Buffered Routing Tree Construction Under Buffer Placement Blockages

Abstract

Interconnect delay has become a critical factor in determining the performance of integrated circuits. Routing and buffering are powerful means of improving the circuit speed and correcting the timing violations after global placement. This report presents a practical, 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 based routing algorithm is implemented to speed up the process. Some heuristics are used to reduce the problem complexity. These heuristics include limiting the number of intermediate solutions that we keep, using a continuous buffer sizing method, and restricting the buffer locations along the Hanan graph. This algorithm was applied to a number of real industrial designs and achieved an average of 7.9% delay improvement compared to the conventional design flow based on sequential net topology construction followed by buffering.

1 Introduction

Timing optimization has been a challenging problem in deep sub-micron (DSM) designs. With the rapid decrease in device sizes, resistance per unit length in interconnects is rising. At the same time, global wires are lengthening while chip sizes increase. These two factors make interconnect delay play an increasingly important role in determining circuit performance. Many optimization techniques have been developed to reduce interconnect delays. These techniques include, for example, logic restructuring and optimization, wire routing, gate sizing, and buffer insertion. Among these techniques, global routing and buffer sizing stand out as the most effective means of reducing interconnect delay.

Conventional design flow proceeds as follows: first, net topology is determined by constructing a Steiner tree or a shortest path routing tree, then buffers are inserted into this topology and sized. In [1], a dynamic programming-based algorithm for inserting and sizing buffers into a given net topology is proposed. The objective is to maximize the required arrival time at the output pin of the driver of the net. This technique has proven to be quite effective when the inserted buffers can be placed anywhere on the chip layout. However, in reality there are many placement blockages in the circuit, which restrict the areas on the chip where the buffers can be placed. These blockages occur, for example, because of the existence of pre-designed cores and, more generally, macro-cells. Notice that these blockages are placement blockages, not routing ones. Thus, wires (while perhaps not using all possible layers) can go through these blockages. The algorithm of [1] does not perform well in such circumstances, and hence a new algorithm must be developed that takes placement blockages into account. The new algorithm must perform 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.

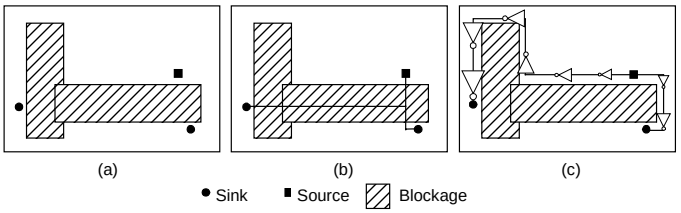

Figure 1. An example of a wire buffer with a placement blockage problem.

A small example is depicted in Figure 1. The source, sink pins, and placement blockages are shown in Figure 1(a). Using a conventional tool flow, the global router, which considers all the placement blockages as routing-available spaces, will construct a Steiner tree net topology as shown in Figure 1(b). The flowing wire buffering tool cannot insert buffers for this net, since it is blocked by the blockage 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 following 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 optimal solution can be achieved by our algorithm, as evidenced in the final result shown in Figure 8.

Furthermore, the ASIC design flow affects the technique used to find the solution as well. In order to improve circuit timing or for many other reasons, buffers are inserted multiple times at different design stages. This post-layout buffer insertion will influence circuit placement and other issues, such as congestion. To deal with these problems, there are some small differences in diverse design flows. In some designs buffer stations, which are blank spaces large enough for several buffers, are distributed throughout the layout area. Throughout the design process buffers are only placed in these buffer stations. As a result, the layout changes are limited to these station areas. In some other designs, there are no buffer stations. The buffers are placed at the optimal locations. Gate overlapping, congestion, and other violations are resolved by the steps that follow the buffer placement. Obviously, the first

method keeps the layout change outside the buffer stations. It reduces the possibility of solution oscillation. However, the buffer station locations are fixed, and the buffer insertion solutions must concede to the location of the stations. Thus, the solution quality may drop with the reduction of the number of buffer station locations.

Recently 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 attempt to find the shortest Elmore delay path between a source pin and a sink pin. They use 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. The authors of [3] searched for a shortest path as well, but they transformed the conventional constrained optimization problem into a cost ratio maximization problem to speed up the process. However, because of the nature of the shortest-path algorithm, these methods only work 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, [4] depends on the assumption that the possible buffer locations are pre-defined. Therefore, it works well in the design flow with buffer stations. However, in practice, for the design flow 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.

In this report we propose a dynamic programming-based algorithm to perform global routing and buffer/inverter insertion and sizing for the design flow without buffer stations. It addresses the multi-pin nets as well and thus absorbs the advantages of both [2] and [4]. Also, pre-defined buffer locations are neither needed nor used. 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 based routing algorithm is used. Although the worst-case complexity of line search is the same as that of maze routing, the average complexity can be reduced greatly. Buffers are inserted not only at the nodes of the graph but also on the long edges of the graph. Some effective heuristics are implemented to simplify the problem. The heuristics include limiting the number of intermediate solutions that we keep, using a continuous buffer sizing method, and restricting the buffer locations along the Hanan graph.

The remainder of the report 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 Generalities

2.1 Problem definition

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

2.2 Buffer/inverter delay model

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 $d=\tau(p+gh)$.[1] p is the parasitic delay of the gate. g is called the *logical effort* of the gate and depends only on the topology of the gate and its ability to produce the output current. g is for the repeaters/buffers. h is called the *electrical effort* (or gain), which is defined as $load/c_{in}$, while c_{in} is the input pin capacitance of the buffer, and *load* is the capacitance load of the buffer. p and g are independent of the buffer sizes, so when h is fixed, the delay of the buffer is also fixed.

2.3 Interconnect delay model

To account for the interconnect delay, the Elmore delay model [6] is used in this report. 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 capacitance load driven by the wire.

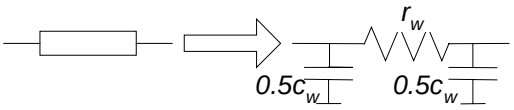

Figure 2. Elmore delay model.

3 Algorithm

Our algorithm is based on dynamic programming. At 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, we merge the existing solutions to obtain a new high-level solution.

[1] 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. For simplicity and without loss of generality, we will drop τ in the discussion that follows.

Buffers/inverters are not only inserted at the nodes of the Hanan graph. The long edges in the graph are divided into small segments, and buffers/inverter are inserted at the end 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 (RSMT) 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]. Since longer wire means longer wire delay, we also make use of the above theorem in this report. We first generate the Hanan grid of a given net, then insert the buffer/inverter and build the net topology simultaneously on this grid. Considering the important effect of the placement blockages, we regard not only the source and sink pins but also the corners of the blockages as Hanan points and build the Hanan graph accordingly. The Hanan graph of the example in Figure 1 is shown in Figure 3. The small solid round dots represent the Hanan points from the corners of the blockages. These points are actually one minimum wire pitch outside the blockages in both horizontal and vertical directions. This means that the Hanan points and the edges between them can be used to insert buffers.

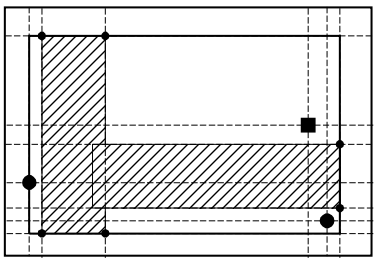

Figure 3. The Hanan graph of the example in Figure 1.

Notice in Figure 3 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 Hanan graph, which in turn decreases the time complexity and memory space requirement of the algorithm. Therefore, only the unblocked corners are used to construct the Hanan graph.

Observation *Only those corners that are not blocked by any blockage are regarded as Hanan points.*

The above observation can save the memory usage and shorten the computation time without degrading the quality of the global routing net topology under construction.

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 case, we need to go around the blockages, so the previous choice is not adequate. The algorithm in Figure 4 shows a method for creating the Hanan graph for a net *net* and blockage set *B*.

```
Algorithm Hanan_graph(net, B)
1. initialize the Hanan_graph H as the
   minimum-bounding box for the source and
   sinks;
2. while there is a blockage b∈B that is cut
   by H, do
3.     enlarge H to cover b completely;
4. return H;
```

Figure 4. The *Hanan_graph* algorithm.

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

A common operation of dynamic programming-based algorithms is to divide the original problem into smaller sub-problems and combine the solutions to lower-level sub-problems to generate new higher-level solutions. In our algorithm, each solution *sol* has a 5-tube labeling (*root, cap, req, reachable_set, repeater*). These labels are defined as follows.

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

cap and *req* are common to most of the previous works on buffer insertion. Since we intend to construct the topology of the net as well, we must keep track of the nodes that are included in each solution. Thus, *reachable_set* is also needed. Pointer *root* points to the node, where the information about the location, type (sink/source/Hanan point), and admissibility of the buffer at that node is stored. *repeater* represents the type and size of the buffer inserted at the *root*.

To perform buffer sizing, a continuous buffer-sizing model is adopted. 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 library size is negligible. (2) In order to perform buffer sizing with a discrete sizing model, each buffer size has to be tried whenever a new buffer is inserted. Moreover, many of the solutions have to be stored for later use during dynamic programming. This results in a

very long computation time and large memory space usage. (3) When using the gain-based delay model [5], the delay is only a function of the gain. As a result, a number of parallel-connected small buffers with the same gain *h* and driven by the same source can be merged into a larger buffer with the same gain *h* [8]. Such an operation preserves the timing. For a fixed gate library, a fixed buffer/inverter gain is pre-defined, and the delay of a buffer/inverter is a fixed value. Therefore, during the bottom-up solution generation stage, all that is calculated for the *repeaters* is their type and size (size is a continuous value). In the end, when the final solution of the whole net is generated, the continuous size buffers/inverters are mapped to real gates in the library according to their assigned size.

We start the process by building the lowest level solutions, base solutions, for each point in the Hanan grid:

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 buffer, then 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 always returns the solution with the largest *req* in it. This means that in our algorithm we give priority to expansion to less critical sinks (or partial solutions). After we create base solutions, all the base solutions rooted at the sink pins are pushed into the *priority_sols.*

3.3 Generating higher level solutions

With the appearance of a blockage, the solutions cannot grow toward the source node in a greedy, shortest-route manner. All of the four directions for expansion (up, down, left, or right) should be tried. A simple method used to try these 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 to be used in practice. Thus 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. We use an idea from the Hightower's algorithm [9] to find the escape node for each expansion step. Figure 5 shows the algorithm we use to find the escape nodes during solution tree growth. *sol* is the solution popped from the *priority_sols.*

Definition *We say 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 lines pass the blockages
      boundary.
   insert the escape node to escapes;
5. return escapes;
```

Figure 5. The *escape_nodes* algorithm.

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 first. 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.

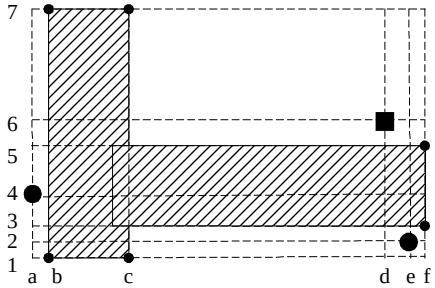

Figure 6. Hanan graph with coordinates.

When a solution *u* expands to an escape node, it merges with all the solutions that are rooted there and whose *reachable_set* does not overlap with that of *u*. This check is necessary because both source/sink nodes and Hanan points that are reachable in a solution are included in its *reachable_set*. Therefore, the above condition avoids (1) creating a cycle in the routing tree by connecting to a sink pin multiple times and (2) going back to a node that is already in the reachable set, which may lead to convergence problems. 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. $repeater_v$=0: 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 the capacitance of the wire edge between $root_u$ and $root_v$ */

   req_w=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 $root_v$ */

   $reachable\_set_w$=$reachable\_set_u$∪$reachable\_set_v$∪{nodes on the path from $root_u$ to $root_v$};

   $repeater_w$=0;

2. $repeater_v$> 0: 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=Min(req_u - $d^w_{u,v}$ - $d_{repeater}$, req_v); /*where $d_{repeater}$ is the fixed delay of the buffer/inverter */

   $reachable\_set_w$=$reachable\_set_u$∪$reachable\_set_v$∪{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 be subjected to.

The above two cases may happen at the same node, depending on whether or not the buffers/inverters are inserted there. [2]

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

3.4 Edge buffering

Because of large chip size, the length of an edge between two nodes in the Hanan grid should be very large. This makes buffering only at the Hanan grid node inadequate. During expansion from a popped node to an escape node, we should consider inserting buffers/inverters on the edge between these two nodes. For a given library designed in a particular process technology, the maximum length L that there is not a repeater to be inserted is easily determined [10]. As a result, if the length of an edge is longer than L, we divide the edge into several smaller 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. The solutions v at the 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 the pruning is to reduce the problem complexity and speed up the algorithm.

Definition *Consider a set of solutions with the same root and that drive the same sink pin set. If there is* $u(root, cap_u, req_u, reachable\_set_u, repeater_u)$ *and* $v(root, cap_v, req_v, reachable\_set_v, repeater_v)$,*and* $cap_u > cap_v$, *and* $req_u \le req_v$, *or* $cap_u < cap_v$, *and* $req_u \ge req_v$ *then we say u is dominated by v or 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*) are processed. This condition can help to reduce the memory space and computing complexity. This pruning does not affect the optimality of the algorithm. In our algorithm, pruning is performed in two places. The first place is when a solution is merged with the solutions rooted at its escape node. This is different from many of the previous works where pruning is performed only for the new solutions that are created from identical lower-level solutions. Here many existing solutions may drive the same sink pin set in their *reachable_set* and have the same *root* as those of the newly created solutions. As a result, pruning must be performed with respect to both the old solutions and the new ones. The other place where we do pruning is during the internal edge buffering step. In this case, all the solutions that are rooted at end points of the edge segments drive the same sink pin set. The pruning is done on the solutions that are rooted at the same segment end.

[2] There are three possibilities: no repeater, a buffer, or an inverter.

In a large design, even after the above pruning, the time complexity and memory space required may still be very high. To address this problem, we control the number of points kept in the *(cap, req)* curve of a solution set during the expansion process. Although we lose optimality by dropping some non-dominated points in order to limit the number of stored points to a small value, the problem size becomes much smaller and our algorithm speed improves greatly. We use a *dynamic bucket sorting technique* to make sure that we keep representative solutions for the various ranges of the *cap* and the *req*. We limit the number of distinctive points in any solution curve to be less than or equal to a user-defined constant *P*. Two solutions are considered the same if the *(cap, req)* difference between them is smaller than some delta value. At the beginning, a very small delta value is given to compare two adjacent solutions. Therefore, very few solutions will be dropped. This delta value is then gradually increased until the number of distinctive solutions is less than or equal to *P*.

3.6 Algorithm flow

The flow of this algorithm is presented in Figure 7.

```
Algorithm Wire_buffering_with_placement_blockage(net_list)
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 escapet_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. The *Wire_buffering_with_placement_blockage* algorithm.

Line 17 and 18 are necessary because they can greatly shorten the algorithm run time. Figure 8 shows the buffered routing solution generated by our algorithm for the example in figure 3. The solid lines represent the routing. Buffers and inverters are inserted into the wiring solution. 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, then our algorithm would generate a solution that connects source (d, 6) to sink (e, 2), using a route similar to one in Figure 8. However, it would have connected to sink (a, 4) by going from (e, 2) to (a, 2) and from (a, 2) to (a, 4). Eventually (that is if we continue solution generation until the *priority_sols* comes empty), we would generate the optimal solution, which is the one depicted in Figure 8. This would, however, significantly increase our runtime. Line 17 and 18 of Figure 7 ensure that we would expand the solution from sink (e, 2) to the source directly after the sink (a, 4) reaches the source via a solution.

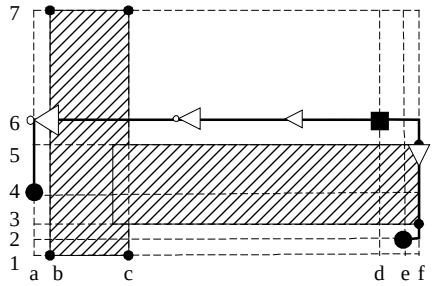

Figure 8. Routing and buffering result of our algorithm for the example in Figure 1.

3.7 Complexity Analysis

Suppose we have an $N \cdot M$ Hanan grid. Notice that pruning is performed on all the solutions that have the same root node and the same sink node set, not on the solutions with the same root node and the same reachable set. 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 groupings. 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 other words, the edge buffering does not increase the space complexity. Therefore, 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 big 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. A solution needs to traverse at most $O(N \cdot M)$ edges in the Hanan grid to connect the source pin to the sinks under the condition of waveform propagation-based maze routing. In other words, we never process an edge in the Hanan graph more than once. We keep at most $O(N \cdot M \cdot 2^n \cdot K)$ non-dominated solutions. To construct a solution w from solutions u and v, we will not use an edge that is already used by u or v because this would cause overlap between the reachable sets. Thus, a new non-dominated solution will inherit all the traversed edges from its child solutions. Consequently, the number of the 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; the most solutions it needs to merge is $2^{(n-m)} \cdot K$. Thus the time complexity for solution growth is $O(N^2 \cdot M^2 \cdot 2^{(2n-m)} \cdot K^2)$.

This algorithm is intended for use after the global placement step. At this stage, the fanout optimization [11] has already been performed during the logic synthesis. As a result, the source pins drive a relatively small number of sinks. For example, if the fanout optimization algorithm is written so as to limit the maximum fanout count of any source pin to, for example, p, then 2^p is fixed and small. Under this assumption the worst case space and time complexities of our dynamic programming-based algorithm are $O(N \cdot M)$ and $O(N^2 \cdot M^2)$. Note that K is also fixed and is hence dropped out of the "O" notation.

4 Results

This algorithm was implemented and run on several industrial circuits. We compared the results with a conventional flow V.G., which first determines the net topology by global routing and then buffers the net by the algorithm used in [1]. 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. Table 1 shows the results. Because the authors of [3] and [4] all use some pre-defined buffer locations and the choice of these locations strongly influences the final results and [2] performs two-pin net routing, we do not make comparisons with them. These circuits are real designs from major ASIC companies. Notice that the largest design has more than 750K gates.

Table 1. Experimental results.

	Cell Number	Inverters Inserted		Buffers Inserted	
		V.G.	B+R	V.G.	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 (%)	
	V.G.	B+R	V.G.	B+R	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

There is no limitation on the topology of the net produced by our algorithm. The net tree growth is completely driven by the required arrival time. In these experiments, we only keep two solutions 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.

In Table 1 B+R denotes the results of our method. Compared to the method used in [1], nets may go around blockages or go through them. Thus there are fewer buffers and inverters inserted than when using the algorithm of [1]. The results were generated on a distributed computing environment and hence detailed runtime cannot be collected. As the table shows clearly, we completed the largest circuit in eight hours of CPU time. The same circuit requires nearly 4 hours CPU time for the conventional flow(V.G.). The reason for this is that much of the CPU time for both the conversional flow and our flow is spent on timing update. Furthermore, our pruning process and speed-up heuristics are quite effective in controlling the time and space complexity of our proposed dynamic programming-based algorithm.

5 Conclusions

In this report, we presented a dynamic programming-based algorithm to perform wire buffering with placement blockage. The algorithm does not rely on the specification of fixed

placement buffer stations. To decrease the problem complexity, several optimality-preserving pruning rules and some pruning heuristics were proposed to search the solution space. The proposed algorithm was implemented in an industrial design environment and run on several large benchmarks. Experimental results show that our 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.

Reference

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

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

[3] A. Jagannathan, S. Hur, and J. Lilis, "A Fast Algorithm for Context-aware Buffer Insertion," Proc. Design Automation Conference, pp. 368-373, 2000.

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

[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," Proc. Design Automation Conference, pp. 352-355, June 1998.

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

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

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