Buffered Routing Tree Construction Under Buffer Placement Blockages*

Wei Chen and Massoud Pedram
University of Southern California
Los Angeles, CA 90089

Premal Buch
Magma Design Automation
Cupertino, CA 95035

Abstract

Interconnect delay has become a critical factor in determining the performance of integrated circuits. Routing and buffering are powerful approaches to improve circuit speed and correct timing violations after global placement. This paper presents a dynamic- programming based algorithm for performing net topology construction and buffer insertion and sizing simultaneously under the given buffer-placement blockages. The differences from some previous works are that (1) the buffer locations are not pre-determined, (2) the multi-pin nets are easily handled, and (3) a line-search routing algorithm is implemented to speed up the process. Heuristics are used to reduce the problem complexity, which include limiting number of intermediate solutions, using a continuous buffer sizing model, and restricting the buffer locations along the Hanan graph. The resulting algorithm, named BRBP, was applied to a number of industrial designs and achieved an average of 7.9% delay improvement compared to a conventional design flow.

1. Introduction

With the rapid decrease in device sizes, resistance per unit length of interconnects has risen. Meanwhile, chip sizes and global wire lengths continue to grow rapidly. These factors make interconnect delay play an increasingly important role in circuit performance. Many optimization techniques have been developed to reduce interconnect delays. Among them, global routing and buffer sizing have a significant effect on interconnect delay.

Conventional design flow proceeds as follows. First, net topology is determined by constructing a Steiner tree or a shortest path routing tree, next buffers are inserted into this topology and sized. In [1], a dynamic-programming based algorithm to insert and size buffers for a given net topology is proposed. The objective is to maximize the required arrival time at the driver pin of the net. This technique has proven quite effective when the inserted buffers can be placed anywhere on the chip. However, in reality there are many placement blockages in the circuit that restrict the areas on the chip where the buffers can be placed. Since they do not block routing, wires can go through these blockages (though they may not use all possible layers). The algorithm of [1] does not perform well in such a situation, and hence a new algorithm must be developed that performs net topology design and buffer insertion simultaneously; otherwise, the existence of a fixed, a priori topology that may go through placement blockages will greatly limit the effectiveness of the subsequent buffer insertion step.

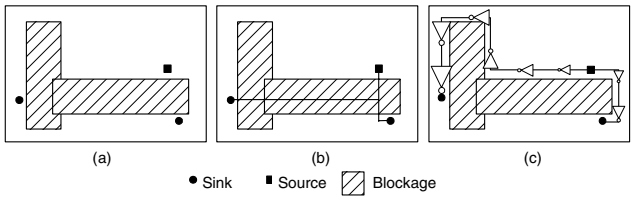

Figure 1. An example of wire buffering with placement blockages

A small example is given in Figure 1(a). Based on a conventional flow tool, the global router, which can route through the placement blockages, constructs a Steiner tree as shown in Figure 1(b). The subsequent wire-buffering tool cannot insert buffers for this net because it is blocked almost completely. Another choice is to specify that the placement blockages are also routing blockages. The global router can go around all the blockages, and the subsequent wire buffering process can insert buffers on the net, as shown in Figure 1(c). However, for the left sink, a net, which goes through the vertical blockage and connects the sink to the source, is actually a better solution.

The authors of [2] presented a shortest-path based algorithm to perform routing and buffer insertion simultaneously with restrictions on the buffer locations. The authors attempted to find the shortest Elmore delay path between a source pin and a sink pin. They used maze routing to expand the solution from the sink to the source. At every node of the grid, a buffer can be inserted to improve the timing. However, due to the nature of shortest-path algorithm, this method only works for two-pin nets. A dynamic-programming based algorithm that handles multi-pin nets was given in [4]. This method does not perform buffer sizing, which can be very effective and useful in optimizing circuit timing. In addition, reference [4] depends on the assumption that the possible buffer locations are pre-defined. It works well in design flows

* This work is supported in part by grant number MIP-9988441 from the National Science Foundation. Wei Chen is with Synopsys, Inc. at the present time.

with buffer stations. However, in practice, for design flows without buffer stations, the possible buffer locations are very difficult to determine a priori because the buffers can be placed anywhere outside the blockages, and the buffer locations influence the wire topology to a great extent. Reference [3] addressed a similar problem.

In this paper, a dynamic programming based algorithm is proposed that performs global routing and buffer/inverter insertion and sizing for the design flow without buffer stations. It can also handle multi-pin nets. We assume that placement blockages for the buffers are given. Areas other than these blockages are available for inserting buffers. Net topology is generated concurrently with the determination of buffer locations and size. Instead of maze routing, a line-search routing algorithm is used. Buffers are inserted not only at the nodes of the graph but also on the long edges of the graph.

The remainder of the paper is organized as follows. The problem definition and the delay model are introduced in Section 2. The dynamic-programming based algorithm is presented in Section 3. Experimental results and conclusions are given in Sections 4 and 5, respectively.

2. Preliminaries

We define the *Buffered Routing with Placement Blockage* (BRBP) problem as follows. Given (1) a set of placement blockages where routing is allowed but no buffers can be placed and (2) locations of the source pin and the sink pins of all the nets, simultaneously build the net topologies and insert sized buffers/inverters at places where they are allowed to improve the circuit timing.

To calculate the delay of the buffer/inverter, the logical-effort based delay model [5] is adopted. This model is a reformulation of the conventional RC model of CMOS gate delay. The delay of a buffer $d=\tau\ (p+gh)$. τ is a scaling parameter that characterizes the semiconductor process being used. It converts the unit-less quantity $(p+gh)$ to d, which has time units. Without loss of generality, we drop τ from now on. p is the parasitic delay of the gate. g is called the *logical effort* of the gate, which depends on the topology of the gate. h is the *electrical effort* (or gain), which is defined as C_l/c_{in}, while c_{in} is the input pin capacitance, and C_l is the capacitance load. p and g are independent of the gate size, so when h is fixed, the delay is also fixed.

To account for the interconnect delay, the Elmore delay model [6] is used in this paper. If the unit length wire resistance and capacitance are denoted by r_0 and c_0, respectively, and the wire length is denoted by l_w, the resistance and capacitance of the wire are: $r_w=r_0 \cdot l_w$ and $c_w=c_0 \cdot l_w$. The wire delay is calculated as follows: $d_w = r_w \cdot (\frac{1}{2} c_w + C)$, where C is the load driven by the wire.

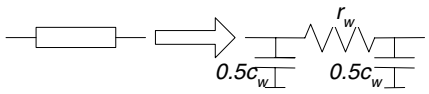

Figure 2. Elmore delay model

3. BRBP Algorithm

This paper presents a dynamic-programming based algorithm for solving the Buffered Routing problem in the presence of Placement Blockages (named the BRBP algorithm). First, a Hanan graph is created from the locations of the source, sinks, and blockages. For each sink pin, base solutions are generated, and a line search technique is adopted to propagate the solutions. Starting with low-level solutions, the existing solutions are merged to obtain a new higher-level solution. Buffers/inverters are not inserted only at the nodes of the Hanan graph. Long edges in the graph are divided into small segments, and buffer/inverter insertion is attempted for endpoints of each segment. This process is repeated until the topology of the complete net is achieved.

3.1 Hanan graph

Hanan proved that there always exists a Rectilinear Steiner Minimum Tree for a terminal set where all Steiner points are placed on the Hanan grid, which is the set of points formed by the intersection of horizontal and vertical lines through the terminals [7]. In the BRBP algorithm, the Hanan grid of a given net is created first and subsequent operations are performed on this graph. Considering the importance of the placement blockages, corners of the blockages are treated as sink pins. The Hanan graph of the example in Figure 1 is shown in Figure 3. The small solid round dots represent the valid corners of the blockages. Notice that the two corners on the left side of the horizontal blockage are dropped. The reason is that they are both covered by the other blockage. In general, if a corner is blocked, there is no chance to insert a buffer around the corner. Furthermore, the fewer the number of Hanan points, the smaller the time complexity and memory requirement. Therefore, only the unblocked corners are used to construct the Hanan graph.

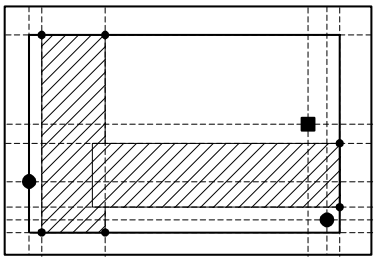

Figure 3. The Hanan graph of the example in Figure 1

Another question with respect to the Hanan graph is how to set its boundary. In previous works, the boundary is the minimum bounding box of the source pin and the sink pins of the net. In this problem, the option to go around all the blockages should always remain. Consequently, the previous choice is not adequate. The algorithm in Figure 4 shows a method for creating the Hanan graph for a net n and blockage set B.

Algorithm Hanan_graph(*n, B*)

1. initialize the Hanan graph *H* as the minimum-bounding box for the source and sinks;
2. while there is a blockage $b \in B$ that is cut by *H,* do
3. enlarge *H* to cover *b* completely;
4. return *H*;

Figure 4. Algorithm Hanan_graph

Hanan_graph begins with *H* as a traditional minimum bounding box for the entire source and sink pins. It then iteratively enlarges the boundary of *H* to hold all the blockages cut by *H* until the boundary does not intersect any blockage. In Figure 3, the smaller solid rectangle represents the boundary of the Hanan graph of that net.

3.2 Data structure and base solutions

In the algorithm presented here, each solution *sol* has a quintuple labeling (*root, cap, req, reachable_set, repeater*). These labels are defined as follows.

- ***root*** - a pointer to the node in the Hanan grid, which is root of the node tree formed by the current solution *sol*
- ***cap*** - the capacitive load of *sol* as seen from *root*
- ***req*** - the required arrival time for *sol* at *root*
- ***reachable_set*** - a set of pointers to the nodes that are reachable from *root* (the node tree formed by *sol*)
- ***repeater*** - repeater of type (i.e., buffer, inverter, or null) and size inserted at *root.*

A continuous buffer-sizing model is adopted to perform the buffer sizing. The reasons for this are as follows. (1) In today's ASIC design library, the number of available sizes for the buffer/inserter is so large that the error in rounding the continuous buffer size to a discrete buffer size is negligible. (2) In order to perform buffer sizing with a discrete sizing model, each available size has to be tried whenever a new buffer is inserted. Moreover, many of the sizing solutions have to be stored for later use during dynamic programming. This results in a very long computation time and large memory usage. (3) With the gain-based delay model [5], delay is a function of only the gain. As a result, a number of small buffers driven by the same source with the same gain h can be merged to a large buffer with the same gain h [8].

For each point in the Hanan graph, the lowest level solutions, i.e. base solutions are generated:

1. For a sink pin point *p*, there is one base solution *sol(p, cap, req, {p}, 0}*. *cap* and *req* are the capacitive load and required arrival time of this sink.
2. For a source pin point or Hanan point *p*:
   a. If point *p* cannot admit a buffer, then there is only one base solution *sol(p, 0, +∞, {p}, 0}*.
   b. If point *p* can be used to insert a buffer/inverter, there are three base solutions associated with this point. They are (1) *sol1(p, 0, +∞, {p}, 0},* (2) *sol2(p, 0, +∞, {p}, {buffer,0}},* (3) *sol3(p, 0, +∞, {p}, {inverter,0}}*. In *sol2* and *sol3*, a continuous size buffer and inverter of size 0 are inserted at point *p*.

To generate the net topology, a priority queue *priority_sols* is maintained, which returns solution with the largest *req*. This means high priority is given to expand less critical sinks (or partial solutions). All of the base solutions rooted at the sink pins are initially pushed into the *priority_sols.*

3.3 Solution propagation

Once a blockage appears, the solutions cannot grow toward the source node in a greedy, shortest-route manner. All directions for expansion should be tried. A simple method used to try the four directions is maze routing. Each time a solution is popped out from the *priority_sols,* it grows to its neighboring nodes in a wave propagation manner. The maze-routing based method is used in [2] and [4]. However, it is possible for the Hanan grid to be too large for maze routing. As a result, the expansion should stop at some carefully selected nodes, which are called *escape nodes*, instead of simply the neighboring nodes. Line search is an effective and well-known technique used to speed up the maze routing process. Hightower's algorithm [9] is used to find the escape node for each expansion step. Figure 5 shows the algorithm to find the escape nodes during solution tree propagation. Assume that *sol* is the solution popped from *priority_sols*.

Definition - A point is covered by a blockage when a horizontal or a vertical escape line drawn from the point intersects the blockage.

Algorithm *escape_nodes(sol)*

1. *root* = *sol->root*;
2. *hor* and *ver* are the horizontal and vertical escape lines that are drawn from the *root*;
3. node list *escapes*=Φ;
4. for up and down direction (left and right direction) of the two escape lines, find a escape node that is
   a. a source or sink node; or
   b. a Hanan point formed by two lines passing the source or a sink node; or
   c. a Hanan point that is not covered by any blockage that covers *root*; or
   d. the first unblocked Hanan point after the escape line passes a blockage boundary;

   insert the escape node to *escapes*;
5. return *escapes*;

Figure 5. Algorithm escape_nodes

The Hanan graph in Figure 3 is redrawn in Figure 6 with coordinates. Assume that the sink pin s_1 at (a, 4) has a larger required arrival time than that of the sink pin s_2 at (e, 2). The base solution *sol* for s_1 is popped. Since s_1 is on the boundary, it grows in only three directions: up, down, and right to nodes (a, 5), (a, 3), and (d, 4), following rules 3.c, 3.c, and 3.b, respectively. For the base solution rooted at node s_2, it grows to nodes (e, 4), (e, 1), (b, 2), and (f, 2), following rules 3.b, 3.c, 3.d, and 3.c, respectively.

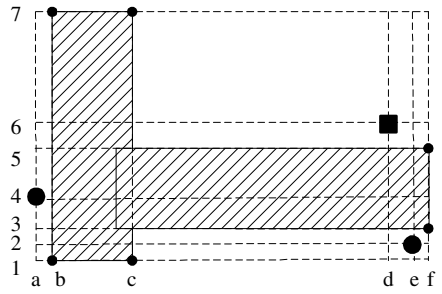

Figure 6. Hanan graph with coordinates

When a solution u expands to an escape node, it is merged with solutions that are rooted there and whose *reachable_set* does not overlap with that of u. This check is necessary because it avoids (1) creating a cycle in the routing tree by connecting a sink pin multiple times and (2) going back to a node that is already in the reachable set, which lead to re-convergence. Suppose a solution $u(root_u, cap_u, req_u, reachable\_set_u, repeater_u)$ merges with solution $v(root_v, cap_v, req_v, reachable\_set_v, repeater_v)$. The new higher-level solution w *is*:

1. when no repeater is inserted at the $root_v$:
   $root_w = root_v$;
   $cap_w = cap_u + c^w_{u,v} + cap_v$; /* where $c^w_{u,v}$ is capacitance of the wire edge between $root_u$ and $root_v$ */
   $req_w = \text{Min}(req_u - d^w_{u,v}, req_v)$; /* where $d^w_{u,v}$ is the delay of the wire edge between $root_u$ and $root_v$ */
   $reachable\_set_w = reachable\_set_u \cup reachable\_set_v \cup$ {nodes on the path from $root_u$ to $root_v$};
   $repeater_w = 0$;
2. when a repeater is inserted at the $root_v$:
   $root_w = root_v$;
   $cap_w = (cap_u + c^w_{u,v})/\beta + cap_v$; /* where β is the fixed gain of the repeater */
   $req_w = \text{Min}(req_u - d^w_{u,v} - d_{repeater}, req_v)$; /* where $d_{repeater}$ is the fixed delay of the buffer/inverter */
   $reachable\_set_w = reachable\_set_u \cup reachable\_set_v \cup$ {nodes on the path from $root_u$ to $root_v$};
   $repeater_w$ = { buffer/inverter, calculated by cap_w and β };

Since we use a gain-based, continuous sizing model, the delay of the buffer/inverter can be fixed no matter how great a capacitive load it will drive. The above two cases may happen at the same node, depending on whether or not the buffers/inverters are inserted there.

All of the new higher-level solutions are pushed to *priority_sols*. After a popped-out solution expands to all the nodes identified by the algorithm *escape_nodes* and merges with all the non-overlapping solutions, another solution is popped. This process is repeated until the *priority_sols* queue is empty. This stopping criterion guarantees the optimal solution is achievable. To speed up the optimization, however, the search can be stopped as soon as a solution that reaches all the sinks is found. This may not be the optimal solution.

3.4 Edge buffering

The length of an edge between two nodes in a Hanan grid may be very large. This makes buffering only at the Hanan grid inadequate. During expansion from a popped solution to an escape node, if the edge between these two nodes is not blocked and very long, inserting buffers/inverters on the edge should be considered. For a given library, the maximum length L that no repeater is needed is easily determined [10]. As a result, if the length of an edge is longer than L, we divide the edge into several small segments. At end of each segment, new solutions for (1) no repeater, (2) buffer, and (3) inverter are constructed. Obviously cases (2) and (3) occur only if the point in question can admit a buffer/inverter. Suppose a solution $u(root_u, cap_u, req_u, reachable\_set_u, repeater_u)$ has been generated at some stage of the process. Solutions v at end point $node_{end}$ of the edge segment are calculated as follows:

1. If $node_{end}$ does not admit a buffer*:*
   $root_v = node_{end}$;
   $cap_v = cap_u + c_{segment}$; //where $c_{segment}$ is the capacitance of the wire segment of length L
   $req_v = req_u - d^w_{u,end}$; //where $d^w_{u,end}$ is the delay of the wire segment between $root_u$ and $node_{end}$
   $reachable\_set_v = reachable\_set_u$;
   $repeater_v = 0$;
2. If $node_{end}$ admits a buffer*:*
   $root_v = node_{end}$;
   $cap_v = (cap_u + c_{segment})/\beta$;
   $req_v = req_u - d^w_{u,end} - d_{repeater}$;
   $reachable\_set_v = reachable\_set_u$;
   $repeater_v$={buffer/inverter, calculated by cap_v and β};

3.5 Pruning

Pruning is common in dynamic-programming based algorithms. The goal of pruning is to reduce the problem complexity and computing time.

Definition - Consider a set of solutions with the same root and driving the same sink pin set. For every $u(root, cap_u, req_u, reachable\_set_u, repeater_u)$ and $v(root, cap_v, req_v, reachable\_set_v, repeater_v)$, if $cap_u > cap_v$ and $req_u \le req_v$, then u is dominated by v. Similarly, if $cap_v > cap_u$ and $req_v \le req_u$, then v is dominated by u.

If u is dominated by v, u is dropped from the solution queue. Notice here that all the solutions that are rooted at the same node and drive the same sink pin set (not the same

reachable set) form a solution set. This pruning does not affect the optimality of the algorithm. In this algorithm, pruning is performed in two cases. The first case is when a solution merges with the solutions rooted at its escape node. The other case is edge buffering. In this case, all the solutions that are rooted at the same segment end point drive the same sink pin set. The number of points kept in the *(cap, req)* curve of a solution set during pruning is restricted. Although dropping some non-dominated points may compromise optimality, the problem size becomes much smaller and the algorithm speed improves greatly. A dynamic bucket-sorting technique is used to ensure that solutions for various ranges of *cap* and *req* are kept.

3.6 Algorithm flow

The flow of this algorithm is presented in Figure 7.

```
1.  topologically sort the net list of circuit in the order from
    primary output to primary input;
2.  for each net net from the above list {
3.  {
4.      H=Hanan_graph(net, B);   //B: blockages
5.      initialize base solutions and priority_sols;
6.      while (priority_sols≠Φ)
7.      {
8.          sol = solution popped from the priority_sols;
9.          node list escape_list= escape_nodes(sol);
10.         for each node escape in escape_list
11.         {
12.             expand sol to escape; /*if needed, do
                  edge buffering and pruning */
13.             merge sol with the solutions rooted at
                  escape; /*if edge is internally
                  buffered, replace sol with a list of
                  solutions rooted at the end point of the
                  last edge segment before escape */
14.             prune solutions rooted at escape with
                  the same sink pin set;
15.             if (new solutions are not dominated by
                  other solutions)
16.                 insert new solutions to the
                      priority_sols;
17.             if (escape==source pin)
18.                 delete from the priority_sols all
                      solutions that drive the same sink
                      pin set as sol;
19.         }
20.     }
21.     best_sol is the best solution rooted at the source
          pin and driving all the sinks;
22.     implement best_sol and map buffers/inverters to
          real gates in the library;
23.     update timing;
24. }
25. return;
```

Figure 7. Algorithm Wire_buffering_with_placement_blockage

Lines 17 and 18 are necessary because they can greatly shorten the algorithm run time. Figure 8 shows the buffered routing solution generated by the algorithm for the example in Figure 3. The solid lines represent the routing. Buffers and inverters are inserted. Suppose the required time of sink (a,4) is much larger than that of sink (e,2). If lines 17 and 18 did not exist, sink (e,2) would not be connected to the source directly until all of the solutions originating from sink (a,4) had been popped out. This would significantly increase the runtime.

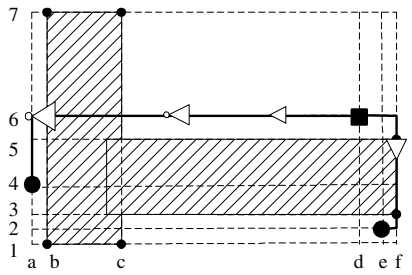

Figure 8. Routing and buffering result of the BRBP algorithm for the example in Figure 1

3.7 Complexity Analysis

Suppose we have an $N \cdot M$ Hanan grid. Assume that at most K solutions are kept after each pruning. At any time, there are up to $2^n \cdot K$ solutions rooted at any node, where n is the number of the sink nodes and 2^n is the number of all possible sink node combinations. During the internal edge buffering, it is possible to create many solutions. However, pruning on these new solutions is performed immediately, and at most K new solutions are kept after the pruning. After the K solutions combine with the solutions rooted at their escape node, the next round of pruning takes place. In addition, edge buffering does not increase the space complexity. Consequently, the worst-case space complexity of this algorithm is $O(N \cdot M \cdot 2^n \cdot K)$.

The time complexity depends on the number of the solutions popped from the *priority_sols* queue. There is a great difference between the number of the solutions pushed into the *priority_sols* queue and the number of valid solutions popped from the *priority_sols* queue because a large number of solutions are pruned later. At most $O(N \cdot M \cdot 2^n \cdot K)$ non-dominated solutions are saved. To construct a solution w from solutions u and v, no edge that is already covered by u or v is taken because this would cause overlap between the reachable sets. A solution will traverse at most $O(N \cdot M)$ edges in the Hanan grid to connect the source pin to a node under the condition of waveform propagation-based maze routing. Consequently, the number of valid solutions is $O(N^2 \cdot M^2 \cdot 2^n \cdot K)$. Because the line-search algorithm can greatly decrease the number of steps, the above analysis is pessimistic. When a solution u is propagated to an escape node, the most time consuming part is the merge operation. Suppose u connects m sink pins; it needs at most $2^{(n-m)} \cdot K$ solutions in order to merge. Thus the time complexity is $O(N^2 \cdot M^2 \cdot 2^{(2n-m)} \cdot K^2)$.

The BRBP algorithm is developed for post global placement. At this stage, fanout optimization [11] has already been performed during the logic synthesis. As a result, any source pin drives a relatively small number of sinks. For example, if fanout optimization is performed so as to limit the maximum fanout count of any source pin to be p, then 2^p is fixed. Under this assumption the worst case space and time complexities of the BRBP algorithm are $O(N \cdot M)$ and $O(N^2 \cdot M^2)$. N and M are usually not large numbers if the number of sinks is small. Note that K is also fixed and is hence dropped out of the *"O"* notation.

4. Experimental results

Table 1. Experimental results

	Cell Number	#Inverters Inserted		#Buffers Inserted	
		Conv.	B+R	Conv.	B+R
ex1	8087	945	900	1349	1240
ex2	38127	5503	5380	7032	6534
ex3	62187	2634	2615	9623	9282
ex4	767982	64384	62830	71238	68204

	Longest Path Delay		Median port Slack		Improvement (%)	
	Conv.	BRBP	Conv.	BRBP	Delay	Slack
ex1	2537	2412	495	504	5.2	1.9
ex2	18153	16793	6784	6933	8.1	2.2
ex3	24672	22718	5204	5345	8.6	2.7
ex4	17948	16391	1420	1452	9.5	2.3

This algorithm for Buffered Routing with Placement Blockages (BRBP algorithm) was implemented and run on several industrial designs. The experimental results are compared with a conventional flow proposed in [1], which first determines the net topology by global routing and then buffers the net. The comparison is made with respect to two metrics. The first is the longest path delay. The second is the median slack time of all the output ports in the designs. The slack of a port is defined as the required arrival time minus the real arrival time of the port. Because[2] is limited to a two-pin net, [4] used some pre-defined buffer locations, and the choice of these locations strongly influences the final results we do not make comparisons with them.

In Table 1, BRBP column denotes the results of the BRBP algorithm. Compared to the conventional flow in [1], nets may go around blockages or go through them. Thus there are fewer buffers and inverters inserted. The results were generated on a distributed computing environment. Hence detailed runtime cannot be collected. The largest circuit was completed in eight hours of CPU time.The same circuit requires nearly 4 hours CPU time for the conventional flow. The reason for this is that much of the CPU time is spent on timing update. Furthermore, our pruning process and speed-up heuristics are quite effective in controlling the time and space complexity. In these experiments, only two solutions are kept after each pruning. One is the solution with the least capacitive load. The other is the one with the latest required arrival time. In the experiments, the number of sinks is no more than 10.

5. Conclusions

In this paper, a dynamic-programming based algorithm to perform buffered wire routing in the presence of placement blockage was presented. The algorithm does not rely on the specification of fixed placement buffer stations. The proposed algorithm was implemented in an industrial design environment and run on several large benchmarks. Experimental results show that this algorithm achieves, on average, a 7.9% improvement on the longest path delay and a 2.3% improvement on the median port slack when compared to the standard industry tool flows.

References

[1] L.P.P.P Van Ginneken, "Buffer Placement in Distributed RC-Tree Networks for Minimal Elmore Delay," in *Proc. IEEE International Symposium on Circuits and Systems,* 1990, pp. 865-868.

[2] H. Zhou. D. F. Wong, I. Liu, and A. Aziz, "Simultaneous Routing and Buffer Insertion with Restriction on Buffer Location," in *Proc. Design Automation Conference*, 1999, pp. 96-99.

[3] C. J. Alpert, "A Steiner Tree Construction for Buffers, Blockages, and Bays," *IEEE Trans. On CAD of Integrated Circuits and Systems*, pp. 556-562, Apr. 2001.

[4] J. Cong and X. Yuan, "Routing Tree Construction Under Fixed Buffer Location," in *Proc. Design Automation Conference*, 2000, pp. 379-384.

[5] I. Sutherland and R. Sproul, "The Theory of Logical Effort: Designing for Speed on the Back of an Envelope," *Advanced Research in VLSI*, Santa Cruz, 1991.

[6] W. C. Elmore, "The Transient Response of Damped Linear Networks with Particular Regard to Wide-band Amplifiers," *Journal of Applied Physics*, pp.55-63, 1948.

[7] M. Hanan, "On Steiner's Problem with Rectilinear Distance," *SIAM Journal of Applied Mathematics*, pp. 255-265, 1966.

[8] D. Kung, "A Fast Fanout Optimization Algorithm for Near-Continuous Buffer Libraries," in *Proc. Design Automation Conference*, 1998, pp. 352-355.

[9] D. W. Hightower, "A Solution to Line-routing Problem on the Continuous Plane," in *Proc. Design Automation Workshop*, 1969, pp. 1-24.

[10] D. Sylvester, C. Hu, O. S. Nakagawa, and S-Y. Oh, "Interconnect Scaling: Signal Integrity and Performance in Future High-speed CMOS Designs," in *Proc. of VLSI symposium on Technology*, 1998, pp.42-43.

[11] H. Touati, "Performance-oriented Technology Mapping," Ph.D. thesis, *University of California, Berkeley, Technical Report UCB.ERL M90/109*, Nov. 1990.