



A New Algorithm for Routing Tree Construction with Buffer Insertion and Wire Sizing under Obstacle Constraints *

Xiaoping Tang, Ruiqi Tian[†], Hua Xiang, and D.F. Wong
University of Texas at Austin, Austin, TX 78712

{tang, ruiqi, xiang, wong}@cs.utexas.edu

Abstract

Buffer insertion and wire sizing are critical in deep submicron VLSI design. This paper studies the problem of constructing routing trees with simultaneous buffer insertion and wire sizing in the presence of routing and buffer obstacles. No previous algorithms consider all these factors simultaneously. Previous dynamic programming based algorithm is first extended to solve the problem. However, with the size of routing graph increasing and with wire sizing taken into account, the time and space requirement increases enormously. Then a new approach is proposed to formulate the problem as a series of graph problems. The routing tree solution is obtained by finding shortest paths in a series of graphs. In the new approach, wire sizing can be handled almost without any additional time and space requirement. Moreover, the time and space requirement is only polynomial in terms of the size of routing graph. Our algorithm differs from traditional dynamic programming, and is capable of addressing the problem of inverter insertion and sink polarity. Both theoretical and experimental results show that the graph-based algorithm outperforms the DP-based algorithm by a large margin. We also propose a hierarchical approach to construct routing tree for a large number of sinks.

1. Introduction

Rapid advances in integrated circuit technology have led to a dramatic increase in the complexity of VLSI circuits. As a result of scaling, the devices become much smaller and faster, but interconnects get more resistive. Interconnect delay, especially global interconnect delay, has become the dominant factor in deep submicron design in determining the overall circuit performance and complexity. Many techniques are employed to reduce interconnect delay, such as buffer insertion, wire sizing and routing topology generation. Since buffers are implemented by transistors, they can not be placed over the existing macro blocks. Thus, macro blocks present themselves as routing resources for wires but obstacles for buffers. Moreover, routing congestion areas forbid wires to go through. All these factors make routing one of the most challenging tasks in VLSI design. As transistor count and chip dimension get larger and larger, routing would become a computationally expensive stage in the system design process.

Many buffer insertion and wire sizing algorithms have been proposed in the past few years. Most of them were using dynamic programming in a bottom-up fashion. Van Ginneken [21] presented a polynomial time algorithm based on dynamic programming for optimal delay buffer insertion on a given tree topology. Lillis et al. [13] and Okamoto et al. [15] considered simultaneous routing and buffer insertion by combining P-Tree algorithm and A-Tree algorithm with buffer insertion, respectively. Salek et al. [17] presented an algorithm on routing tree construction for an ordered set of critical sinks. Alpert et al. [2] proposed a two-step approach of steiner tree construction followed by buffer insertion for difficult instances. Proper wire segmenting was discussed for buffer insertion in a routing tree in [1]. These algorithms can efficiently construct a

buffered routing tree, but they can not handle buffer obstacles and routing obstacles. Cong et al. [6] first studied wire sizing based on a dynamic programming algorithm. Sapatnekar [19] considered area and timing constraints. Chen et al. [3] showed the optimal continuous wire shape under Elmore delay model is an exponential function. Kay and Pileggi [10] proposed an efficient greedy approach for wire sizing. Chu and Wong [4] presented a quadratic programming approach to simultaneous buffer insertion and wire sizing for a routing path. Later on, it is extended to combine with dynamic programming for routing tree optimization [14]. However, it is based on a fixed tree topology. Zhou et al. [22] introduced the concept of buffer obstacle and proposed an algorithm using dynamic programming to minimize interconnect delay by simultaneous routing and buffer insertion for 2-pin net with restriction on buffer locations. Most recently, Lai and Wong [12] proposed a graph-based formulation of the Maze Routing problem with buffer insertion and wire sizing. Jagannathan et al. [8] studied the buffer insertion problem to maximize the delay reduction to cost ratio. Both of them can handle constraints of routing obstacles and buffer obstacles. However, they can only deal with 2-pin nets, which limits the application. An algorithm is presented in [7] for simultaneous routing tree construction and buffer insertion for multiple-pin nets under fixed buffer locations. It used dynamic programming approach to construct the routing tree in a bottom-up fashion. The number of buffer locations it handled is small. Furthermore, it didn't consider wire sizing. (Our experiments show that proper wire sizing together with buffer insertion can reduce the delay about 10% further compared to inserting buffers only for a routing tree, and hence is important and necessary.) Although the algorithm in [7] can be extended to deal with the problem of routing tree with buffer insertion and wire sizing under obstacle constraints, the time and space requirement increases enormously with the size of routing graph increasing and with wire sizing taken into account.

In this paper, we present a new approach to formulate the problem as a series of graph problems. We construct a graph for each subset of the sinks. In the graph the source vertex is the subset, and other vertices represent possible buffer choices at different buffer locations. The shortest path from the subset vertex to each other vertex v in the graph represents the optimal routing tree with appropriate buffer insertion and wire sizing which connects v and the subset of sinks. Then a routing tree is created by increasingly considering more sinks. The final solution turns out to be the shortest paths from the subset including all sinks to the source node of the net. Our algorithm differs from traditional dynamic programming approach, and is much more efficient. In the new approach, wire sizing can be handled almost without any additional time and space requirement, i.e. wire library does not affect the complexity. Moreover, the time and space requirement is only polynomial in terms of the size of buffer locations and buffer library. The algorithm is capable of addressing the problem of inverter insertion and sink polarity. Both theoretical and experimental results show that the graph-based algorithm outperforms the DP-based algorithm by a large margin. We also propose a hierarchical approach to construct routing tree for a large number of sinks.

The rest of the paper is organized as follows. Section 2

*This work was partially supported by the National Science Foundation under grant CCR-9912390.

[†]Ruiqi Tian is also with Motorola Inc., Austin, TX 78721.

reviews the delay model and formally defines the problem of routing tree with buffer insertion and wire sizing under obstacle constraints. In section 3, the previous dynamic programming algorithm is extended to solve the problem, and its complexity and limitation are briefly described. Section 4 gives our new algorithm and analyzes its time and space complexity. We also present a hierarchical approach to handle large number of sinks in section 5. Several capabilities of extension are discussed in section 6. Finally, we show the experimental results with concluding remarks in Section 7.

2. Delay Model and Problem Formulation

In this section, we briefly review the delay model and formally define the problem of constructing a routing tree with buffer insertion and wire sizing under obstacle constraints.

2.1 Delay Model

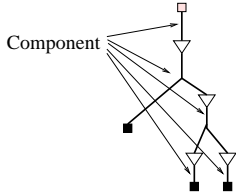


Figure 1: Components decomposed by buffers (source, or sink) in a routing tree.

As in most previous works, we use the π -RC model for wires and switch-level RC model for buffers. In this paper, we adopt the distributed RC delay model developed by Rubinstein et al. [16] for delays. Given a distributed RC circuit, the signal delay at a node is computed as follows:

$$d = \sum_{\text{all nodes } k} R_k^* \cdot C_k^*$$

where R_k^* is the resistance of node k and C_k^* is the downstream capacitance for all branches from node k . Each buffered routing tree is a set of **components** which are decomposed by buffers in the tree. Each component is a tree or a path without any buffer inside. Fig 1 illustrates the components of a routing tree. The delay of a component is computed by Rubinstein model. The total delay from the source to a sink in the buffered routing tree is the sum of the component delays and buffer delays along the path from the source to the sink.

The reasons that we choose this delay model are: (i) Although the model is simpler than Elmore delay model, it still captures the property of distributed circuit, and gives the same delay as Elmore delay model for a routing path; (ii) It gives a uniform upper bound of the delay for every receiver node in a component; (iii) It implicitly encourages more sharing of routing resources than Elmore delay model.

2.2 Problem Formulation

The goal of our algorithm is to construct a routing tree with feasible buffer insertion and proper wire sizing in the presence of obstacles, such that the maximum delay from the source to sinks is minimized. Without buffer obstacles, the optimal routing tree is a shortest path tree rooted at the source node. But in the presence of macro blocks, which are available for wiring but infeasible for buffer insertion, the shortest path tree does not guarantee to be the optimal routing tree with minimized maximum delay from the source to sinks. Previous algorithms which ignore macro blocks during routing tree construction may result in a much inferior solution.

We illustrate the importance of the problem by an example of the 3-pin net in Fig 2. Shaded boxes represent routing obstacle regions where wiring is not allowed, and gray boxes

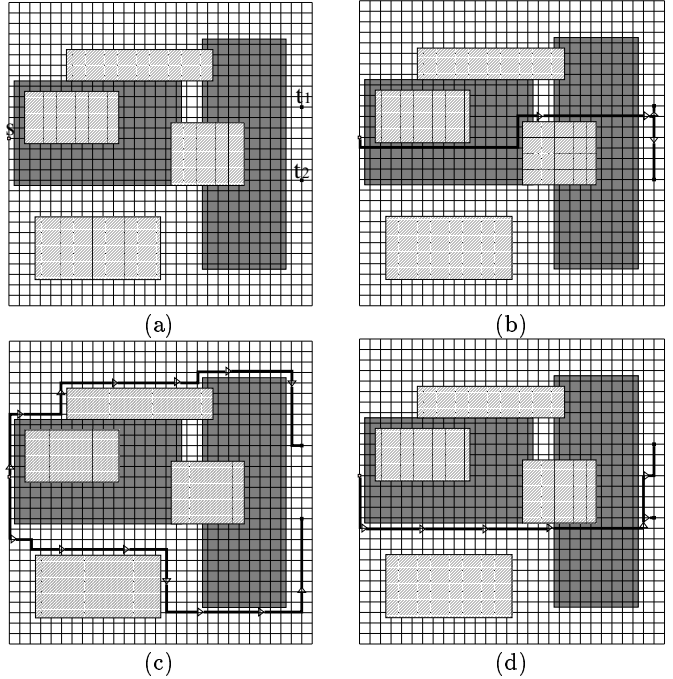


Figure 2: (a) Routing tree problem (b) Shortest path tree + buffer insertion, delay = 1183ps (c) Avoid obstacles + buffer insertion, delay = 1220ps (d) Our routing tree approach, delay = 977ps.

represent macro blocks where buffering is not allowed. The routing area has the unit length of 0.5mm. The technology parameters are taken from [20] with unit length wire resistance $0.076\Omega/\mu m$, unit length capacitance $0.118fF/\mu m$, driver resistance of source and output resistance of buffer 180Ω , load capacitance of sink and input capacitance of buffer $23.4fF$, and intrinsic delay for buffer $36.4ps$. An optimal shortest path tree is shown in Fig 2(b), and its delay is 1183ps with proper buffer insertion. The shortest path tree has a long part going over macro blocks, which forbid buffer insertion. As a result, it produces a large delay. An alternative approach is to regard macro blocks as routing obstacles and avoid wiring over them. An optimal solution by this approach is shown in Fig 2(c), and its delay is 1220ps. Although buffers can be inserted anywhere, it wastes routing resource and may be too long to have short delay. By wisely routing with proper buffer insertions, we can obtain a better solution which is shown in Fig 2(d) with delay 977ps.

In the problem of routing tree construction, the routing area is represented by a grid graph $G = (V, E)$. Routing obstacles and buffer obstacles are present in the routing area. Each edge in E is a wire segment. Wire library W provides wire choices, and buffer library B contains buffer choices to be inserted at grid nodes in G . There are four types of nodes in $G(V, E)$:

1. Source node s : grid node representing the source of a net
2. Sink node $t_i, i = 1, \dots, k$: grid node representing a sink of a net
3. Buffer node f : grid node which allows buffer to be inserted
4. Wire node r : grid node which forbids buffering but allows wiring.

In the rest of the paper, we use F to denote the set of buffer nodes in G , and R to denote the set of wire nodes. The problem of routing tree with buffer insertion and wire sizing under obstacle constraints is formally described as follows:

Problem 1. RTBW Given a routing graph $G = (V, E)$, a wire library W , a buffer library B , a source node $s \in V$ and k sink nodes $t_1, t_2, \dots, t_k \in V$ of a net, find a buffered routing tree T rooted at s and leafed at t_i , $i = 1, \dots, k$, for each node $v \in T$, $b(v) \in B \cup \{0\}$ where $b(v) = 0$ indicates no buffer is inserted at v and $b(v) \neq 0$ requires v is a buffer node, for each segment $l \in T$ wire $w(l) \in W$, such that the maximum delay from s to t_i is minimized.

3. Previous Approach by DP

Dynamic programming is a popular method for various buffer insertion and wire sizing problems [21, 17, 15, 22, 7]. A dynamic programming algorithm solves every subproblem and saves the partial solution in a table, then computes an optimal solution in a bottom-up fashion. In this section, we extend the algorithm in [7] (called DP-RTBW) to solve RTBW problem, and analyze its complexity and limitation.

Each node is labeled by a set of tuples (C, D, Γ) representing a subtree rooted at the node where C is the downstream capacitance, D is the delay of the subtree; and Γ is the sink set of the subtree. Two tuples, belonging to two adjacent nodes in the routing graph, are merged by a wire connection to create a new tuple if the two tuples have disjoint subsets of sinks. A priority queue Q is constructed to sort the partial solutions on the key D , and initialized with the union of all label sets of sinks. A tuple (C_1, D_1, Γ_1) is redundant with respect to another (C_2, D_2, Γ_2) if $\Gamma_1 = \Gamma_2$, $C_1 \geq C_2$, $D_1 \geq D_2$, which can be discarded safely. The pseudocode is omitted.

The time and space complexity of DP-RTBW is very large. Let C_{max} denote the max value of all downstream capacitance. The number of sink subsets is 2^k . Thus each node could have $O(2^k C_{max})$ tuples. Since there are V nodes in routing graph, the total number of tuples is $O(2^k V C_{max})$. Each node has at most 4 adjacent nodes in routing graph. Thus a tuple at a node will merge with $O(2^k C_{max})$ tuples at other nodes, each costing $O(W)$. Each time inserting a new tuple to the priority queue takes $O(\log(2^k V C_{max}))$, and pruning can be amortized. Therefore, by multiplying the number of tuples, merging cost and inserting cost, we get a loose time bound: $O(4^k W V C_{max}^2 \log(2^k V C_{max}))$, where k is the number of sinks, W is the number of wire choices, and C_{max} is the maximum value of downstream capacitance. A loose space complexity bound is $O(2^k V C_{max})$. C_{max} is usually very large, even huge to make the algorithm impractical¹. By carefully examining the possible capacitance combinations at each node, we can get a much more accurate bound of complexity.

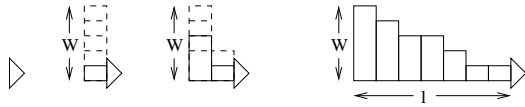


Figure 3: Illustration of a wire path with length l and W wire choices.

Let us consider the wire path with length l and W choices as in Fig 3. We know the wire width must be decreasing along the path (otherwise the tuple label will be redundant and pruned). Then the number of different non-redundant combination is C_{l+W-1}^{W-1} . Since there are B different buffer choices, the number of capacitance is $C_{l+W-1}^{W-1} B$. Let L be the distance from a grid node v to a sink. If we assume each node between v and the sink can be a buffer location (which is possible), then the node v can have $\sum_{l=0}^L C_{l+W-1}^{W-1} B = C_{L+W}^W B$ tuples related to the sink. When the number of

¹These bounds depend on C_{max} . It suggests that we use a rough metric for capacitance, thus we should reduce the time and space requirements. But it will greatly affect the optimality and quality.

sinks is k , roughly, a node v can have $O(\prod_{i=1}^k (C_{L+W}^W B)) = O(L^{Wk} B^k)$ capacitance combinations or tuples. Thus, the space requirement is $O(V L^{Wk} B^k)$, and the time requirement is $O(V L^{Wk} B^k \cdot W L^{Wk} B^k \cdot \log(V L^{Wk} B^k)) = O(V W L^{2Wk} B^{2k} \log(V L^{Wk} B^k))$. In routing graph, $L = O(\sqrt{V})$. Hence, the space complexity is $O(V V^{Wk/2} B^k)$ and time complexity is $O(W V^{Wk+1} B^{2k} \log(V^{Wk/2+1} B^k))$. With the number of sinks increasing, the complexity increases enormously.

4. New Graph-Based Algorithm

In this section, we describe a new graph-based algorithm, and analyze its complexity. At first, we summarize the notations. Without ambiguity, we also use a set X to represent its cardinality (i.e. $|X|$).

4.1 Notation

- W : wire library of different wire choices.
- B : buffer library of different buffer choices.
- F : the set of nodes which allows buffer to be inserted (also called buffer node).
- $N = \{s\} \cup \{t_i | i = 1, \dots, k\} \cup F$: the set consisting of the source, the sinks and the possible buffer nodes. Clearly, $|N| \leq |V|$.
- Γ : the non-empty subset of sinks.
- **Wire Path**: a path connecting two nodes in N by properly sized wires but no buffer between (refer to Fig 4).
- **Buffered Path**: a path connecting two nodes in N with buffers inserted between. A wire path is a special buffered path. Without ambiguity, we also call a buffered path as a path.
- **Buffer Combination**: a tree component connecting three or more nodes in N without internal buffers. For convenience, we call the upstream node in a buffer combination as **driver**, and the other node as **receiver** (refer to Fig 6).
- **Subtree**: a tree connecting a buffer node or the source to a subset of sinks.
- **BC-Subtree**: a subtree beginning with a buffer combination. A BC-subtree is a special subtree.
- **Wire Path Table Ψ** : a table storing pre-computed optimal buffer-to-buffer wiresizing solutions for wire paths.
- **Buffer-Combination Table Φ** : a table storing pre-computed optimal wiresizing solutions for buffer combinations.
- **Buffered Path Table \tilde{P}** : a table storing pre-computed optimal solutions for buffered path.
- S_a : the space size of wire path table Ψ and buffer-combination table Φ .
- T_a : the runtime to compute Ψ and Φ .

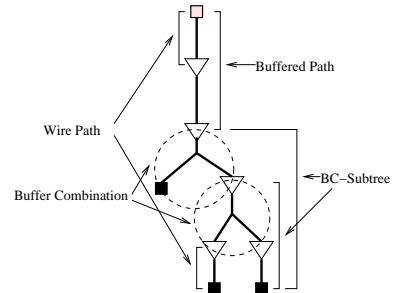


Figure 4: Illustration of notations.

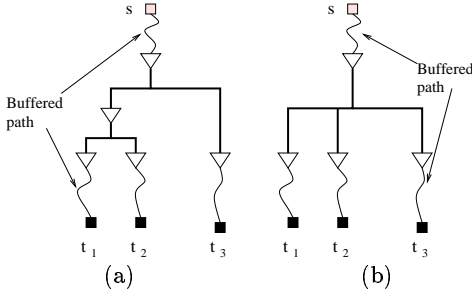


Figure 5: Two configurations of a routing tree with 3 sinks.

4.2 General Idea

Notice that a buffered routing tree is a set of buffer combinations plus the connecting buffered paths as shown in Fig 5. A buffered path is a set of connected wire paths. Both wire path and buffer combination can be pre-computed.

Given a routing grid graph $G = (V, E)$, let $N = \{s\} \cup \{t_i | i = 1, \dots, k\} \cup F$. Clearly, $|N| \leq |V|$. For a grid graph G with a lot of buffer obstacles, $|N|$ is much less than $|V|$. For any $u, v \in N$, the minimum delay of a wire path from u to v (denoted as $d(u, v)$) in G , the driver resistance in u , the load capacitance in v , and the wire library W . Since the buffer library B is given, the possible driver resistance and load capacitance are known. Buffer-to-buffer wiresizing solutions can be pre-computed and stored in a table for wire path (denoted as Ψ). The minimum delay value from u with buffer b_u to v with buffer b_v can be found by looking up the wire path table $\Psi(b_u, b_v, d(u, v))$. It should be noted that we assume the driver resistance of the source equals one of the resistances of buffers in B and the load capacitance of sinks equals one of the capacitance of buffers. If not, it can be solved easily by adding an additional buffer type with source resistance and sink capacitance into B and letting the buffer type be used at source and sinks only.

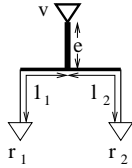


Figure 6: A buffer combination with 3 nodes (v, r_1, r_2).

The delay of a buffer combination (v, r_1, \dots, r_t), where v is driver and r_1, \dots, r_t are receivers, is more complicated to compute. However, it is still a function of the distance configuration of the buffer combination, the driver resistance of v , the load capacitances of $r_i, i = 1, \dots, t$, and the wire library W . As an example, Fig 6 shows a buffer combination of 3 nodes (v, r_1, r_2). Its minimum delay is determined by the stem length e , branch length l_1, l_2 , the driver resistance of v , the load capacitances of r_1, r_2 , and the wire library W . In practice, the degree of a buffer combination ($= t + 1$) is small (otherwise, it will cause large delay). If we restrict the maximum degree of a buffer combination, the distance configuration (e.g. e, l_1, l_2 in Fig 6) can be obtained by computing a steiner tree of a small number of nodes. Thus we can still build a pre-computed table (denoted as Φ) to store the minimum delay for possible buffer combinations.

After getting wire path table Ψ , we can compute the optimal buffered path between any two nodes. As in [12] a buffer node is splitted into B nodes, we construct a BP-Graph $G_{BG} = (V_{BG}, E_{BG})$:

$$V_{BG} = \{s\} \cup \{t_i | i = 1, \dots, k\} \cup \{f_b | b = 1, \dots, B, f \in F\}$$

$$E_{BG} = \{(u, v) | u, v \text{ are not the same node in } V\}$$

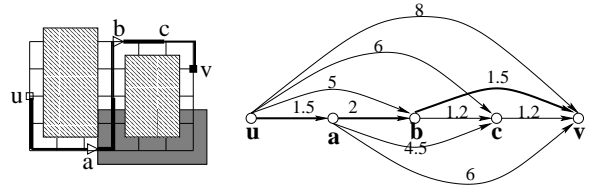


Figure 7: A shortest path between u and v represents an optimal buffered path (no buffer at location c).

Clearly, $|V_{BG}| = O(BN)$. The edge $(u, v) \in E_{BG}$ is an optimal wiresized path, and its weight $W(u, v)$ is obtained by looking up the table $\Psi(b_u, b_v, d(u, v))$. Therefore, a shortest path between any pair in V_{BG} is an optimal buffered path connecting the two nodes of the pair (see Fig 7). Then, we apply all-pair shortest path algorithm on G_{BG} , and store the values in buffered path table of the size $O(BN \times BN)$ (denoted as \tilde{P}). Thus, the minimum delay of any pair in V_{BG} , which is determined by the optimal buffered path, can be found by looking up the table \tilde{P} .

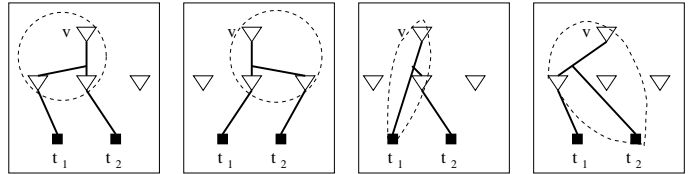


Figure 8: 4 BC-subtrees rooted at v connecting t_1, t_2 (other BC-subtrees are omitted).

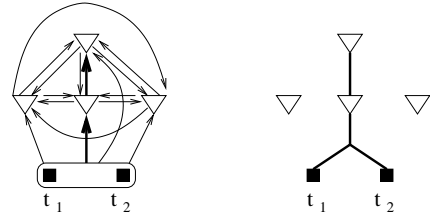


Figure 9: A shortest path represents an optimal subtree.

A BC-subtree connects a node with a subset of sinks, beginning with a buffer combination rooted at the node. Let us consider BC-subtree rooted at v connecting 2 sinks: t_1, t_2 (see Fig 8). If we enumerate all possible buffer combinations rooted at v (v be driver), we will get all possible BC-subtrees. Fig 8 illustrates 4 of the BC-subtrees. Since the delay of buffered path and buffer combination is known by looking up table, we can get the delay of BC-subtrees immediately. Then the best BC-subtree is picked up for v . In this way, we get the optimal BC-subtrees for every node. Note that the optimal BC-subtree rooted at v may not be the optimal subtree for v . Then we construct a graph called G_Γ as shown in Fig 9. In the graph the source vertex is $\Gamma = \{t_1, t_2\}$, and other vertices are possible buffer nodes. The edge (Γ, v) represents the optimal BC-subtree rooted at v , and its weight is the maximum delay of the BC-subtree. The edge $(u, v), u, v \neq \Gamma$ represents the optimal buffered path between u and v , which can be looked up in table \tilde{P} . Then the shortest path from Γ to each other vertex v corresponds to the optimal subtree connecting to t_1 and t_2 . If we apply single-source shortest path algorithm (such as Dijkstra's algorithm), we can get all optimal subtrees for every node. The algorithm proceeds to create subtrees by increasingly considering more sinks. The final routing tree turns

out to be the shortest paths from the subset including all sinks to the source node of the net.

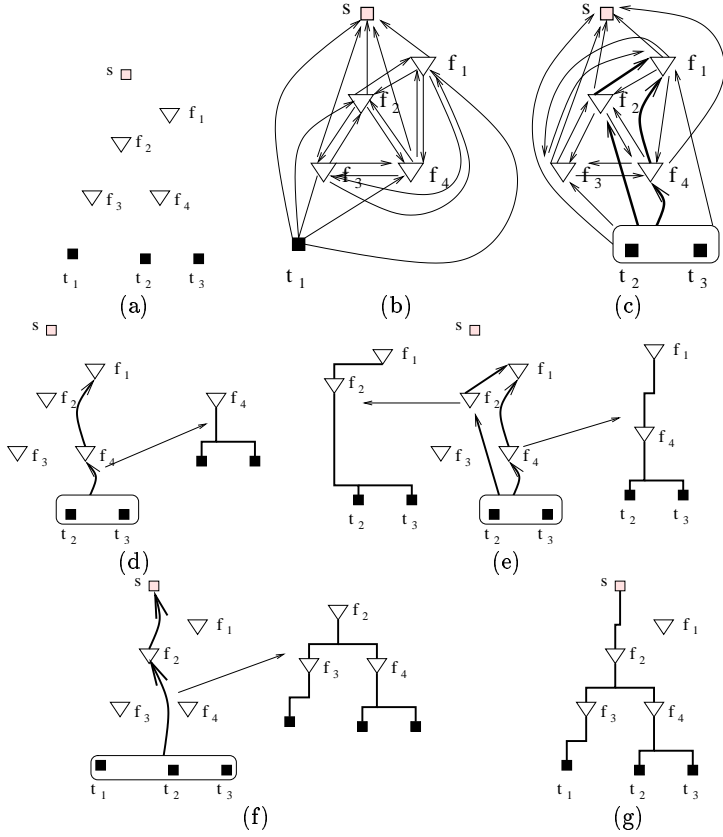


Figure 10: Illustration of G_Γ computations. (a) A routing problem with 3 sinks and 4 buffer locations. (b) G_{t_1} , every edge is a buffered path. (c) G_{Γ_1} , $\Gamma_1 = \{t_2, t_3\}$, the edges adjacent to Γ_1 are BC-subtrees and other edges are buffered paths. (d) Part of G_{Γ_1} , the edge (Γ_1, f_4) represents a BC-subtree, and the edge (f_4, f_1) is a buffered path. (e) Part of G_{Γ_1} , two paths from Γ_1 to f_1 represent two subtrees, respectively. (f) Part of G_{Γ_2} , $\Gamma_2 = \{t_1, t_2, t_3\}$, the edge (Γ_2, f_2) represents a BC-subtree. (g) Routing tree solution.

4.3 New Algorithm

Let Γ be a non-empty subset of all sinks. Then the number of Γ 's is $2^k - 1$. We construct a series of $2^k - 1$ graphs, each corresponding to a Γ . Let \tilde{F} denote the set $\{f_b | b = 1, \dots, B, f \in F\}$. Each graph $G_\Gamma = (V_\Gamma, E_\Gamma)$ is constructed as:

$$\begin{aligned} V_\Gamma &= \{s\} \cup \{\Gamma\} \cup \tilde{F} \\ E_\Gamma &= \{(\Gamma, s)\} \cup \{(\Gamma, v) | v \in \tilde{F}\} \cup \\ &\quad \{(u, v) | u, v \in \{s\} \cup \tilde{F} \text{ and } u \neq v \text{ in } V\} \end{aligned}$$

The edge $(u, v) \in E_\Gamma$, $u \neq \Gamma$, represents a buffered path which connects v with u , thus, its weight $W(u, v)$ can be found in table \tilde{P} . The edge (Γ, v) , $v \in \{s\} \cup \tilde{F}$, represents an optimal BC-subtree rooted at v and containing the sinks of Γ . The weight $W(\Gamma, v)$ is determined as:

$$W(\Gamma, v) \leftarrow \min_{\phi} (W(\phi) + \max_{p_1, \dots, p_i} (D_{p_1}(p_1, r_1), \dots, D_{p_i}(p_i, r_i)))$$

where $\phi = (v, r_1, \dots, r_i)$ is buffer combination, p_1, \dots, p_i is a disjoint non-empty partition of Γ (p_j is a true subset of Γ), $W(\phi)$ is the delay value found in the buffer combination table

Φ , and $D_p(p, v)$ denotes the delay value of the shortest path from p to v in the graph G_p (p is a subset of all sinks).

Then we apply single-source shortest path algorithm to the graph G_Γ sourced at Γ , and compute the delay value of the shortest path from Γ to each v in G_Γ . Therefore, a shortest path from Γ to v corresponds to an optimal subtree rooted at v and containing Γ .

Since the construction of G_Γ depends on G_p where p is a proper subset of Γ , we construct and compute the graphs G_Γ in increasing order of the degree of Γ ($|\Gamma|$). $G_{sinkset}$ is the last graph. The final solution is the value of $D_{sinkset}(sinkset, s)$. By storing the back-up information at each node, we can construct the whole buffered routing tree.

Fig 10 shows part of G_Γ constructions. Fig 10(a) illustrates a routing tree problem. Fig 10(b) and (c) illustrates G_{t_1} and G_{Γ_1} , $\Gamma_1 = \{t_2, t_3\}$ respectively. Fig 10(d) illustrates part of G_{Γ_1} , $\Gamma_1 = \{t_2, t_3\}$. The edge (Γ_1, f_4) represents an optimal BC-subtree beginning with the buffer combination (f_4, t_2, t_3) . Then, the shortest path from Γ_1 to f_4 is an optimal subtree rooted at f_4 and containing Γ_1 , and the shortest path from Γ_1 to f_1 is an optimal subtree rooted at f_1 . Fig 10(e) illustrates two paths from Γ_1 to f_1 , each representing a subtree rooted at f_1 and containing Γ_1 . Fig 10(f) illustrates part of G_{Γ_2} , $\Gamma_2 = \{t_1, t_2, t_3\}$. The edge (Γ_2, f_2) is an optimal BC-subtree beginning with the buffer combination (f_2, f_3, f_4) , where the partition of Γ_2 is $p_1 = \{t_1\}, p_2 = \{t_2, t_3\}$. Part of the BC-subtree, rooted at f_4 , is referred to G_{Γ_1} as the shortest path (a subtree) from Γ_1 to f_4 (The other part of the BC-subtree is the shortest path from t_1 to t_3 in G_{t_1}). Fig 10(g) shows the final routing tree solution. As we can see, every subtree of the routing tree corresponds to a shortest path in one of the G_Γ graphs.

Since all information in G_Γ , $|\Gamma| = 1$ (single sink) is stored in the graph G_{BG} and table \tilde{P} , these graphs G_Γ are not needed to be constructed and computed. We only need to start with $|\Gamma| = 2$.

Every graph G_Γ is a fully-connected graph with the same configuration (the same set of vertices except Γ and the same set of edges except (Γ, v)). The edge weight $W(u, v)$, $(u, v) \in E_\Gamma$ and $u \neq \Gamma$, is in table \tilde{P} . Actually, we don't need to store these edges. The weight $W(\Gamma, v)$, $v \in \{s\} \cup \tilde{F}$, can be stored at the same place with $D_\Gamma(\Gamma, v)$. By carefully constructing the graph, we only need $\Theta(V_\Gamma) = O(BN)$ space to represent both the graph and the results.

The pseudocode of the algorithm is shown in Fig 11.

Algorithm GRAPH-RTBW

```

Compute all-pair shortest distance  $d(u, v)$  in  $G$ 
Compute wire path table  $\Psi$  and buffer combination table  $\Phi$ 
Construct the graph  $G_{BG}$ 
Compute all-pair shortest path in  $G_{BG}$ 
and store in buffer path table  $\tilde{P}(u, v)$ 
FOR  $i = 2$  TO  $k$  DO
  FOR each subset  $\Gamma, |\Gamma| = i$  DO
    FOR  $v$  in  $G_\Gamma$  DO
       $W(\Gamma, v) \leftarrow \infty$ 
    FOR  $j = 2$  TO  $t$  DO //  $t$ : combination threshold
      FOR each  $\phi = (v, r_1, \dots, r_j)$  DO
        FOR each  $\Gamma, |\Gamma| = i$  DO
          FOR each disjoint non-empty partition  $p_1, \dots, p_j$  of  $\Gamma$ 
            DO
               $W(\Gamma, v) \leftarrow \min(W(\Gamma, v),$ 
                 $W(\phi) + \max(D_{p_1}(p_1, r_1), \dots, D_{p_j}(p_j, r_j)))$ 
          FOR each subset  $\Gamma, |\Gamma| = i$  DO
            Apply Dijkstra's algorithm on  $G_\Gamma$  with source  $\Gamma$ 
            Store  $D_\Gamma(\Gamma, v)$ 
  RETURN  $D_{sinkset}(sinkset, s)$ 

```

Figure 11: Graph-based algorithm for RTBW problem

4.4 Complexity

Let S_a denote the space size of wire path table Ψ and buffer combination table Φ , and T_a denote the runtime to compute

the two tables. The space needed for $d(u, v)$ in grid $G = (V, E)$ is $O(V^2)$. $\tilde{P}(u, v)$ and graph G_{BG} require $O(B^2 N^2)$ space. As mentioned above, each G_Γ and $D_\Gamma(\Gamma, v)$ requires only $O(BN)$ space to represent. All of the graphs G_Γ and $D_\Gamma(\Gamma, v)$ need $O(2^k BN)$ space in total. The following theorem summarizes the space requirement of Algorithm GRAPH-RTBW:

Theorem 1. *The space complexity of Algorithm GRAPH-RTBW is $O(S_a + V^2 + B^2 N^2 + 2^k BN)$ for a given routing grid graph $G = (V, E)$, where k is the number of sinks, B is the buffer choices in the buffer library, and N is the set $\{s\} \cup \{t_i | i = 1, \dots, k\} \cup F$.*

Computing $d(u, v)$ in G can be done in $O(V^2 \log V)$ time. It takes $O(B^3 N^3)$ time to construct G_{BG} and compute $\tilde{P}(u, v)$ in G_{BG} . Dijkstra algorithm on each G_Γ runs in $O(V_\Gamma \log V_\Gamma + E_\Gamma) = O(B^2 N^2)$ time. The time required by Dijkstra algorithm for all G_Γ s is $O(2^k B^2 N^2)$. The number of different $\Gamma, |\Gamma| = i$ from k sinks is C_k^i . The number of different partition p_1, \dots, p_j of $\Gamma, |\Gamma| = i$, is $O(j^i)$. Let us compute the number of buffer combinations $\phi = (v, r_1, \dots, r_j)$ for each graph G_Γ . Choosing $j + 1$ nodes from G_Γ , the number of the combinations is $C_{V_\Gamma}^{j+1} = O(C_{BN}^{j+1})$. Since each node of the $j + 1$ nodes can be the driver, the number of buffer combinations is $O((j + 1)C_{BN}^{j+1})$. Note that, the order of receivers r_1, \dots, r_j does not need to be counted here because partition p_1, \dots, p_j is ordered. Thus, given that the degree of buffer combination is restricted to be $\leq t + 1$ (t be threshold), the time required to compute all G_Γ s is:

$$\begin{aligned} & \sum_{i=2}^k \sum_{j=2}^t j^i C_k^i (j+1) C_{BN}^{j+1} \\ &= \sum_{j=2}^t \sum_{i=j}^k j^i C_k^i (j+1) C_{BN}^{j+1} \\ &\leq \sum_{j=2}^t \sum_{i=0}^k j^i C_k^i (j+1) C_{BN}^{j+1} \\ &= \sum_{j=2}^t (1+j)^{k+1} C_{BN}^{j+1} \\ &\leq c \cdot (t+1)^k B^{t+1} N^{t+1} \end{aligned}$$

Note that the runtime required for Dijkstra algorithm is $O(2^k B^2 N^2) = o((t+1)^k B^{t+1} N^{t+1})$ ($t \geq 2$). Thus it can be omitted safely. The following theorem describes the time complexity of Algorithm GRAPH-RTBW.

Theorem 2. *The runtime of Algorithm GRAPH-RTBW is $O(T_a + V^2 \log V + B^3 N^3 + (t+1)^k B^{t+1} N^{t+1})$ for a given routing grid graph $G = (V, E)$, where t is the threshold on the degree of buffer combination, k is the number of sinks, B is the buffer choices in the buffer library, and N is the set $\{s\} \cup \{t_i | i = 1, \dots, k\} \cup F$.*

Especially, if we restrict the routing tree to be a binary tree (i.e. $t = 2$), which is the common case and gives optimal delay in practice, we will get the following time bound.

The number of non-empty 2-partition p_1, p_2 of $\Gamma, |\Gamma| = i$, is $2^i - 2$. Thus, the total time to compute all G_Γ s is:

$$\begin{aligned} & \sum_{i=2}^k (2^i - 2) C_k^i 3 C_{BN}^3 \\ &\leq (BN)^3 \sum_{i=2}^k (2^i - 2) C_k^i \\ &= (BN)^3 \left(\sum_{j=0}^k (2^j - 2) C_k^j + 1 \right) \\ &= (BN)^3 (3^k - 2^{k+1} + 1) \end{aligned}$$

Theorem 3. *If the routing tree is a binary tree, the runtime of Algorithm GRAPH-RTBW is $O(T_a + V^2 \log V + (3^k - 2^{k+1} + 2) B^3 N^3)$ for a given routing grid graph $G = (V, E)$, k is the number of sinks, B is the buffer choices in the buffer library, and N is the set $\{s\} \cup \{t_i | i = 1, \dots, k\} \cup F$.*

The complexity of GRAPH-RTBW is much better than that of DP-RTBW (space: $O(S_a + V^2 + B^2 N^2 + 2^k BN)$ to $O(VV^{W^{k/2} B^k})$, time: $O(T_a + V^2 \log V + (3^k - 2^{k+1} + 2) B^3 N^3)$ to $O(WV^{W^{k+1} B^{2k} \log(V^{W^{k/2+1} B^k}))$). Notably, the complexity of GRAPH-RTBW is mainly dependent on the number of sinks and the cardinality of N , while DP-RTBW is related to the size of routing graph $G = (V, E)$. In the applications where more buffer obstacles exist, i.e., N is much less than V , we will get much more benefit by using GRAPH-RTBW algorithm.

5. Hierarchical Approach

As we can see, the time and space complexity of GRAPH-RTBW increase exponentially in terms of the number of sinks. In some routing tree problems where the number of sinks is large, the performance of GRAPH-RTBW will degrade considerably; even some can not be handled. In this case, we can consider a hierarchical approach. The hierarchical approach has two aspects: two-phase routing and clustering.

5.1 Two-phase Routing

First, we use a rough grid graph, the sinks near a grid node are grouped into one new sink. By this way, we could reduce not only the number of sinks but also the number of $|BN|$. Then, the GRAPH-RTBW algorithm is applied to the rough grid graph to get a rough routing tree. After that, we can use a fine grid on the rough routing tree obtained, and compute a new fine routing tree again (refer to Fig 12). Note that unused grid nodes in rough grid do not appear in the fine grid, which will reduce the size of the fine grid.

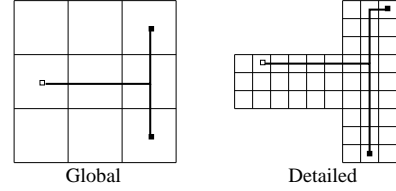


Figure 12: Two-phase routing approach.

5.2 Clustering

Clustering is useful in “divide and conquer” strategy. Organizing sinks into sensible groupings not only catches the idea of hierarchical design, but also reduces the complexity of our algorithm. Many clustering algorithms are available in [9]. We use Zahn’s clustering algorithm, because it is efficient and suitable to our problem.

1. Construct an MST (minimum spanning tree) for the set of sinks.
2. Remove the largest edges in the MST (consider balance between connected components as well).
3. Identify clusters. (Every connected component forms a cluster)

If we remove $n - 1$ edges in MST, we will get n clusters. Each cluster could be further divided into several subclusters. The connected component is the MST for the cluster, so it can be re-used. The process can proceed in multi-level.

Then we regard the center of a cluster as a new sink. Thus we build a buffered routing tree to connect the source to these

new sinks. Note that the center of a cluster may not be a good position to locate a new source for the sinks in the cluster. Instead, we use the buffer in the routing tree close to the bounding box of the cluster to be the new source. Then a new routing tree is to be built to connect the new source to the sinks in the cluster. The process can go recursively in multi-level. Each level, we can specify a threshold for the number of clusters. As an example, Fig 13 shows that the MST of all sinks is decomposed into 3 clusters. Fig 14 shows the routing tree connecting the source to the centers of all clusters. The buffer close to the bounding box of a cluster is regarded as the new source for the sinks in the clusters, which forms a subproblem.

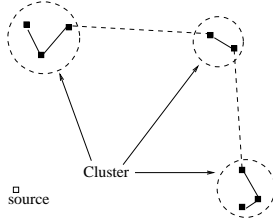


Figure 13: 3 clusters of 8 sinks. Dashed lines are removed edges in the MST.

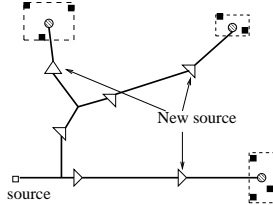


Figure 14: A routing tree connecting the source to the centers of all clusters. The buffer close to the bounding box of a cluster acts as a new source.

6. Discussion

In this section, we discuss some possible extensions and improvements of our algorithm.

6.1 Pruning

In practice, a buffer combination (component of a buffered tree) with a large number of branches will result in a large delay. As mentioned above, a threshold can be put on the degree of buffer combination in the algorithm. Thus, the runtime of the algorithm can be greatly reduced.

In the algorithm described above, we enumerate all combinations of the nodes in G_{Γ} , which is not necessary. In practice, we know a buffer must be inserted inside if the interconnect length is beyond a certain value. This specific value is technology dependent, which can be used as a threshold for enumerating buffer combinations. A buffer combination could be left out if the distance from its driver to one of its receivers is greater than the threshold. This pruning technique can speed up the algorithm considerably.

6.2 Inverter

Inverters consume less resource than normal buffers, and are used as an alternative of buffers in many applications. One simple way is to use two cascaded inverters as a single buffer. Thus our algorithm can still apply without any modification. However, it is beneficial not to use cascaded inverters but rather to maintain an even number of inverters along every source-sink path. With inverter taken into account, the GRAPH-RTBW algorithm can be modified as follows. Each

buffer node in G_{BG} and $G_{\Gamma S}$ is splitted into two nodes further, one representing a node with even number of inverters from source, the other representing a node with odd number of inverters. The source is labeled as odd, and all sinks are labeled as even. For an edge $(u, v) \in E_{BG}$ of G_{BG} , u and v must have different labels of even or odd. We can still build a table of buffered path for looking up at the construction and computation of G_{Γ} . During the generation of buffer combination $\phi = (v, r_1, \dots, r_j)$, the driver v must have the different label of even or odd from those of all receivers r_1, \dots, r_j .

6.3 Sink Polarity

Sinks may have different polarities (positive/negative) [2]. A sink with negative ('-') polarity needs an additional inverter to get the right signal. With polarity constraint taken into account, we can easily modify GRAPH-RTBW for inverter insertion as follows. We label the sink with positive ('+') polarity as even, and the sink with negative ('-') polarity as odd. Each wire path and buffer combination connects nodes with different labels (even/odd). We can still compute the buffered path between any pair of nodes, with even/odd label alternatively along the path. Thus, our GRAPH-RTBW algorithm for inverter insertion is still applicable to construct a routing tree with polarity constraint.

7. Experimental Results

We have implemented both DP-RTBW and GRAPH-RTBW algorithms in C language, and tested them on a Pentium II(400MHz) machine with 256M memory. We use the technology parameters in [20]. The driver resistance of source is set to be 180Ω , and the load capacitance of sink $23.4fF$ respectively. One kind of buffers is used in the program with input capacitance of $23.4fF$ and output resistance of 180Ω . The intrinsic delay of buffer is set to be $36.4ps$. The resistance R_w and capacitance C_w for a wire with width x and length l are given by $R_w = r_w l/x$ and $C_w = c_a x l + c_f l$ respectively, where r_w is unit resistance, c_a is up-down wire capacitance per unit area, and c_f is fringing wire capacitance per unit length. We use the parameters: $r_w = 0.076\Omega/\mu m$, $c_a = 0.024fF/\mu m$ and $c_f = 0.094fF/\mu m$. There are three choices of wire in our wire library: $\{x = 1, 2, 3\}$.

We first compared the solutions with wiresizing and buffer insertion to those without wiresizing. We used test cases with 1-8 sinks and 4-6 obstacles. Table 1 lists the comparison results. The grid used for routing has unit length of $0.5mm$. In non-wiresizing case, we used the wires with width $x = 1$. It shows that on average, proper wire sizing together with buffer insertion can reduce the delay up to 10% further compared to inserting buffers only for a routing tree while the runtime is almost the same. Compared to non-wiresizing runtime, wiresizing takes only additional time to compute table Ψ and Φ , which can be done very fast (less than 1 second). This explains why the runtime is almost the same. As we notice, the wirearea in wiresizing increases by a ratio of 2 compared to that in non-wiresizing, but the number of buffers decrease. Therefore, wiresizing can reduce the number of buffers inserted.

We also compared our GRAPH-RTBW algorithm with the dynamic programming based DP-RTBW algorithm. Table 2 shows the runtime and memory requirement for test cases T1-T8. N/A means the result is not available due to huge runtime and memory requirement. The number of buffer locations in the test cases is up to 500, which is significantly greater than that in [7] (30). Our GRAPH-RTBW outperforms DP-RTBW in all test cases. With the number of sinks increasing, the performance of DP-RTBW degrades very fast, while GRAPH-RTBW scales reasonably well. In the case that the routing region has more buffer obstacles, the GRAPH-RTBW algorithm can benefit from the reduced number of buffer nodes. For example, the test case T7 has more buffer obstacles, and thus results in less runtime than T6 even though T6 has less sinks.

Table 1: Comparison of buffered routing trees with and without wiresizing. (delay imp. means delay improvement)

Data			non-wiresizing				wiresizing				delay imp.
name	pin	chipsize (mm^2)	delay (ps)	wirearea	buffer	time (s)	delay (ps)	wirearea	buffer	time (s)	
T1	2	11.5x14	1080.2	45	6	9.7	979.41	90	4	9.8	9.3%
T2	3	12.5x11.5	1010.2	42	6	7.5	918.73	87	4	7.5	9.1%
T3	4	12.5x12	1059.7	46	7	15.7	962.33	96	5	15.7	9.2%
T4	5	15x13	1095.3	69	12	57.8	998.45	140	9	57.9	8.9%
T5	6	12.5x14	1109.9	77	12	131.6	1008.1	170	12	131.9	9.2%
T6	7	15x11	838.21	70	12	655.1	767.29	154	11	655.3	8.6%
T7	8	12.5x24	1133.4	115	20	181.2	1032.1	235	16	181.3	9.0%
T8	9	15x13	1182.5	111	19	6270	1076.5	210	17	6271	9.0%

Table 2: Runtime and memory requirement of DP-RTBW vs. GRAPH-RTBW. Note that the delay is the same (omitted).

Data		DP-RTBW		GRAPH-RTBW	
name	pin	memory (MB)	time (s)	memory (MB)	time (s)
T1	2	3.33	12.4	1.47	9.8
T2	3	3.93	48.6	0.65	7.5
T3	4	12.18	480.6	1.48	15.7
T4	5	24.07	2100.4	1.59	57.9
T5	6	61.83	11218	2.13	131.9
T6	7	120.07	52671	2.47	655.3
T7	8	169.04	120908	1.49	181.3
T8	9	N/A	N/A	3.99	6271.9

Table 3: Runtime and delay for hierarchical router.

Data		Hierarchical		optimal	delay
name	pin	delay(ps)	runtime(s)	delay(ps)	off
H1	10	936.08	31.26	883.31	5.9%
H2	11	872.85	33.97	811.74	7.5%
H3	12	883.67	33.85	820.60	7.7%
H4	50	859.17	345.4	N/A	N/A
H5	100	1133.1	1316.4	N/A	N/A

We have implemented a hierarchical router based on the approach mentioned above, and tested its quality on Sun Solaris(Ultra10). We call LEDA package [11] to compute the minimum spanning tree and connected components. The test cases are generated randomly for the given number of sinks. For test cases where the number of sinks is not very large, we use flat GRAPH-RTBW algorithm to get the optimal delay for comparison purpose. Table 3 lists the results. Note that our hierarchical router is only 8% off from GRAPH-RTBW in terms of delay. In [7], DP approach is compared with a modified A-tree algorithm[15], and it shows that DP can outperform A-tree by up to 46% in terms of delay.

8. Concluding Remark

GRAPH-RTBW algorithm constructs a routing tree with simultaneous buffer insertion and wiresizing in the presence of routing and buffer obstacles. Our algorithm differs from dynamic programming, and is capable of addressing the problem of inverter insertion and sink polarity. Compared to dynamic programming approach, it reduces time and space complexity a lot. Wire sizing combined with buffer insertion can reduce the delay about 10% further compared to inserting buffers only for a routing tree. Wire sizing can reduce the number of inserted buffers although it increases wire area. In the applications where the number of sinks is very large, we can use the hierarchical router with acceptable quality.

9. References

- [1] C. Alpert and A. Devgan, "Wire segmenting for improved buffer insertion", DAC-97, pp. 588-593, 1997.
- [2] C. Alpert et al., "Buffered Steiner trees for difficult instances", ISPD-01, pp. 4-9, 2001.
- [3] C.P. Chen, Y.P. Chen, and D.F. Wong, "Optimal wire-sizing formula under the Elmore delay model", DAC-96, pp. 487-490, 1996.
- [4] C. Chu and D.F. Wong, "A quadratic programming approach to simultaneous buffer insertion/sizing and wire sizing", IEEE Transactions on Computer-Aided Design, Vol. 18, No. 6, pp. 787-798, 1999.
- [5] J. Cong, L. He, K.Y. Khoo, C.K. Koh, and D.Z. Pan, "Interconnect design for deep submicron ICs", ICCAD-97, pp. 478-485, 1997.
- [6] J. Cong, K.S. Leung, and D. Zhou, "Performance-driven interconnect design based on distributed RC delay model", DAC-93, pp. 606-611, 1993.
- [7] J. Cong and X. Yuan, "Routing tree construction under fixed buffer locations", DAC-00, pp. 379-384, 2000.
- [8] A. Jagannathan, S.W. Hur, and J. Lillis, "A fast algorithm for context-aware buffer insertion", DAC-00, pp. 368-373, 2000.
- [9] A.K. Jain and R.C. Dubes, "Algorithms for clustering data", Prentice hall, 1988.
- [10] R. Kay and L.T. Pileggi, "EWA: efficient wiring-sizing algorithm for signal nets and clock nets", IEEE Transactions on Computer-Aided Design, Vol. 17, No. 1, pp. 40-49, 1998.
- [11] LEDA package: <http://www.mpi-sb.mpg.de/LEDA/>.
- [12] M. Lai and D.F. Wong, "Maze routing with buffer insertion and wiresizing", DAC-00, pp. 374-378, 2000.
- [13] J. Lillis, C.K. Cheng, and T.T.Y. Lin, "Optimal wire sizing and buffer insertion for low power and a generalized delay model", ICCAD-95, pp. 138-143, 1995.
- [14] Y.Y. Mo and C. Chu, "A hybrid dynamic/quadratic programming algorithm for interconnect tree optimization", ISPD-00, pp. 134-139, 2000.
- [15] T. Okamoto and J. Cong, "Buffered Steiner tree construction with wire sizing for interconnect layout optimization", ICCAD-96, pp. 44-49, 1996.
- [16] J. Rubinstein, P. Penfield, and N.A. Horowitz, "Signal delay in RC tree networks", IEEE TCAD 2:3, pp. 202-211, 1983.
- [17] A.H. Salek, J. Lou and M. Pedram, "A simultaneous routing tree construction and fanout optimization algorithm", ICCAD-98, pp. 625-630, 1998.
- [18] A.H. Salek, J. Lou and M. Pedram, "MERLIN: semi-order-independent hierarchical buffered routing tree generation using local neighborhood search", DAC-99, pp. 472-478, 1999.
- [19] S.S. Sapatnekar, "RC interconnect optimization under the Elmore delay model", DAC-94, pp. 387-391, 1994.
- [20] Semiconductor Industry Association, National Technology Roadmap for Semiconductors, 1997.
- [21] L.P.P.P. van Ginneken, "Buffer placement in distributed RC-tree networks for minimal Elmore delay", ISCAS-90, pp. 865-868, 1990.
- [22] H. Zhou, D.F. Wong, I.M. Liu, and A. Aziz, "Simultaneous Routing and Buffer Insertion with Restrictions on Buffer Locations", DAC-99, pp. 96-99, 1999.