1759	3	1863	4	1865	1866	1755	0.1
n313	P-TreeAT	1	646	5290	8	1161	9	1207	10	1212	1212	5285	1.2
	P-TreeA	1	647	5285	8	1161	9	1207	10	1212	1212	5285	0.5
	BP-TreeN	1	---	---	5	1062	6	1223	6	1223	---	---	
	C-Tree	19	646	5280	8	1059	9	1151	10	1202	1202	5280	0.2
	C-Tree	14	608	5748	8	1170	9	1212	10	1218	1218	5742	0.2
	C-Tree	6	222	10475	4	962	6	1197	7	1203	1203	9028	0.4
	C-Tree	2	301	9541	1	759	3	1200	4	1206	1206	9541	0.3
n869	P-TreeAT	1	127	4241	8	185	13	310	17	315	315	4236	4.0
	P-TreeA	1	131	4213	7	284	11	380	15	387	387	4213	0.9
	BP-TreeN	1	---	---	5	319	8	529	11	552	---	---	
	C-Tree	21	130	4213	7	280	10	376	12	378	473	4451	0.5
	C-Tree	6	113	4337	3	468	6	558	9	578	578	4337	0.7
	C-Tree	4	91	4533	2	451	6	573	10	582	582	4533	1.0
	C-Tree	2	-114	8083	1	156	4	595	7	610	610	8083	1.7
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	---	---	748.1
	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

2) Three of the family of solutions generated by the buffer-insertion algorithm. The min-opt solution is the solution with the minimum number of buffers required to fix polarity constraints. The full-opt solution is the one that yields the maximum slack, regardless of the buffers used and mid-opt reflects a solution in between the min and full solutions. Although the problem formulation seeks to evaluate the entire family, the three solutions give a reasonable picture of the tradeoff curve generated by the family.
3) The slack to the most critical sink and wire length after a postprocessing step on the full-opt buffered solution. Potentially a tree with significantly extra wire length was used to guide the buffer insertion. Once buffers are inserted, some of this additional wire length may be eliminated via small changes in the route. Our algorithm sought to reduce wire length as long as it did not increase slack while maintaining the locations and topology from the full-opt buffered tree.
4) The total central processing unit (CPU) time for the entire process (tree construction, buffer insertion, and postprocessing). Runtimes are reported for a Sun SparcUltra-60 with 2 GB of memory.

We make several observations.

1) For the solution in the family with highest slack, C-tree was able to find solutions with slacks at least as high as P-treeA, P-treeAT, or flat C-tree for at least one clustering (except for n873 for which C-tree's slack was inferior by 1 ps). Sometimes the C-tree slacks were significantly better (e.g., n869, n870, and big1), but most of the time the highest slacks were fairly indistinguishable among the algorithms.
2) For the solution in the family with highest slack, C-tree found a better solution than BP-tree for seven of the 12

TABLE III
ALGORITHM COMPARISONS FOR THE SECOND SET OF NETS

Net Name	Algorithm	# Clusts	Before Opt		Min Opt		Mid Opt		Full Opt		Post Process		CPU
			slack	wire	bufs	slack	bufs	slack	bufs	slack	slack	wire	
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	4	-1858	7937	1	-1037	4	503	7	545	549	7744	0.8
	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	---	---	784.1
	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	4	-885	5736	2	-640	4	54	6	83	83	5702	1.1
	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	4	-2751	4546	3	33	7	250	10	267	267	4546	1.4
	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	5	-1319	27224	1	-330	5	1653	8	1685	1685	27134	7.5
	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	28	-1682	13995	15	-704	22	-74	29	-68	-68	12340	3.2
	C-Tree	12	-1781	15117	6	-890	12	-64	18	-34	-34	14691	2.6
	C-Tree	6	-1862	16179	1	-1188	6	-41	13	-33	-32	16119	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	21	-282	10300	11	652	14	1013	17	1021	1022	9422	1.5
	C-Tree	10	-375	11225	6	433	10	1019	14	1038	1038	10819	1.2
	C-Tree	4	-317	10616	1	224	5	981	11	1020	1028	10522	1.8
	C-Tree	2	-264	9965	1	278	5	992	9	1028	1028	9962	0.9

nets. Overall, the differences in delay were fairly small, with C-tree's best solution averaging a 29-ps improvement over BP-tree's best solution.

3) The more clusters used by C-tree, the fewer the number of buffers are needed to fix polarity constraints. With two clusters, one inverting buffer is always sufficient to fix polarity, which shows C-tree handles the case in Fig. 2. However, fewer clusters results in additional wire length. Indeed, the extreme case of two clusters almost doubles 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. Depending on the requirements of the user, the number of clusters can be used within C-tree to trade off wire length with buffer area. Fig. 7 illustrates this tradeoff for six of the nets. Wire length generally decreases as the number of buffers increases, especially when only a few buffers are used. For any number of clusters greater than $n/2$, C-tree was able to obtain slack comparable to that of the best approach. Thus, any number of clusters between, say, 2 and $n/2$ are reasonable choices for optimizing the timing.
4) The postprocessing step did not affect slack much at all, but occasionally reduced wire length (e.g., for big3).
5) BP-tree and P-tree AT are clearly the most inefficient algorithms, as runtimes were over 100 times that of

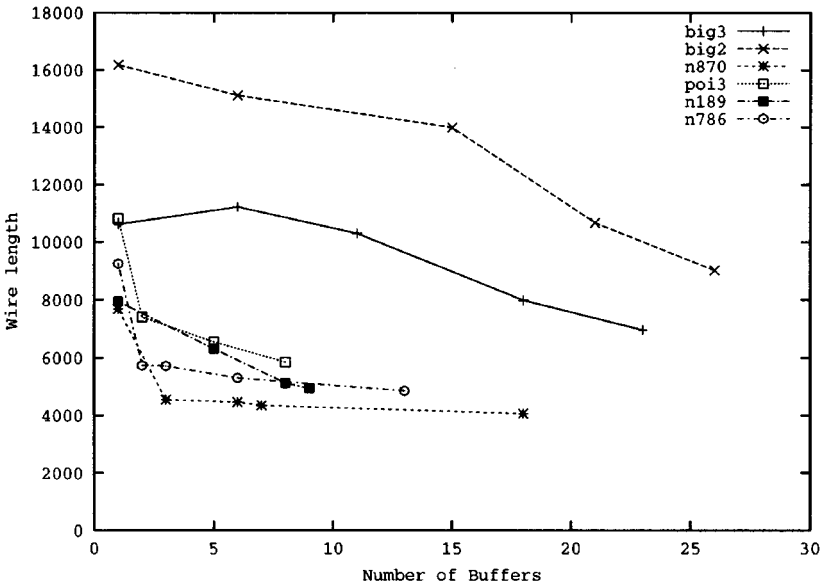


Fig. 7. Tradeoff between the number of buffers inserted and wire length for different degrees of clustering within C tree.

C-tree for n870 and they could not complete all of the test cases. P-treeA is slightly more inefficient than the Prim– Dijkstra approach, but C-tree is actually the fastest of the three constructions. For example, for big1 (the largest net), C-tree alone took under 0.2 s to run for each of the clusterings reported in Table III, while flat C-tree took 0.6 s. For C-tree, the dynamic-programming buffer-insertion algorithm dominates the runtime of the entire flow.

6) For the larger nets, P-treeA, BP-tree, and flat C-tree required many more buffers to find a feasible solution than C-tree. For example, P-tree required 32, 27, and 27 buffers to satisfy polarity constraints for big1, big2, and big3, respectively, while BP-tree required 99, 20, and 19 buffers (BP-tree in fast mode is much more wasteful in buffering resources). Via clustering, C-tree could generally find a solution with slack at least as high as P-tree with 4, 6, and 9 buffers, respectively. For n870, C-tree with four clusters found a solution with seven buffers and slack 250 ps, which is 128 ps more than the best result found by P-treeAT (which needs at least 17 buffers to satisfy constraints). For big1, a five-cluster C-tree solution with five buffers has slack 1653 ps, which is over 400 ps better than best slacks obtained by P-tree or flat C-tree.

B. Variations

It may seem a bit surprising that there is little slack differentiation among the algorithms. The reason for this might be that a single critical sink dominates the slack value. For example, in net n313, the minimum RAT is 1233 ps, which is also an upper bound on the slack. From Table II, observe that a majority of the full-opt solutions obtain slack value of over 1200 ps, which is close to optimum. Thus, for this net, the critical sink must lie close to the source, which makes it an easy to obtain a good slack result (though still hard to potentially minimize resources).

To reduce the impact of this effect, we ran the same experiments on the four largest nets with an RAT value of zero for every sink. This serves to isolate the effects of polarity on the difficulty of the instances. The results are summarized in Table IV. Here, the advantages of C-tree are magnified, especially for net n870. C-tree obtains slacks as low as -511 compared to -1408 for P-tree and -1808 for flat C-tree. From looking at the topology of the solutions, we observed that P-tree and flat C-tree contain chains of inverters that alternately drive positive and negative sinks over a short distance. These chains cause the huge difference in delays. C-tree avoids these chains by clustering according to polarity. The other three nets also show large improvements for C-tree.

Finally, we ran the same experiments using the original RAT values, but setting all sinks to positive polarity. In this case, we observed very little difference among the algorithms. Thus, at least for this suite of test cases, the case of Fig. 1 is not nearly as critical as the case in Fig. 2. It is the polarity differences that make these instances difficult.

C. Choosing the Right Number of Clusters

There clearly are tradeoffs between the number of clusters and resource utilization. Typically as the number of clusters decreases, the number of buffers also decreases, wire length increases and the slack generally improves. However, it is difficult to know *a priori* what the right number of clusters will be in advance. Intuitively, the amount clustering performed should increase with the number of sinks of a net. The larger a net, the more susceptible it is to wasting buffering resources as in Fig. 2.

Even two instances with the same number of sinks could require different clustering solutions. For example, one instance could have sinks spread far apart, while another could have natural clusters of sinks. In this example, we would want the latter instance to have fewer clusters. Thus, for the following experiments, we modified the stopping criteria of Fig. 3 to instead stop when the diameter of the largest cluster falls below a certain threshold (where the diameter of cluster N is $\max\{\mathrm{dist}(s_i, s_j) \| s_i, s_j \in N\}$). Since the distance metric is scaled for every instance, we can use constant values of the

TABLE IV
EXPERIMENTAL RESULTS WITH ALL SINK RAT VALUES SET TO ZERO

Net Name	Algorithm	# Clusts	Before Opt		Min Opt		Mid Opt		Full Opt		Post Process	
			slack	wire	bufs	slack	bufs	slack	bufs	slack	slack	wire
n870	P-TreeA	1	-3319	4089	17	-1409	23	-1349	29	-1345	-1345	4089
	C-Tree	43	-3307	4062	18	-1896	22	-1812	27	-1808	-1808	4062
	C-Tree	17	-3431	4356	10	-1064	18	-731	26	-721	-721	4356
	C-Tree	8	-3515	4560	5	-805	7	-664	10	-608	-608	4555
	C-Tree	4	-3500	4546	3	-798	4	-683	6	-597	-597	4546
	C-Tree	2	-4522	7689	1	-2710	3	-562	5	-516	-511	5964
big1	P-TreeA	1	-3065	14734	32	-1461	43	-1222	53	-1206	-1186	16104
	C-Tree	88	-2650	16068	31	-1021	41	-916	51	-904	-904	16068
	C-Tree	37	-3385	24043	22	-1233	34	-619	46	-609	-608	22996
	C-Tree	14	-3640	26833	8	-1124	14	-576	21	-508	-498	26650
	C-Tree	5	-3570	26737	2	-1957	9	-540	16	-508	-500	26394
	C-Tree	2	-3426	27629	1	-1975	9	-536	24	-502	-501	27545
big2	P-TreeA	1	-1577	8899	28	-1144	36	-617	48	-607	-606	9002
	C-Tree	79	-1471	8960	28	-626	36	-562	45	-556	-556	8960
	C-Tree	31	-1864	13292	18	-937	30	-453	42	-448	-447	12619
	C-Tree	12	-2074	16134	7	-1041	10	-401	16	-351	-351	15977
	C-Tree	7	-2167	17089	3	-1270	8	-372	14	-351	-349	17219
	C-Tree	2	-1811	13267	1	-1163	4	-364	8	-348	-348	13267
big3	P-TreeA	1	-1204	6907	26	-839	35	-554	43	-545	-545	6907
	C-Tree	63	-1175	6794	26	-807	30	-660	40	-658	-658	6794
	C-Tree	22	-1440	9879	10	-625	16	-373	22	-357	-353	9160
	C-Tree	11	-1609	11911	6	-834	10	-341	15	-326	-326	11645
	C-Tree	7	-1548	11254	4	-852	8	-354	12	-327	-327	11249
	C-Tree	2	-1436	10098	1	-981	2	-322	2	-322	-322	9967

TABLE V
COMPARISONS OF SLACK IMPROVEMENT (VERSUS ESS), BUFFERING, AND WIRING RESOURCES FOR VARIOUS GROUPS OF NETS AND EIGHT DIFFERENT C-TREE DIAMETER THRESHOLDS

Net Category	Measurement	ESS	Diameter Threshold							
			0.00	0.05	0.10	0.15	0.20	0.30	0.50	0.75
5-9 (66)	Slack improvement	0	280	375	378	359	378	329	322	372
	Buffers	133	138	136	132	130	136	134	131	130
	Wire length	15.9	16.1	16.2	16.4	16.5	16.7	17.0	17.4	19.1
10-19 (70)	Slack improvement	0	3908	3584	3684	4301	4509	4603	4687	4676
	Buffers	354	336	351	334	325	307	291	286	256
	Wire length	58.3	64.8	62.7	64.0	68.2	70.0	70.5	70.8	69.8
20-29 (139)	Slack improvement	0	1062	1215	1464	1857	1941	1806	2490	2517
	Buffers	495	495	496	455	449	414	365	318	255
	Wire length	42.3	42.9	46.1	49.4	51.5	53.0	55.0	54.2	52.0
30-49 (45)	Slack improvement	0	437	473	729	650	693	817	855	895
	Buffers	191	193	185	176	160	166	153	134	121
	Wire length	14.2	14.4	15.7	16.8	17.3	17.7	18.1	17.9	17.7
50-74 (37)	Slack improvement	0	1826	2430	2589	2662	2754	2868	2370	2189
	Buffers	579	552	472	405	371	375	349	318	323
	Wire length	102.9	107.8	123.9	133.1	134.3	137.0	154.8	160.7	162.6
75-100 (30)	Slack improvement	0	3173	3416	3697	3836	3977	4140	3854	3758
	Buffers	354	306	255	255	243	241	207	202	189
	Wire length	66.9	72.2	81.1	86.2	88.1	90.1	98.5	95.3	93.8

Number of nets in each class is shown in parentheses in the first column.

diameter threshold D over a variety of instances. For example, if $D = 0$, every cluster will have zero diameter which means every sink is in its own cluster (corresponding to flat C-tree). If $D = 1$, this will create two clusters if there are sinks with opposite polarities and one cluster otherwise. We examine various diameter thresholds between zero and one to try to grasp the appropriate diameter value for a given number of sinks.

In the following experiments, we ran C-tree on 387 large nets in an industrial test case with 274 000 cells. We grouped the nets into six categories according to their number of sinks: 10–19, 20–29, 30–49, 50–74, and 75–100. For the nets in each category, we ran C-tree with various diameter thresholds and compared the results to a Steiner-tree construction called electrical subsystem (ESS) (an internal IBM tool) that seeks a minimum wire length routing topology. We measure the total improvement in slack over all the nets in nanoseconds amd the number of buffers inserted and the total wire length in design millimeters. The results are shown in Table V.

Not surprisingly, ESS always yields the minimum wire-length result over all the constructions while C-tree seeks a timing based objective. The flat Prim–Dijkstra construction (corresponding to a diameter threshold of zero) has slightly more wire length than ESS since it tries to trade off wire length with the radius of the tree. As the diameter threshold increases, the wire length increases as well, though sometimes there is a slight dropoff in wire length as the threshold reaches 0.5.

Observe that the number of buffers decreases as the diameter threshold increases. Further, the trend is more pronounced for the larger nets. It is harder to observe a trend in terms of slack improvement. For example, for the 75–100 class of nets, the most improvement is observed for a diameter threshold of 0.3, while for the 20–29 class of nets, 0.3 yields the lowest result for all thresholds larger than 0.1. In general, larger diameters thresholds tend to yield better slack values while using few buffers, though at a potentially significant price for wire length. Since many modern designs are wire congested, we believe it is better to keep the diameter threshold relatively low (e.g., below 0.1) for most classes of instances so that the wire does not increase by more than 5%–10% over the ESS (e.g., minimum wire length). However, if a net is the bottleneck for the design, lying in the most critical path, we would recommend running C-tree followed by buffer insertion for three or four different diameter threshold values and picking the one which yields the best timing. Designs that are more area constrained would clearly benefit from using higher diameter threshold values.

V. Conclusion

We have identified a class of buffered Steiner-tree instances for which existing algorithms are inadequate. These instances have a large number of sinks and varying temporal and polarity constraints. We proposed a two-level clustering based heuristic called C-tree for these instance types. Our clustering heuristic utilizes a new distance metric that combines spatial, temporal, and polarity characteristics. Experiments on industrial nets show that C-tree is able to obtain results with slack equal to or better than previous approaches while using fewer buffers. Compared to simultaneous buffer-insertion and Steiner-tree construction, C-tree obtains better slack on average while using significantly less CPU time (and buffering resources). By adjusting the number of clusters, C-tree can trade off between buffering and wiring resources, though we are still hoping to be able to identify clustering stopping criteria to automatically identify the "sweet spot" in the resource/performance tradeoff. We hope that this paper stimulates more research on these types of problems.

While experimenting with industrial designs, we have found that, while performing placement-driven synthesis on application-specific integrated-circuit designs, there exist nets with several hundred sinks that require optimization. Further, excellent solution quality is critical as these nets often lie on a negative slack path because the net itself has such poor delay characteristics. We believe issues such as alternative tree constructions within clusters, different mechanisms for locating the tapping point, multilevel instead of two-level clustering, and alternative distance functions could improve our approach. We also need to identify more difficult instances for which different approaches can distinguish themselves.

References

[1] C. J. Alpert and A. Devgan, "Wire segmenting for improved buffer insertion," in *Proc. 34th IEEE/ACM Design Automation Conf.*, June 1998, pp. 588–593.

[2] C. J. Alpert, A. Devgan, and S. T. Quay, "Buffer insertion for noise and delay optimization," in *Proc. 35th IEEE/ACM Design Automation Conf.*, June 1998, pp. 362–367.

[3] ——, "Buffer insertion with accurate gate and interconnect delay computation," in *Proc. 36th IEEE/ACM Design Automation Conf.*, June 1999, pp. 479–484.

[4] C. J. Alpert, G. Gandham, J. Hu, J. L. Neves, S. T. Quay, and S. S. Sapatnekar, "Steiner tree optimization for buffers, blockages and bays," *IEEE Trans. Computer-Aided Design*, vol. 20, pp. 556–562, Apr. 2001.

[5] 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. Computer-Aided Design*, vol. 14, pp. 890–896, July 1995.

[6] H. B. Bakoglu, *Circuits, Interconnections and Packaging for VLSI*. Reading, MA: Addison-Wesley, 1990.

[7] K. D. Boese, A. B. Kahng, B. A. McCoy, and G. Robins, "Near-optimal critical sink routing tree constructions," *IEEE Transactions Computer-Aided Design*, vol. 14, pp. 1417–1436, Dec. 1995.

[8] C. C. N. Chu and D. F. Wong, "Closed form solution to simultaneous buffer insertion/sizing and wire sizing," in *Proc. Int. Symp. Physical Design*, Apr. 1997, pp. 192–197.

[9] J. Cong, "Challenges and opportunities for design innovations in nanometer technologies," *SRC Working Papers*, Dec. 1997.

[10] J. Cong, L. He, C.-K. Koh, and P. H. Madden, "Performance optimization of VLSI interconnect layout," *Integr. VLSI J.*, vol. 21, no. 1, pp. 1–94, 1996.

[11] J. Cong, K. S. Leung, and D. Zhou, "Performance-driven interconnect design based on distributed *RC* delay mode," in *Proc. IEEE/ACM Design Automation Conf.*, June 1993, pp. 606–611.

[12] J. Cong and X. Yuan, "Routing tree construction under fixed buffer locations," in *Proc. IEEE/ACM Design Automation Conf.*, June 2000, pp. 379–384.

[13] S. Dhar and M. A. Franklin, "Optimum buffer circuits for driving long uniform lines," *IEEE J. Solid-State Circuits*, vol. 26, pp. 32–40, Jan. 1991.

[14] W. C. Elmore, "The transient response of damped linear network with particular regard to wideband amplifiers," *J. Appl. Phys.*, vol. 19, pp. 55–63, Jan. 1948.

[15] T. F. Gonzalez, "Clustering to minimize the maximum intercluster distance," *Theor. Comput. Sci.*, vol. 38, pp. 293–306, 1985.

[16] A. Jagannathan, S.-W. Hur, and J. Lillis, "A fast algorithm for context-aware buffer insertion," in *Proc. IEEE/ACM Design Automation Conf.*, June 2000, pp. 368–373.

[17] M. Lai and D. F. Wong, "Maze routing with buffer insertion and wire-sizing," in *Proc. IEEE/ACM Design Automation Conf.*, June 2000, pp. 374–378.

[18] 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 IEEE/ACM Design Automation Conf.*, June 1996, pp. 395–400.

[19] 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*, vol. 31, pp. 437–447, Mar. 1996.

[20] ——, "Simultaneous routing and buffer insertion for high performance interconnect," in *Proc. 6th Great Lakes Symp. VLSI*, Mar. 1996, pp. 148–153.

[21] T. Okamoto and J. Cong, "Buffered Steiner tree construction with wire sizing for interconnect layout optimization," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, July 1996, pp. 44–49.

[22] L. P. P. P. van Ginneken, "Buffer placement in distributed RC-tree networks for minimal Elmore delay," in *Proc. Int. Symp. Circuits and Systems*, May 1990, pp. 865–868.

[23] H. Zhou, D. F. Wong, I.-M. Liu, and A. Aziz, "Simultaneous routing and buffer insertion with restrictions on buffer locations," in *Proc. ACM/IEEE Design Automation Conf.*, June 1999, pp. 96–99.

Charles J. Alpert (S'92–M'96) received the B.S. degree in math and computational sciences and the B.A. degree in history from Stanford University, Stanford, CA, in 1991 and the Ph.D. degree in computer science from the University of California, Los Angeles, in 1996.

He is currently a Research Staff Member with the IBM Austin Research Laboratory, Austin, TX. His current research interests include placement, interconnect synthesis, clock distribution, and global routing.

Dr. Alpert received the Best Paper Award at the ACM/IEEE Design Automation Conference in 1994, 1995, and 2001 and the SRC Mahboob Khan Outstanding Mentor Award in 2001.

Gopal Gandham received the B. Tech. degree in electronics and communication engineering from Andhra University, Waltair, India, in 1994 and the M.S. degree in automation and computer vision from the Indian Institute of Technology, Kharagpur, India, in 1996.

He is currently an Advisory Engineer with the IBM Corporation, East Fishkill, NY. His current research interests include phyiscal design and optimization on deep-submicrometer design flow.

Milos Hrkic received the B.S. degree in computer science from the University of Illinois, Chicago, in 2000. He is currently working toward the Ph.D. degree in computer science at the same university.

He was an Intern with the IBM Austin Research Laboratory, Austin, TX, in 2001. His current research interests include VLSI interconnect synthesis.

Jiang Hu received the B.S. degree in optical engineering from Zhejiang University, Hangzhou, China, in 1990, the M.S. degree in physics from the University of Minnesota, Duluth, in 1997, and the Ph.D. degree in electrical engineering from the University of Minnesota, Minneapolis, in 2001.

He is currently with the IBM Microelectronics Division, Austin, TX, working on VLSI CAD tools development. His current research interests include VLSI physical design, especially on interconnect routing, optimization, and planning.

Dr. Hu received the Best Paper Award at the Design Automation Conference in 2001.

Andrew B. Kahng was born in San Diego, CA, in October 1963. He received the A.B. degree in applied mathematics/physics from Harvard College, Cambridge, MA, and the M.S. and Ph.D. degrees in computer science from the University of California at San Diego, La Jolla.

He was with the Computer Science Department, University of California, Los Angeles, from 1989 to 2000, most recently as Professor and Vice-Chair. He is currently a Professor of Computer Science and Engineering and Electrical and Computer Engineering with the University of California at San Diego. He has authored or coauthored over 160 papers in the VLSI CAD literature, centering on physical layout and performance analysis. His current research interests include VLSI physical layout design and performance analysis, combinatorial and graph algorithms, and stochastic global optimization.

Prof. Kahng received the National Science Foundation Young Investigator Award and a Design Automation Conference Best Paper Award. He was the founding General Chair of the ACM/IEEE International Symposium on Physical Design (ISPD), a cofounder of the ACM Symposium on System-Level Interconnect Prediction (SLIP), Technical Program Chair of the 2001 Electronic Design Processes Symposium of the IEEE Design Automation and Test Committee, and is also on the steering committees of ISPD-2001 and SLIP-2001 . Since 1997, he has defined the physical design roadmap for the International Technology Roadmap for Semiconductors (ITRS) and is currently Chair of the U.S. and International Technical Working Groups for Design in the 2001 ITRS renewal.

John Lillis received the M.S. and Ph.D. degrees in computer science from the University of California at San Diego, La Jolla, in 1993 and 1996, respectively.

From 1996 to 1997, he was a Post-Doctoral Researcher with the University of California, Berkeley, supported in part by the National Science Foundation CISE program. He joined the Electrical Engineering and Computer Science Department, University of Illinois, Chicago, in 1997, where he is currently an Assistant Professor of Computer Science. His current research interests include design automation for VLSI, particulary physical design and timing optimization and combinatorial optimization.

Dr. Lillis received the National Science Foundation CAREER award in 1999.

Bao Liu was born in Guilin, China, in 1973. He received the B.S. and the M.S. degrees in electrical engineering from Fudan University, Shanghai, China, in 1993 and 1996, respectively. He is currently working toward the Ph.D. degree in computer science and engineering at the University of California at San Diego, La Jolla.

His Master's thesis studied FPGA implementation of VLSI DRC algorithm. He was with the China IC Design Center, Beijing, China, and has also interned at Cadence Design Systems in 1999 and Conexant Systems in 2000. His current research interests include VLSI interconnect construction and estimation.

Stephen T. Quay received the B.S. degree in electrical engineering and the B.S. degree in computer science from Washington University, St. Louis, MO, in 1983.

Since 1983, has worked in many areas of chip layout and analysis with IBM in Endicott, NY, and Austin, TX. He is currenlty a Senior Engineer for IBM Microelectronics, Austin, TX, where he develops design automation applications for interconnect performance optimization.

Sachin S. Sapatnekar received the B.Tech. degree from the Indian Institute of Technology, Bombay, India, in 1987, the M.S. degree from Syracuse University, Syracuse, NY, in 1989, and the Ph.D. degree from the University of Illinois at Urbana-Champaign in 1992.

From 1992 to 1997, he was an Assistant Professor with the Department of Electrical and Computer Engineering, Iowa State University. He is currently an Associate Professor with the Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis. He has coauthored two books, *Timing Analysis and Optimization of Sequential Circuits* (Norwell, MA: Kluwer, 1999) and *Design Automation for Timing-Driven Layout Synthesis* (Norwell, MA: Kluwer, 1992), and coedited *Layout Optimizations in VLSI Designs* (Norwell, MA: Kluwer, 2001).

Dr. Sapatnekar received the National Science Foundation Career Award and the Best Paper Awards at the Design Automation Conference in 1997 and 2001 and the International Conference on Computer Design in 1998. He is an Associate Editor of the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS and the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS II: ANALOG AND DIGITAL SIGNAL PROCESSING. He has served on the Technical Program Committee for various conferences, including Technical Program and General Chair for Tau and the International Symposium on Physical Design, and is currently a Distinguished Visitor for the IEEE Computer Society and a Distinguished Lecturer for the IEEE Circuits and Systems Society.

A. J. Sullivan received the B.S. degree in electrical engineering at Washington University, St. Louis, MO, in 1990.

Since 1990, he has worked in many areas of design automation and layout with the IBM Corporation, East Fishkill, NY.