An Efficient Routing Tree Construction Algorithm with Buffer Insertion, Wire Sizing and Obstacle Considerations

Sampath Dechu
Physical Design Automation Group
Micron Technology Inc.
Boise, ID 83716
Tel: +1-208-368-2748
e-mail: sdechu@micron.com

Zion Cien Shen
Dept Of ECpE
Iowa State University
Ames, IA 50014
Tel: +1-515-294-7706
e-mail: zionshen@iastate.edu

Chris C. N. Chu
Dept Of ECpE
Iowa State University
Ames, IA 50014
Tel: +1-515-294-3490
e-mail: cnchu@iastate.edu

Abstract— In this paper, we propose a fast algorithm to construct a performance driven routing tree with simultaneous buffer insertion and wire sizing in the presence of wire and buffer obstacles. Recently several algorithms [1, 2, 3, 4] have been published addressing the routing tree construction problem. But all these algorithms are slow and not scalable. In this paper we propose an algorithm which is fast and scalable with problem size. The main idea of our approach is to specify some important high-level features of the whole routing tree so that it can be broken down into several components. We apply stochastic search to find the best specification. Since we need very few high-level features, the size of stochastic search space is small which can be searched in very less time. The solutions for the components are either pre-generated and stored in lookup tables, or generated by extremely fast algorithms whenever needed. Since it is efficient to obtain solutions for components, it is also efficient to construct and evaluate the whole routing tree for each specification. Experimental results show that, for trees of moderate size, our algorithm is at least several hundred times faster than the recently proposed algorithms [3, 4]. Experimental results also show that the trees generated by our algorithm have almost same delay and resource consumption as the trees generated by SP-Tree.

1. INTRODUCTION

As VLSI technology enters deep submicron era, interconnect delay becomes a dominant factor in determining circuit performance. Interconnect optimization techniques like buffer insertion and wire sizing have been shown to be effective in reducing interconnect delay [5]. In modern VLSI design, it is very common to consider buffer insertion and wiring sizing during performance-driven routing. For the routing of two-terminal nets and when there is no restriction on buffer positions in the routing area, the route with optimal delay can be constructed by inserting buffers and sizing wires of the shortest path from source to sink. In other words, routing and interconnect optimization can be performed sequentially. However, for multi-terminal nets or when there are macro blocks where wires can be passed but buffers cannot be placed, the optimal routing tree can only be found if routing, buffer insertion and wire sizing are considered simultaneously [4, 6].

Many algorithms have been proposed in the past few years to construct routing trees with buffer insertion and wire sizing in the presence of routing and buffer obstacles. The approaches used can be classified as either sequential or simultaneous approaches. In sequential approach, the routing tree is constructed followed by buffer insertion and wire sizing. In simultaneous approach, routing tree is constructed by simultaneously considering routing, buffer insertion and wire sizing. The algorithm proposed in [7] follows the sequential approach. Moreover, it does not consider wire obstacles, buffer obstacles and wire sizing. In [8], as part of sequential approach, Hu *et al.* extended van Ginneken's algorithm to solve the problem of buffer insertion on a given routing tree, considering buffer blockages.

Several algorithms have been proposed based on the simultaneous approach. Topology search based algorithms given in [1, 2, 3] limit the routing topology space to certain topologies and search exhaustively for the best solution in that limited space. The final routing tree obtained from these algorithms depends on the criteria used to limit the topology space and the initial routing topology given to these algorithms. In order to obtain a better solution, a larger topology space needs to be considered and the exhaustive search usually takes a significant amount of time. All these algorithms are not scalable and they cannot handle wiresizing. For two-terminal nets, Zhou *et al.* [9] presented a dynamic programming algorithm for simultaneous routing with buffer insertion, considering both buffer and wire obstacles. Lai and Wong [10] formulated the simultaneous routing with buffer insertion and wire sizing in the presence of buffer and wire obstacles as a graph-theoretic shortest path problem. However, these two algorithms cannot be easily extended to handle multi-terminal nets.

Recently, Tang *et al.* [4] presented an algorithm graph-RTBW for multi-terminal nets that considers buffer insertion, wire sizing, and buffer and wire obstacles simultaneously. In their approach, the routing problem is converted into a collection of graph problems. One graph is constructed for each subset of sinks. In the graph, one vertex represents the subset, and other vertices represent possible buffer choices at different buffer locations. The shortest path from the subset vertex to every other vertex v in the graph corresponds to the optimal subtree with appropriate buffer insertion and wire sizing connecting v and the subset of sinks. Dynamic programming is used to construct routing solutions for larger subset of sinks based on solutions for smaller subsets of sinks. Finally, the routing solution for all sinks is obtained. They use Rubinstein delay model [11] for interconnect delay calculation. As they consider all the subsets of sinks, the runtime is exponential to the number of sinks. Hence the algorithm is very slow.

In this paper, we present a very fast and scalable algorithm named Fast-RTBW for solving this problem. The main idea of our approach is to specify some important high-level features of the whole routing tree so that it can be broken down into several components. We apply stochastic search to find the best specification. Since we need very few high-level features, the size of stochastic search space is small. As size of the solution space is small, the time required to search for high level specifications of the routing tree is very less. The solutions for the components are either pre-generated and stored in lookup tables, or generated by extremely fast algorithms whenever needed. Since it is efficient to obtain solutions for components, it is also efficient to construct and evaluate the whole routing tree for each specification. Experimental results show that, for trees of moderate size, our algorithm is at least several hundred times faster than the recently proposed algorithms [3, 4], without much difference in delay and resource consumption.

Our approach has the following advantages over previous approaches:

1. Our approach is much faster and scalable. We apply stochas-

tic search on a *small* search space. The evaluation of a specific routing tree is also fast, because lookup tables and fast algorithms are used to find component solutions. Runtime of our algorithm decreases as number of blockages in the design increases, because we have to search less number of positions for buffers. The graph-RTBW algorithm uses exhaustive search by trying all combinations of subset of sinks, buffer positions and buffer sizes. As a result, the algorithm is very slow. Topology search based approaches try to reduce the runtime by limiting the search space to certain topologies. But these algorithms cannot handle wire sizing and are not scalable with problem size. These algorithms becomes slower as the number of blockages increases in the design.

2. We do not have restriction on the fanout of buffers. The graph-RTBW algorithm theoretically can handle general fanout. However, the expensive term in the time complexity of the algorithm is $O((t+1)^k B^{t+1} N^{t+1})$, where t is the bound on fanout, k is the number of sinks, B is number of buffer types, N is number of possible buffer locations. So, in practice, the algorithm can only handle a fanout of 2. The example given in Figure 1 demonstrates the disadvantage of having a restriction on fanout. Figure 1 (a) shows routing tree obtained by graph-RTBW for $t = 2$ and Figure 1(b) shows routing tree obtained by our algorithm. The delay of routing tree shown in Figure 1(b) is 37.7% better than the delay of routing tree shown in Figure 1(a). Also, we observe that routing resources are wasted in the first case.

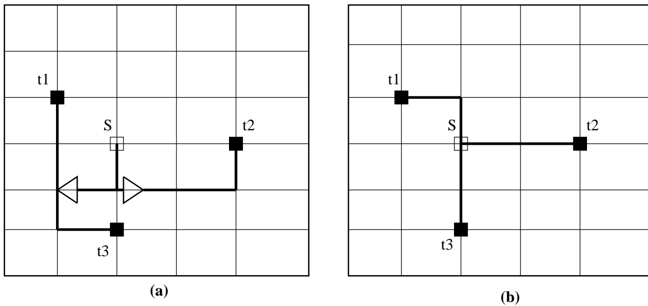

Figure 1: (a) Routing solution by graph-RTBW. (b) Routing solution by our algorithm

3. We use Elmore delay model [12] for calculating the delay of interconnects. It gives better estimation of delay when compared to Rubinstein delay model used in [4]. By using Elmore delay model, we can minimize the delay of critical path of multi-terminal net, which is not feasible by using Rubinstein model.

The rest of the paper is organized as follows: Section 2 presents the problem formulation. Section 3 explains the notation and the proposed algorithm in detail. Section 4 presents the complexity analysis of our algorithm. Experimental results are given in Section 5.

2. PROBLEM FORMULATION

The goal of the algorithm is to construct a routing tree with buffer insertion and wire sizing in the presence of wire and buffer obstacles, such that the maximum delay from source to sinks is minimized. We use π-type RC model for wires and switch-level RC model for buffers. We use Elmore delay model to calculate the interconnect delay. The problem formulation given below is the same as in [4].

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

3. THE FAST-RTBW ALGORITHM

In our approach, we divide routing tree into several components. Component solutions can be either pre-computed and stored in lookup tables or computed by fast algorithms whenever needed. Based on these lookup tables and fast algorithms, we search for the optimal routing tree using stochastic search technique. We use simulated annealing for searching in our implementation. The following subsections define notation we use in this paper and explain the details of our approach.

3.1 Notation

We use same notation given in [4] for the following terms:

- W: Library of different wire types.
- B: Library of different buffer types.
- V: All nodes present in grid graph G.
- $N = \{s\} \bigcup \{t_i \,| i = 1, \ldots, k\} \bigcup \{\text{buffer nodes}\}$: Set containing source, sink and buffer nodes. Buffer nodes are the nodes where buffers can be placed. Clearly, in the presence of obstacles, number of elements in N is less than number of elements in V, which is set of all nodes in grid graph.
- **Wire Path:** A path connecting two nodes in N by properly sized wires but no buffers between.
- **Buffered Path:** A path connecting two nodes in N with buffers inserted between them.

We introduce the following new terms:

- **Leaf Buffer Path:** A buffered path connecting a buffer node directly to a sink without any wire branch in between. Leaf buffer path can be of zero length. A sink will be a leaf buffer path with zero length, if there is no buffer driving it directly without any wire branch in between.
- **Branch:** Branch is a tree component connecting three or more nodes in N, without any internal buffers. Every branch contains a driver buffer which is driving the branch and several receiver buffers which are connected to driver. Number of receivers in branch is called degree of branch denoted by t. In [4], branch is referred as Buffer Combination.
- **Stem Buffered Branch:** A branch which can have buffers on stem. Every branch can be considered as stem buffered branch with no buffers on stem.
- **Component Driver:** Buffer driving a stem buffered branch or a leaf buffer path is called component driver.

Figure 2 shows, the notations that we use in this paper. In the figure, the source is called the component driver, because we assume that the driver resistance of the source equals one of the resistance of buffers in buffer library B and the load capacitance of the sinks equals one of the capacitance of buffers in B. If not, we can always add additional buffer types with source resistance and sink capacitance in B and letting the buffer type to be used only at source and sinks.

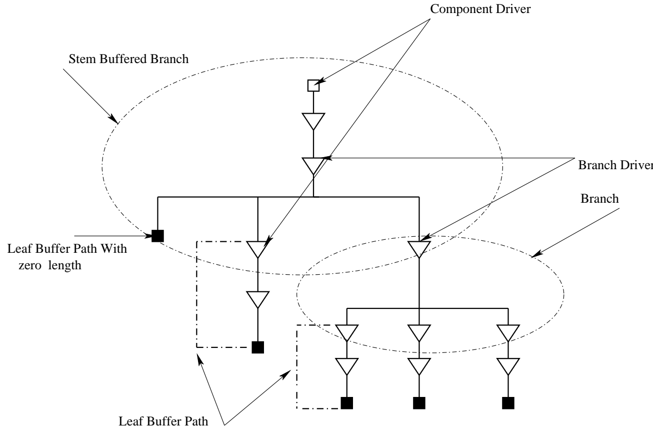

Figure 2: Illustration of Notations

3.2 Decomposition of Routing Tree

If source and sinks are also considered as buffers present in buffer library B, from Figure 3, it is clear that every routing tree contains only two components:

1. Stem buffered branch
2. Leaf buffer path

If we know, the arrival time at each receiver, the locations of the receivers and the location of driver of a stem buffered branch, we can find the optimal sizes for the buffers. The routing for that stem buffered branch with proper wire sizing can also be found. For any buffered path, if we know the location of the driver and receiver buffers, we can find the optimal buffered path connecting them. We use lookup tables to calculate the delay of these two tree components. By dividing the routing tree into stem buffered branches and leaf buffer paths, we can pre-compute the solutions of components of the routing tree and tabulate them. We can use these lookup tables to construct the routing tree in very less time.

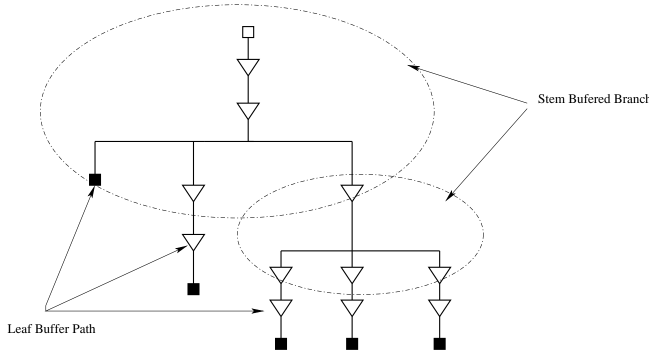

Figure 3: Routing Tree Decomposition

3.3 Component Construction

In this section, we explain how the solutions of components are pre-computed. As explained in the previous section, every routing tree consists of only stem buffered branch and leaf buffer path. Leaf buffer path is nothing but buffered path between a node in N and a sink. Stem buffered branch consists of two parts, one is branch and the other is buffered path between component driver and branch driver. Hence if we have pre-computed delays of buffered paths and branches, we can calculate the delays of components. We construct the delay lookup tables as explained below,

3.3.1 *Buffered Path Delay Table (BPDT)*

Buffered Path Delay Table contains the delay of optimal buffered path between two nodes in N. If we know the sizes of buffers at buffer nodes and locations of those buffers, we can find the delay of the optimal buffered path between those two buffer nodes by looking $BPDT(b_1, b_2, l_1, l_2)$, where b_1 and b_2 are buffer sizes, and l_1 and l_2 are buffer locations.

To construct buffered path delay table, we construct the following tables:

1. Shortest Wire Path Length Table($SWPLT$): This contains the shortest wire path length between two nodes in the routing grid. We can represent this table in a functional form as $SWPLT(l_1, l_2)$, where l_1 and l_2 are the locations of two nodes. We use Lee's maze routing algorithm to construct the table. This length is nothing but distance between two nodes, if we route them avoiding only wire obstacles.

2. Shortest Buffered Path Length Table ($SBPLT$): This contains the shortest buffered path length between two nodes in routing grid. We can represent this table in a functional form as $SBPLT(l_1, l_2)$, where l_1 and l_2 are the locations of two nodes. We use Lee's maze routing algorithm to construct this table. This length is nothing but, distance between two nodes, if we route them avoiding both buffer and wire obstacles.

3. Wire Path Delay Table($WPDT$): This table contains the optimal wire path delay with wiresizing given the driver, load and length of the path. We can represent this table in functional form as $WPDT(b_1, b_2, L)$, where b_1 and b_2 are buffer sizes placed at source and sink, respectively, and L is the length of the wire path length between these two nodes. If the locations of two nodes l_1 and l_2 are given, we can get L from $SWPLT(l_1, l_2)$. We use dynamic programming technique to construct $WPDT$.

4. Buffered Wire Segment Delay Table ($BWSDT$): This table contains the optimal delay of a wire segment, after placing buffers on it. Assuming a buffer of size b_1 is driving a buffer of size b_2 through a wire segment of length L, this table can be represented in functional form as $BWSDT(b_1, b_2, L)$. We use dynamic programming technique to construct this table.

Using the tables above and the pruning technique given below, we can construct $BPDT$ in a very short time. In our pruning technique, we are separating the pair of nodes whose shortest wire path does not overlap with buffer blockages from all the other pairs. For those pairs, we need not consider the buffer blockages while doing buffer insertion. Hence we can find buffer insertion solution in $O(N^2)$ time. The table $BWSDT$ contains the buffer insertion and wire sizing solution avoiding buffer blockages. Now for the pairs of nodes whose shortest wire path overlaps with buffer blockages, we have to consider buffer blockages while doing buffer insertion. We speed up the buffer insertion by limiting the number of buffers we insert based on the maximum length of wire path between any two pair of nodes. The construction of $BPDT$ is summarized in the algorithm below. It inserts up to three buffers on a wire path. If Step 3 is executed one more time, it inserts up to seven buffers.

Step 1: // *Initialization*
$BPDT(b_u, b_v, l_u, l_v) = BWSDT(b_u, b_v, SBPLT(l_u, l_v))\ \forall u, v \in N$
Step 2: // *For inserting one buffer*
for each $w \in N$
if($SWPLT(l_u, l_w) + SWPLT(l_w, l_v) < SBPLT(l_u, l_v)$)
$BPDT(b_u, b_v, l_u, l_v) = \min(BPDT(b_u, b_v, l_u, l_v),\ WPDT(b_u, b_v, SWPLT(l_u, l_v)))$
$SBPLT(l_u, l_v) = SWPLT(l_u, l_w) + SWPLT(l_w, l_v)$
Step 3: // *For inserting three buffers*
for each $w \in N$
if($SBPLT(l_u, l_w) + SBPLT(l_w, l_v) < SBPLT(l_u, l_v)$)
$BPDT(b_u, b_v, l_u, l_v) = \min(BPDT(b_u, b_v, l_u, l_v),\ WPDT(b_u, b_v, SWPLT(l_u, l_v)))$
$SBPLT(l_u, l_v) = SBPLT(l_u, l_w) + SBPLT(l_w, l_v)$

3.3.2 *Branch Delay Table (BDT)*

Branch Delay Table contains the optimal delay of wire sized Steiner tree connecting three nodes in N. If we know the buffer sizes and the locations of three nodes in N, we can find the optimal delay by $BDT(b_1, b_2, b_3, e, l_1, l_2)$ where b_1 , b_2 and b_3 are buffer sizes of the three buffers and e, l_1, l_2 are stem and branch lengths . These are shown in Figure 4. We use dynamic programming approach to construct this table. This table will give the delay of a branch only when the degree t of the branch is two. For the cases of $t > 2$, we can use any fast Steiner tree construction algorithm to get the solution.

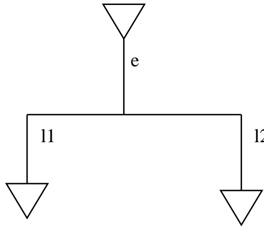

Figure 4: Branch

3.3.3 *Stem Buffered Branch Construction*

Assume that we know the positions and sizes of component drivers and receivers, delays of subtrees at the receivers. To make use of buffered path delay table and branch delay table/fast Steiner tree construction algorithm, for constructing optimal stem buffered branch connecting driver to receivers, the position and size of branch driver should be available. In our approach, for optimal branch driver position, we try all the locations present in the small grid space R which covers bounding box of the component driver and receivers of stem buffered branch. We place the branch driver at particular node in R and try all the buffer sizes available in buffer library. We size and position the branch driver such that, the maximum Elmore delay at the receivers is minimized. For leaf buffer path, if we know the position and size of component driver, we can get the optimal routing using buffered path delay table.

3.4 Routing Tree Construction

In this section and the following section we explain how to size and position the component drivers and compute the delays of subtrees at each component driver. As we mentioned before, we can use any stochastic search algorithm to search for the routing tree. In our implementation, we use simulated annealing as search algorithm. We fix the positions of component drivers in simulated annealing. Assume that the positions of component drivers of all components present in routing tree are known. To fix the sizes of buffers, we use bottom-up dynamic program approach. We start from leaf buffer paths. For a leaf buffer path, we know the position of component driver. For the size, we try all the buffer sizes present in B for that component driver of leaf buffer path. As a result, we have B different routing solutions for that leaf buffer path. For stem buffered branch, solutions for the subtrees at each of the receivers must have been calculated already. So subtree at each receiver of stem buffered branch has B different solutions corresponding to B different sizes for that receiver. Similar to leaf buffer path case, we try all the buffer sizes present in library for component driver, and get B solutions for stem buffered branch, each corresponding to B different sizes of that component driver. We can observe that at any component driver in routing tree, we have only B different solutions for the subtree driven by that component driver. Finally at the source, each receiver of the stem buffered branch driven by source have B different solutions for their respective subtrees. Hence we have B^{t_s} different solutions at the source, where t_s is the degree of stem buffered branch driven by source. Among these, final routing tree solution is the one which gives minimum of maximum Elmore delay at each receiver.

3.5 Routing Tree Perturbation

In simulated annealing we fix the positions of component drivers present in routing tree. We defined moves to perturb the routing tree. Some of the moves defined are applicable only to binary trees. To use same moves for non binary trees, we transform non binary trees into binary trees by adding some **dummy nodes** to it. Final solution from our algorithm is independent of binary tree representation. Figure 5 shows an example of this transformations. While transforming to binary tree, we avoid redundant binary trees for same non binary tree, by putting a restriction that only the right child of any node can be dummy node. We defined moves to make a node to dummy node and also a dummy node to real node. We explain all these moves in the following subsections. In simulated annealing, we apply these moves randomly.

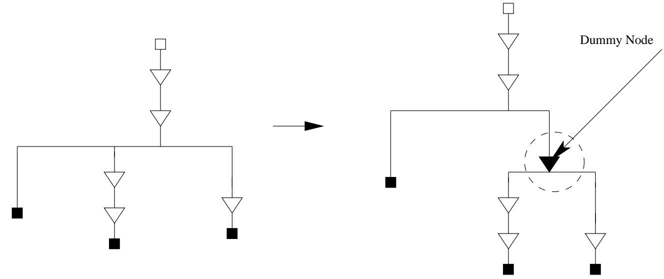

Figure 5: Transformation of Non binary tree to Binary Tree

1. **Move 1 – Component Driver Position Change**:
   In simulated annealing, we change the positions of component drivers of stem buffered branch and leaf buffer path. For component driver of stem buffered branch, we randomly select one buffer node among 8 adjacent nodes and change the position of the component driver to that position. For component driver of leaf buffer path, we make a greedy move. We visit all the buffer nodes among 8 adjacent nodes of component driver and change its position to a node, where the sum of leaf buffer path delay and wire path delay from its parent is minimum.

2. **Move 2 – Swapping of Sinks**:
   This is a topology changing move. In this move, we select component drivers of two leaf buffer paths (which can be sinks them selves when the length of leaf buffer path is zero) driven by two different parents and swap their parents.

3. **Move 3 – Rotation:**
   This is also a topology changing move. In this move, we have two types of rotations [13], one is left rotation and other is right rotation. Figure 6 shows these two operations. Using these operations we can get all binary trees that can be constructed with the given terminals. So we visit all the routing topologies in simulated annealing. When we are making right or left rotations, if a node is violating the restriction that, only right child of a parent node can be dummy, we change that dummy node to real node and then make the rotation. We explain how to make a dummy node to real node in the next move.

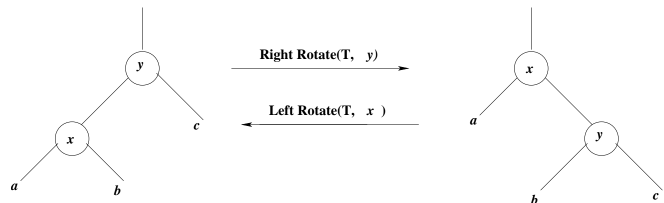

Figure 6: Rotation Operations

4. **Move 4 – Conversion between Dummy and Real Nodes:** In this move, we select a buffer node randomly from set N and make that node dummy if it is a real node, make node real if it is dummy. When we make node dummy, we set a bit which indicates that the node is dummy. When we make a dummy node to real, we need to give that buffer a size and location. We give the size of buffer as minimum size that is available in B and for location we search the unblocked nodes near its left child (which cannot be a dummy node) and assign its location to that node.

4. COMPLEXITY ANALYSIS

4.1 Time Complexity

For a given routing grid graph $G(V, E)$, the time complexity for lookup table construction is calculated below:

1. Shortest Wire Path Length Table and Shortest Buffered Path Length Table: We use Lee's algorithm to construct these tables. Runtime to construct these tables is $O(V^2)$.

2. Wire Path Delay Table: We use dynamic programming technique to do wire sizing. In our approach, we do wire sizing only if the path length between two nodes is less than critical length [14]. *Critical length* is defined as the minimum wire length beyond which buffer insertion will help to reduce the interconnect delay. If path length exceeds the critical length, we connect those two nodes by thickest wire available in library W. Let L be the critical length between two nodes for which wire sizing is needed, then runtime to construct this table is $O(B^2N^2)$.

3. Buffered Path Delay Table: The runtime for constructing this table without pruning is $O(N^3B^3)$. Because of pruning technique that we are using to construct this table, the runtime required is very less. Let T_b be the runtime required to build this table.

4. Branch Delay Table: We use dynamic programming technique to construct this table. Similar to Wire Path, when the branch length or stem length of a branch is greater than critical length, we use thickest wire present in the library to connect the driver to the receivers of branch. We are constructing this table only for branches with degree two. Runtime required to construct this table is given as $O(L^3B^3)$, L is the critical length. For branches with degree more than two, we can use any fast Steiner tree construction algorithm. But for simplicity, we assume that the receivers are directly connected to the driver without any Steiner point. We calculate delay for these branches by looking Wire Path Delay Table, without any extra runtime.

Total runtime of our algorithm is $O(V^2 + B^2N^2 + T_b + L^3B^3 + MC)$ where M is the number of iterations in simulated annealing and C is the time to compute delay of the routing tree in each iteration. We are doing buffer sizing in each iteration of Simulated Annealing, by using dynamic programming technique. Runtime for each iteration of simulated annealing, C is $O(ST_l^2B^{t+1}R)$, where t is maximum degree of a branch in routing tree, S is the number of stem buffered branches present in routing tree, R is the grid space we search for branch driver location of each stem buffered branch, T_l is the time to look up buffered path delay table or branch delay table. Usually T_l and S are small numbers and R is also small number when compared to N. From above expression it is clear that, our algorithm is very fast when compared to graph-RTBW. By neglecting non-dominant terms we can express our runtime as $O(T_b + M(ST_l^2B^{t+1}R))$.

4.2 Memory Complexity

The memory space required for the lookup tables is $O(V^2+B^2N^2+B^3L^3)$. The space required to store binary tree can be given by $O(B(2k-1))$, where k is the number of sinks. Hence total memory required can be given by $O(V^2 + B^2N^2 + B^3L^3 + B(2k-1))$.

4.3 Size of Solution Space

From [15], we know that the tight bound on the number of binary trees with k leaves is $\theta(\frac{k!2^{2.543k}}{k^{1.5}})$. As we have to search for $2k-1$ buffer locations, we have $N^{(2k-1)}$ options for different buffer locations. In our approach we make some nodes dummy to consider routing trees with fanout more than 2. When we are making a node dummy, we avoid the redundant trees by making a node dummy if and only if it is a right child. So we have $2^{(\frac{k-2}{2})}$ different topologies. Hence the total solution space can be given by $O(\frac{k!2^{2.543k}}{k^{1.5}}N^{(2k-1)}2^{(\frac{k-2}{2})})$.

5. EXPERIMENTAL RESULTS

We implemented Fast-RTBW in C language and tested it on a Sun Ultra-2 750MHz machine with 8GB memory. We tested graph-RTBW and SP-Tree also on the same machine. The results reported are obtained by testing three programs on same specified machine.For graph-RTBW, we calculated the Elmore delay for all the trees generated and reported these delay values to maintain the consistency with delay models of other two algorithms. We used same technology parameters given in [4]. Our grid is of size $17 \times 17mm^2$ with horizontal and vertical grid lines spaced at 0.5mm distance from each other.

Table 1 shows the comparison between our algorithm, graph-RTBW and SP-Tree with single buffer and wire type. In Table-1, we used $B = 1$ and $W = 1$, because SP-Tree cannot handle wiresizing and graph-RTBW is implemented only for one buffer type. In our experiments, we run simulated annealing for 10 times and take the best result. All the testcases are randomly generated. As the grid for each testcase is same, the runtime for lookup table construction is same for all the testcases. We can observe that, the time taken for simulated annealing is very less, because of the small solution space. Hence, once the grid is fixed we can construct lookup tables once and use same set of lookup tables to route all the nets present in the grid.For small testcases, we are much faster than graph-RTBWS, but slower than SP-Tree. For, moderate size testcases, we are several hundred times faster than both the algorithms.Total CPU Time for our algorithm includes time for constructing lookup tables and running simulated annealing 10 times. As graph-RTBW minimizes only maximum delay, we have implemented our algorithm to optimize the maximum delay.But, in practice we can optimize minimum slack instead of maximum delay.

In **Table 2,** we compare our algorithm with SP-Tree for more than one buffer type. Even though we are slow for small test cases, we are very fast for moderate and large testcases. We can easily observe that our algorithm is scalable with problem size.

Table 3: $B = 1, W = 3, Blockages = 11, Grid = 17 \times 17mm^2$

DATA	graph-RTBW [4]		FAST-RTBW			
name	delay	CPU	delay	CPU(s)		
	(ps)	(s)	(ps)	LUT	SA-1	Total
NET4	918	171.23	918	22.5	1.91	41.6
NET5	851	310	851	22.5	2.61	48.6
NET6	1055	681	1061	22.5	3.314	55.64
NET8	1166	4971	1172	22.5	4.15	63.95
NET13	*	$>$ 10 hrs	1057	22.5	7.81	100.55
NET15	*	$>$ 10 hrs	885	22.5	8.9	111.51
NET18	*	$>$ 10 hrs	1061	22.5	11	132.35
NET21	*	$>$ 10 hrs	1049	22.5	13.6	158.2
NET23	*	$>$ 10 hrs	1092	22.5	15.91	181.59
NET25	*	$>$ 10 hrs	1041	22.5	17.9	201.49

In **Table 3,** we compare our algorithm with graph-RTBW for more than one wire type. We handle wiresizing without any increase in

Table 1: $B = 1, W = 1, Blockages = 11, Grid = 17 \times 17mm^2$

DATA	graph-RTBW [4]				SP-TREE [3]				FAST-RTBW					
name	delay (ps)	CPU (s)	WL (mm)	buf	delay (ps)	CPU (s)	WL (mm)	buf	delay (ps)	CPU(s) LUT	SA-1	Total	WL (mm)	buf
NET4	1008	170	44.5	13	1009	2	43	9	1008	22.5	1.31	40.63	43	11
NET5	937	305.56	47	14	937	7.1	45	10	937	22.5	3.04	52.94	45	11
NET6	1157	663.06	72.5	22	1166	17	53.5	14	1179	22.5	4.04	62.9	54.5	16
NET8	1281	4895	76	26	1285	91	70.5	16	1288	22.5	3.04	52.94	70	19
NET13	*	> 10 hrs	*	*	1136	1387	79	18	1149	22.5	6.06	83.16	77	20
NET15	*	> 10 hrs	*	*	941	7287	81.5	24	958	22.5	8.01	102.64	83	22
NET18	*	> 10 hrs	*	*	1047.7	16729	120.5	33	1071	22.5	9.79	120.35	124	35
NET21	*	> 10 hrs	*	*	*	> 10 hrs	*	*	1081	22.5	11.6	138.24	122	38
NET23	*	> 10 hrs	*	*	*	> 10 hrs	*	*	1057	22.5	13.3	155.4	130	34
NET25	*	> 10 hrs	*	*	*	> 10 hrs	*	*	1079	22.5	14.8	170	142	40

Table 2: $B = 2, W = 1, Blockages = 11, Grid = 17 \times 17mm^2$

DATA	SP-Tree [3]				FAST-RTBW					
name	delay (ps)	CPU (s)	wl (mm)	buf	delay (ps)	CPU(s) LUT	SA-1	TOTAL	wl (mm)	buf
NET4	848	9.3	44.5	12	851	145	3.924	184.24	43	13
NET5	787	33	45.5	12	790	145	8.025	225.25	45	13
NET6	977	71	52	17	987	145	14.8	293.06	52	20
NET8	1074	455	69	16	1091	145	24.37	388.71	72	24
NET13	954	5983	79.5	23	973	145	57.63	721.3	80	21
NET15	790	29000	85	28	804	145	66.37	808.72	87	20
NET18	*	> 10 hrs	*	*	939	145	88.5	1030.45	129	37
NET21	*	> 10 hrs	*	*	962	145	122.24	1367.4	144	36
NET23	*	> 10 hrs	*	*	924	145	134.3	1487.77	138.5	37
NET25	*	> 10 hrs	*	*	935	145	152.4	1668.5	152.5	43

runtime. But, from Table 1, we can conclude that compared to graph-RTBW, we are always much better in resource consumption. When wiresizing is considered, our algorithm is several hundred times faster than graph-RTBWS. As we do not have any code or testcases that are used in hierarchical graph-RTBWS, we cannot compare our algorithm with hierarchical approach. But we can also apply our algorithm hierarchically.

6. CONCLUSIONS

We have presented a fast and efficient algorithm to construct routing tree with simultaneous buffer insertion and wire sizing. While constructing routing tree, we consider both buffer and wire obstacles present. Compared to topology search approaches and graph-RTBW, our algorithm is several hundreds times faster for moderate and large testcases. Our algorithm is scalable with problem size. We handle wire sizing without any increase in runtime.

7. REFERENCES

[1] J. Lillis, C. K. Cheng, T. T. Lin, C. Y. Ho, "Performance driven routing techniques with explicit area/delay tradeoff and simultaneous wire sizing", in *Proc. ACM/IEEE Design Automation Conf.*, 1996, 395-400.

[2] M.Hrkic, J. Lillis, "S-Tree: A technique for Buffered routing tree synthesis", *SASIMI*, 2001, pp. 242-249.

[3] M.Hrkic, J. Lillis, "Buffer tree synthesis with consideration of temporal locality, sink polarity requirements, solution cost and blockages", in *Proc. Int. Symp. Physical Design.*, 2002, pp. 98-103.

[4] X. Tang, R. Tian, H. Xiang, and D. Wong, "A new algorithm for routing tree construction with buffer insertion and wire sizing under obstacle constraints" in *Proc. Int. Conf. Computer-Aided Design.*, 2001, pp. 49-56.

[5] Jason Cong, Lei He, Cheng-Kok Koh, and Patrick H. Madden. Performance optimization of VLSI interconnect layout. *INTEGRATION, the VLSI Journal*, 21:1–94, 1996.

[6] Takumi Okamoto and Jason Cong. "Buffered Steiner tree construction with wire sizing for interconnect layout optimization". In *Proc. IEEE Intl. Conf. on Computer-Aided Design*, pages 44–49, 1996.

[7] C. Alpert, G. Gandham, M. Hrkic, J. Hu, A. Kahng, J. Lillis, B. Liu, S. Sapatnekar, A. Sullivan, and P. Villarubia. "Buffered Steiner Trees for difficult instances". in *Proc. Int. Symp. Physical Design.*, 2001, pp. 4-9.

[8] J.Hu, C. Alpert, S. T. Quay, and G. Gandham, "Buffer Insertion with Adaptive Blockage Avoidance", in *Proc. Int. Symp. Physical Design.*, 2002, pp. 92-97.

[9] 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.*, 1999, pp. 96-99.

[10] M.Lai and D. Wong, "Maze routing with buffer insertion and wiresizing", in *Proc. ACM/IEEE Design Automation Conf.*, 2000, pp. 374-378.

[11] J. Rubinstein, P. Penfield, and N. A. Horowitz, " Signal delay in RC tree networks", *IEEE Transactions on Computer-Aided Design.*, 1983, pp. 202-211.

[12] W. C. Elmore, "The transient response of damped linear networks with particular regard to wideband amplifiers", *Journal of Applied Physics*, 1948, 19:55-63.

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, "Introduction to Algorithms", *Prentice - Hall,* 1997.

[14] R.Otten and R. Brayton, " Planning for performance" in *Proc. ACM/IEEE Design Automation Conf.*, 1998, pp. 122-127.

[15] Cien Shen and Chris Chu, "Bounds on Number of Slicing, Mosaic, and General Floorplans", *IEEE TCAD*, Vol. 22, No. 10, October 2003.

[16] C. Alpert, C. Chu, G. Gandham, M. Hrkic, J. Hu, C. Kashyap, S. Quay, "Simultaneous driver sizing and buffer insertion using a delay penalty estimation technique", *IEEE TCAD*, 2004.

[17] J . Lillis, C. K. Cheng, and T. T. Y. Lin, "Optimal wire sizing and buffer insertion for low power and a generalized delay model", in *Proc Int. Conf. Computer-Aided Design.*, 1996, pp. 134-139.