Chapter 28: Buffering in the Layout Environment

Jiang Hu,* and C. N. Sze†

1 Introduction

Chapters 26 and 27 presented buffering algorithms where the buffering problem was isolated from the general problem of timing closure. The main problem with this extraction is that there are not necessarily resources available to put the buffers or wires in their desired locations. One way to manage this flow is to put the buffers in their ideal locations and allow a legalization procedure to move them to actual locations. The problem with this approach is that buffers may be moved quite far from their ideal locations, which could completely corrupt the quality of the solution. It is much better to place the buffers in regions where appear to be sufficient space, so that legalization would move the buffers by at most 10-20 routing tracks, which would preserve the original solution. To do this, one must certainly take into account the blockages and preferably local placement and routing congestion. In this chapter, we explore techniques which consider these factors.

2 Placement and Routing Blockages

In realistic chip designs, some regions may be occupied by IP blocks, memory arrays and macros. Such regions allow wires to pass through but have no room for buffer insertion. Therefore, buffer insertion has to be performed with consideration of these buffer blockages. If a wire path has large overlap with blockages as in Figure 28.1(a), no feasible buffering solution can be found. However,

*Department of Electrical and Computer Engineering, Texas A&M University, TX 77843.
†IBM Austin Research Laboratory, Austin, TX 78758.
avoiding buffer blockages completely as Figure 28.1(b) may cause unnecessary wire detour as well as delay degradation. Hence, it is important to have algorithmic techniques which can find a proper routing topology (as Figure 28.1(c)) together with the buffer insertion solution. Another more intuitive example was previously described in Figure 27.4 of Chapter 27, in order to find the best buffered interconnect solution, the routing must consider feasible buffer insertion locations.

![Figure 28.1: Example of buffer blockage.](image)

A similar problem is buffer insertion considering placement and routing congestions. If a buffer is inserted in a crowded place as in Figure 28.2(a), it might be moved far away later during cell placement legalization. Thus, such insertion is not favorable although it is feasible, i.e., buffers are preferred to be inserted in relatively sparse regions like Figure 28.2(b). Similarly, buffer insertion in a routing congested region may intensify the wire routability problem. Hence, buffer insertion algorithms need to be aware of layout environment and be able to handle the trade-off between timing performance and congestion avoidance.

Section 3 will introduce algorithms on blockage avoidance for 2-pin nets, i.e., buffered paths. Blockage buffered Steiner tree methods for multi-pin nets will be described in Section 4. In Section 5, techniques for handling congestions will be discussed.

### 3 Buffered Path with Blockage Avoidance

For 2-pin nets, the problem is to find a buffered path with the minimum delay under the blockage constraints. In the literature, there exists two major approaches: (1) dynamic programming and
Both are based on the Elmore delay model. It have not been showed that these approaches are fast enough to be practical for general optimization cases, but they can be very useful to a handful of the most critical nets.

3.1 Dynamic Programming Based Method

The dynamic programming based method [1, 2] propagates partial solutions from the sink node $t$ through a routing graph $G = (V, E)$ and picks the optimal solution at the source node $s$. The routing graph can be either a uniform grid reflecting routing tracks (Figure 28.3(a)) or an extended Hanan grid which is obtained by drawing vertical and horizontal lines through given pins and blockage boundaries (Figure 28.3(b)). For each edge $(u, v) \in E$, $R(u, v)$ and $C(u, v)$ are the edge resistance and capacitance, respectively. For each node $v \in V$, there is a label $p(v) \in \{0, 1\}$ which is equal to 0 if it overlaps with a buffer blockage and equal to 1, otherwise. Besides the routing graph, the driver resistance $R_d$, sink capacitance $C_t$ and a buffer library $B$ are assumed to be given. Each buffer type $b \in B$ is modeled by its input capacitance $C(b)$, intrinsic delay $K(b)$ and output resistance $R(b)$.

A partial solution at a node $v$ is characterized by a quadruple $\alpha = (c, d, m, v)$, where $c$ is the current input capacitance seen at $v$, $d$ is the delay from $v$ to the sink $t$ and $m$ is a labeling for the buffered path from $v$ to $t$. The label of $m(v) = b$ indicates that buffer $b \in B$ is inserted at node $v$ and $m(v) = 0$ implies that no buffer is inserted there. The solution $\alpha_1 = (c_1, d_1, m_1, v)$ is inferior
to solution \( \alpha_2 = (c_2, d_2, m_2, v) \) if \( c_1 \geq c_2 \) and \( d_1 \geq d_2 \).

The partial solutions are maintained in a priority queue \( Q \) initialized with the solution \((C_t, 0, 0, t)\) at the sink \( t \). Each time, the top solution in \( Q \), which has the minimum delay, is extracted for expansion. A solution \((c, d, m, u)\) is expanded to its neighbor node \( v \) if there is an edge \((u, v) \in E\). The expanded solution is \((c + C(u, v), d + R(u, v) \cdot (c + C(u, v)/2), m, v)\) where the delay increase is based on the Elmore delay model. If a solution \((c, d, m, v)\) is at node \( v \) where \( m(v) = 0 \) and \( p(v) = 1 \), buffers of each type are inserted there to generate new partial solutions. If the buffer type is \( b \in B \), its corresponding buffered solution is \((C(b), d + R(b) \cdot c + K(b), m, v)\) with \( m(v) = b \). If a solution reaches the source node as \((c, d, m, s)\), the driver is added by updating the solution as \((0, d + R_d \cdot c, m, s)\). When a solution with the driver is at the top of the \( Q \), it is the minimum delay solution.

The pseudo code of this algorithm is given in Figure 28.4. Please note that pruning is performed in many steps to remove inferior solutions so that the runtime can be improved. The complexity of this algorithm is \( O(|B||V|(|E| + |B||V|) \log |B||V|) \) [1].
### Algorithm: Buffered Path \((G, B, s, t)\)

<table>
<thead>
<tr>
<th>Input:</th>
<th>Routing graph (G = (V, E)), Buffer library (B), Source node (s \in V) and sink node (t \in V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>Buffered path labeling (m)</td>
</tr>
</tbody>
</table>

1. \(Q \leftarrow \{(C, 0, 0, t)\}\)  
2. while \(Q \neq \emptyset\) do  
3. \((c, d, m, u) \leftarrow \text{extract}_\text{min}(Q)\)  
4. if \(c = 0\), return \(m\)  
5. if \(u = s\), push \((0, d + R_d \cdot c, m, s)\) into \(Q\) and prune continue  
6. for each \((u, v) \in E\) do  
   \(d' \leftarrow d + R(u, v) \cdot (c + C(u, v)/2)\)  
   push \((c + C(u, v), d', m, v)\) into \(Q\) and prune  
7. if \(p(u) = 1\) and \(m(u) = 0\)  
8. for each \(b \in B\) do  
   \(d' \leftarrow d + R(b) \cdot c + K(b)\)  
   \(m(u) = b\)  
   push \((C(b), d', m, u)\) into \(Q\) and prune

Figure 28.4: Pseudo code of the dynamic programming based buffered path algorithm.

### 3.2 Graph Based Approach

The graph based approaches [3, 4] first transform the routing graph \(G = (V, E)\) into a buffer graph \(G_B = (V_B, E_B)\) and then obtain the minimum delay buffered path by the Dijkstra’s shortest path algorithm.

![Buffer graph](image)

Figure 28.5: Buffer graph.

The node set \(V_B\) of the buffer graph is composed by the source node, sink nodes and a set of
buffer nodes. A buffer node always has a buffer inserted and therefore it has to be out of any buffer blockage. An edge $e \in E_B$ is usually directed from the source or a buffer node to a buffer node or the sink node (Figure 28.5(a)). There is a delay associated with each edge. If the Elmore delay model is employed, the delay $d(u, v)$ for edge $(u, v)$ is equal to $R(u) \cdot (C(u, v) + C(v)) + R(u, v) \cdot (C(u, v)/2 + C(v))$ where $R(u)$ is the driving resistance at $u$, $R(u, v)$ is the edge resistance, $C(u, v)$ is the edge capacitance and $C(v)$ is the input capacitance at $v$. Unlike the routing graph introduced in Section 3.1, the two end nodes of an edge in the buffer graph do not have to be geometrical neighbors. An example of buffer graph is shown in Figure 28.5(b). If the edge delay is treated as edge weight, the minimum buffered path is equivalent to the shortest path on this buffer graph. Thus, the optimal solution can be found easily by applying Dijkstra’s shortest path algorithm on the buffer graph.

The graph based approach can be easily extended to handle multiple buffer types and wire sizing. If there are $k$ buffer types, each buffer node is split into $k$ copies each of which corresponds to one type. Edges are inserted among these copies of nodes. In Figure 28.5(c), an example of two buffer types is shown. Similarly, if there are multiple wire widths, the edge weight is chosen as the minimum edge delay among all options of wire widths [4]. Then, discrete wire sizing is naturally handled in the same framework.

Besides minimizing the path delay, the problem can be formulated [3] as maximizing delay reduction to cost ratio $D_{ref} - \frac{d(p)}{g(p)}$ where $D_{ref}$ is a reference delay, $d(p)$ is the path delay of path $p$ and $g(p)$ is the path cost. The path cost is simply the total edge cost along the path. The edge cost can be defined in many different ways. For example, it can be the summation of wire capacitance and buffer/sink capacitance of its downstream end. Let $R_{max}$ represent the maximum ratio can be obtained. Then

$$R_{max} = \frac{D_{ref} - \sum_{e \in p} d(e)}{\sum_{e \in p} g(e)}$$

or equivalently

$$R_{max} \sum_{e \in p} g(e) + \sum_{e \in p} d(e) = D_{ref}.$$
If the weight of each edge $e$ is set to $R_{\text{max}} g(e) + d(e)$, the total path weight is equal to $D_{\text{ref}}$. The value of $R_{\text{max}}$ can be obtained by probing different values in a binary search manner. For a guess $I$ of $R_{\text{max}}$, the shortest path weight is obtained when each edge weight is labeled as $I \cdot g(e) + d(e)$. If the result is greater (smaller) than $D_{\text{ref}}$, the value of $I$ is increased (decreased). When the path weight is sufficiently close to $D_{\text{ref}}$, its corresponding value of $I$ can be treated as $R_{\text{max}}$.

![Figure 28.6: All circles represent the set of non-inferior solutions. The dark circles lie on the lower convex hull.](image)

As the value of $D_{\text{ref}}$ decreases, the cost of the corresponding maximum ratio path increases [3]. There exists a $D_{\text{ref}}$ for which a $(g, d)$ path is optimal if and only if $(g, d)$ lies on the lower convex hull of the trade-off curve between cost $g$ and delay $d$ [3]. The solutions on a lower convex hull is illustrated in Figure 28.6.

4 Buffered Tree with Blockage Avoidance

The situation of multi-pin nets is much more difficult than that of 2-pin nets as Steiner tree construction itself is a hard problem. There are two categories of approaches: (1) constructing a Steiner tree regardless of buffer blockages and then adjusting the tree to avoid blockages; (2) simultaneous Steiner tree construction and buffer insertion with awareness of blockages.
### 4.1 Tree Adjustment Technique

As a relatively easy method, one can start with a Steiner tree regardless blockages and modify the tree to avoid blockages [5]. This can be performed in a fashion similar to the rip-up-and-reroute in congestion avoidance of global routing. In other words, if a path in the tree has large overlap with blockages, it is ripped up and reconnected back to the tree with a path having less overlap with blockages. This is illustrated in Figure 28.7.

![Figure 28.7: Rip-up-and-reroute to avoid blockages.](image)

In each iteration, the path with the largest overlap with blockages is chosen for rerouting. The reconnection procedure is done by running Dijkstra’s algorithm on the extended Hanan grid graph indicated by the dashed lines in Figure 28.7. In this graph, the weight of an edge is its length if it does not overlap with any blockage. If an edge overlaps with a blockage, its weight is its length times $\alpha$, where $\alpha > 1$ is a penalty coefficient. The value of $\alpha$ decides the trade-off between blockage avoidance and wire increase due to detour. After the tree modification, the chance of feasible buffering solutions is increased. Since the rerouting has no knowledge if buffers are needed on a path, it may cause some unnecessary wire detours.

Another technique is to integrate the tree adjustment with buffer insertion [6] so that wire detour is incurred only when it is necessary for buffer insertion. The classic van Ginneken’s buffer insertion algorithm [7] propagates a set of candidate solutions from the sink nodes toward the source and picks the optimal one at the source. The adaptive tree adjustment technique generates a candidate solution with an alternative Steiner node if the original Steiner node is inside a blockage. This adjustment is a part of a candidate solution which is propagated toward the source. This
adjustment is adopted only when its corresponding candidate solution is selected at the source. In other words, the tree adjustment is made according to the need of buffer insertion. In Figure 28.8, an example is depicted to demonstrate this technique.

4.2 Simultaneous Tree Construction and Buffer Insertion

The problem of whether or not to avoid a blockage and how to avoid can be solved by simultaneously constructing Steiner tree and inserting buffers [8–10]. Compared to the tree adjustment techniques, the simultaneous approach can lead to improved solution quality with increased computation cost. There are two major methods of the simultaneous approach: dynamic programming [8] and graph based [9].

4.2.1 Dynamic Programming Based Method

The dynamic programming based method, called RMP (Recursive Merging and Pruning) is performed on a routing graph like Figure 28.3. Similar to the fast path algorithm [1], it propagates candidate solutions over the graph. The difference is that RMP considers merging solutions to form subtrees. Each candidate solution is characterized by \((c, q, RE, buf, v)\) where \(c\) is the downstream load capacitance, \(q\) is the required arrival time, \(RE\) is the reachable sink set in the subtree and \(v\) is
the root of the subtree. The label \( \text{buf} = 1 \) if a buffer is inserted at the node, \( \text{buf} = 0 \) otherwise. The candidate solutions are maintained in a priority queue with the maximum-q solution on the top.

When merging two solutions at a node, one need to ensure that the reachable sink sets of the two solutions are disjoint. If a sink appears in both of the solutions, then the merging implies non-tree topology. If solution \((c_1, q_1, RE_1, \text{buf}1, v)\) and \((c_2, q_2, RE_2, \text{buf}2, v)\) are merged, the merged solution is \((c_1 + c_2, \min(q_1, q_2), RE_1 \cup RE_2, 0, v)\). For solutions at the same node of the routing graph, a pruning can be performed among them if they all have the same reachable sink set. The pruning is same as that described in Section 3.1. The RMP algorithm can reach the optimal solution in exponential time. In order to reduce runtime, one can perform an aggressive pruning which keeps only the minimum-c solution for each reachable sink set [8]. This technique can improve runtime significantly with very limited sacrifice on solution quality.

4.2.2 Graph Based Technique

The graph based method [9] starts with constructing a look-up table storing pre-computed tree components. Then, an abstraction graph is generated with each edge corresponding a tree component which can be obtained from the look-up table. The buffered tree with minimized maximum sink delay is obtained by applying Dijkstra’s shortest path algorithm on the abstraction graph.

The tree components include:

- Wire path: a path connecting two nodes in the routing graph by properly sized wires but no buffers is between them.

- Buffered path: a path connecting two nodes in the routing graph with buffers inserted between.

- Buffer combination: a tree component connecting three or more nodes in the routing graph without internal buffers.

- BC-subtree: a subtree rooted with a buffer combination.
These components are illustrated in Figure 28.9. The minimum delay buffered path can be obtained by the method of [3, 4] which is introduced in Section 3.2. A buffer combination can be treated as an unbuffered Steiner tree. Its delay is specified as the maximum root-leaf delay. If the number of nodes is restricted, the minimum delay buffer combination can be obtained by enumeration. Both the minimum delay buffered paths and the minimum delay buffer combinations are saved in look-up tables for future query.

Based on buffer combinations, BC-subtrees, which are subtrees rooted at a buffer combination, can be constructed to drive a set of sinks. A few examples of BC-subtrees are shown in Figure 28.10.

A buffered Steiner tree (or subtree) is composed by a set of buffered paths and BC-subtrees in
general. Therefore, a general problem is how to construct a buffered tree (or subtree) that drives a
certain set of sinks $\Gamma = \{s_1, s_2, \ldots\}$. This is achieved by using an abstraction graph $G_\Gamma$ illustrated
in Figure 28.11. This graph consists of a source node, which is the set of sinks $\Gamma$, and a set of
possible buffer nodes. An edge $(\Gamma, v)$ represents the optimal BC-subtree rooted at $v$, and its weight
is the maximum delay of the BC-tree. The edge $(u, v)$, where $u, v \not\in \Gamma$, represents the optimal
buffered path between $u$ and $v$ which can be found in the look-up table. Then, the shortest path
from $\Gamma$ to each other node $v$ corresponds to the optimal subtree connecting to the sink set $\Gamma$. The
algorithm proceeds to create subtrees by increasingly considering more sinks.

This algorithm can minimize the maximum source-sink delay, but not the timing slack. In fact,
it can reach the optimal solution in exponential time.

5 Layout Environment Aware Buffered Steiner Tree

The previous sections present different algorithms buffer insertion and buffered tree construction
avoiding buffer placement blockages. Practically, it is essential that buffer insertion algorithms
consider layout environment such as the placement and routing congestion, which obviously leads
to a more complicated problem. In this section, we start with the congestion assessment and then
introduce several related algorithms.
5.1 Measurement of Placement and Routing Congestion

In order to evaluate the placement and routing congestion of a buffered net, a tile graph is usually used to capture the congestion information and at the same time reduce the problem complexity. The tile graph is represented as $G = (V_G, E_G)$ such that $V_G = \{g_1, g_2, \ldots\}$ is a set of tiles and $E_G$ is a set of boundaries each $(g_i, g_j)$ of which is between two adjacent tiles $g_i$ and $g_j$. An example of the tile graph is shown in Figure 28.12. If a tile $g_i \in V_G$ has an area of $A(g_i)$ and its area occupied by placed cells are $a(g_i)$, the placement density is defined as $d(g_i) = \frac{a(g_i)}{A(g_i)}$. Let $W(g_i, g_j)$ be the maximum number of wire tracks that can be routed across the tile boundary $(g_i, g_j)$ and $w(g_i, g_j)$ be the number of used tracks crossing $(g_i, g_j)$. Similarly, the boundary density is $d(g_i, g_j) = \frac{w(g_i, g_j)}{W(g_i, g_j)}$.

In order to increase the penalty of using a congested tile (similarly for a tile boundary), using square cost (i.e., $d(g_i)^2$) ensure the cost increases more rapidly as a tile is closer to becoming full. For example, the cost of using two tiles with densities of 0.1 and 0.9 is 0.82, while the cost of using two tiles with densities of 0.5 is 0.5. When considering both the placement and routing congestion cost for a net, the total cost incurred can be a linear expression of squares of both the tile densities and boundary densities.

Figure 28.12: An example of tile graph.
5.2 Plate Based Tree Adjustment

When we consider both placement and routing congestions at the same time, applying simultaneous Steiner tree construction and buffer insertion seems to be computationally prohibitive for practical circuit designs. In the following, sequential approaches [11, 12] are introduced to solve the problems. A good way to handle the placement and routing congestion is through the following four stages: (1) Timing-driven Steiner tree construction; (2) Plate-based adjustment for congestion mitigation; (3) Local blockage avoidance (refer to Section 2.1); (4) Van Ginneken style buffer insertion. Since stages 1, 3 and 4 have been described in previous sections, the rest of the discussion focuses on the stage of plate based tree adjustment.

5.2.1 Dynamic Programming Based Adjustment

The basic idea for the plate-based adjustment [13] is to perform a simplified simultaneous buffer insertion and local tree adjustment so that the Steiner nodes and wiring paths can be moved to less congested regions without significant disturbance on the timing performance obtained in Stage 1. The plate-based adjustment traverses the given Steiner topology in a bottom-up fashion by the dynamic programming algorithm. During this process, Steiner nodes and wiring paths may be adjusted together with buffer insertion to generate multiple candidate solutions. We only use buffer insertion to estimate the placement congestion of the buffered tree and to guide the tree adjustment. Hence, the output of this stage is still an unbuffered net, only with changes in the Steiner tree routing. Besides, since buffer insertion is merely a mean of placement congestion estimation, a single “typical” buffer type can be used to simplify the calculation, while the Elmore delay model can be used for interconnect and a switch level RC gate delay model is adopted.

For a Steiner node $v_i$ which is located in a tile $g_k$, a plate $P(v_i)$ for $v_i$ is a set of tiles in the neighborhood of $g_k$ including $g_k$ itself. During the plate-based adjustment, we confine the location change for each Steiner node within its corresponding plate. If $v_i$ is a sink or the source node we set $P(v_i) = \{g_k\}$. The shaded box in Figure 28.13(a) gives an example of the plate corresponding to Steiner node $v_4$. The plate indicates any of the possible locations which the Steiner node may
Figure 28.13: (a) Candidate solutions are generated from \( v_2 \) and \( v_3 \) and propagated to every tile, which is shaded, in the plate for \( v_4 \). Solution search is limited to the bounding boxes indicated by the thickened dashed lines. (b) Solutions from \( v_1 \) and every tile in the plate for \( v_4 \) are propagated to the plate for \( v_5 \). (c) Solutions from plate of \( v_5 \) are propagated to the source and the thin solid lines indicate one of the alternative trees that may result from this process.

be moved to.

The search for alternative wiring paths is limited to the minimum bounding box covering the plates of two end nodes. In Figure 28.13, such bounding boxes are indicated by the thickened dashed lines. Therefore, the size of plates define the search range for both Steiner nodes and wiring paths. As a result, the size of the plate controls the quality of solution/runtime trade-off desired by the user. With different plate size, we can obtain the ability to modify the topology to move Steiner points into low-congestion regions while also capping the runtime penalty. An example of how a new Steiner topology might be constructed from an existing topology is demonstrated in Figures 28.13(a)-(c).

It is suggested in [14] that buffer insertion can be performed in a simple non-timing-driven way by following a rule of thumb: the maximal interval between two neighboring buffers is no greater than certain upper bound. Similarly, we restrict the maximum load capacitance \( U \) a buffer/driver may drive, so that sink/buffer capacitance can be incorporated. To keep the succinctness of the tile-based interval metric in [14], we discretize the load capacitance in units equivalent to the capacitance of wire with average tile size. Thus, we can prune out all intermediate solutions with load capacitance greater than \( U \).
During the bottom-up process, each intermediate solution is characterized by a 3-tuple \(s(v_i, c, w)\) in which \(v_i\) is the root of the subtree, \(c\) is the discretized load capacitance seen from \(v_i\), and \(w\) is the accumulated congestion cost. A solution can be pruned if both its \(c\) and \(w\) are no better than another one in the solution set associated with the same node \(v_i\).

Starting from the leaf nodes, candidate solutions are generated and propagated toward the source in a bottom-up manner. Before we propagate candidate solutions from node \(v_i\) to its parent node \(v_j\), we first find both plate \(P(v_i)\) and plate \(P(v_j)\) and define a bounding box which is the minimum sized array of tiles covering both \(P(v_i)\) and \(P(v_j)\). Then we propagate all the candidate solutions from each tile of \(P(v_i)\) to each tile of \(P(v_j)\) within this bounding box. Since the Steiner nodes are more likely to be buffer sites due to the demand on decoupling non-critical branch load from the critical path, allowing Steiner nodes to be moved to less congested area is especially important. Moreover, such move is a part of a candidate solution, and therefore the move will be committed only when its corresponding candidate solution is finally selected at the driver. Thus, the tree adjustment is dynamically generated and selected according to the request of the final minimal congestion cost solution.

### 5.2.2 Hybrid Approach for Tree Adjustment

Although the stage of plate based tree adjustment effectively improve the layout congestion issue, there are several techniques [12] to improve the computational efficiency. Actually, the runtime bottleneck is due to the fact that buffering solution has to be searched along with node-to-node\(^1\) paths in a two-dimensional plane since low congestion paths have to be found at where the buffers are needed.

If we can predict where buffers are needed in advance, then we can merely focus on searching low congestion paths and the number of factors to be considered can be further reduced to one. If we diagnose the mechanism on how buffer insertion improves interconnect timing performance, it can be broken down into two parts: (1) regenerating signal level to increase driving capability

\(^1\)The node may be the source node, a sink node or a Steiner node of degree greater than two. Thus, degree-2 Steiner nodes are not included here.
for long wires and (2) shielding capacitive load at non-critical branches from the timing critical path. In a Steiner tree, buffers that play the first role are along a node-to-node path while buffers for the second purpose are normally close to a branching Steiner node. The majority of buffer insertion algorithms such as van Ginneken’s algorithm are dynamic programming based and have been proved to be very effective for both purposes. However, optimal buffer solutions along a node-to-node path can be found analytically [15, 16]. This fact suggests that we may have a hybrid approach in which buffers along paths are placed according to the closed form solutions while the buffers at Steiner nodes are still solved by dynamic programming, i.e., analytical buffered path solutions replace both the wire segmenting [15] and candidate solution generations at segmenting points in the bottom-up dynamic programming framework. Computing candidate buffered paths analytically is faster than applying dynamic programming, which makes this hybrid approach more efficient than the purely dynamic programming scheme.

It is also suggested in [12] that the plate should be selected as a set of nearby tiles with the least congestion because only the nearby tiles with relatively low buffer placement or routing congestion cost worth considering to be the alternative Steiner node. In fact, if there exists a tile with high congestion cost in the plate, it will never be used as the new Steiner node.

Instead of using a length based buffer insertion, the algorithm uses analytical formula for buffer insertion which is separated from the minimum congestion cost path search process. If given the driver resistance, sink loading capacitance and buffer resistance/capacitance/intrinsic delay, the optimal number of buffers and corresponding placement locations can be found with the equations [15], which are previously described in Section 2.1 of Chapter 26.

We explain our buffered path routing technique by an example. For the thickened path in Figure 28.14(a), if we know the driving resistance at $v_1$ and load capacitance at $v_4$, we may obtain the optimal buffer positions at $v_2$ and $v_3$. However, if we connect $v_1$ and $v_4$ in a two-dimensional plane, there are many alternative paths between them and the optimal buffer locations form rows along diagonal directions. The tiles for the optimal buffer locations are shaded in Figure 28.14(a). Therefore, if we connect $v_1$ and $v_4$ with any monotone path and insert a buffer whenever this path
Figure 28.14: Find low congestion path with known buffer positions indicated by the shaded tiles. If a path passes through a shaded tile, the resulting buffered path should have the same minimum delay. The thin solid curve in Figure 28.14(a) is an example of an alternative minimum delay buffered path. Certainly, different buffer paths may have different congestion cost. Then the minimum congestion cost buffered path can be found by performing the Dijkstra’s algorithm on the tile graph which is demonstrated in Figure 28.14(b). In Figure 28.14(b), each solid edge corresponds to a tile boundary and its edge cost is the corresponding wiring congestion cost. There are two types of nodes, the empty circle nodes that have zero cost and filled circle nodes that have cost equal to the placement congestion cost in corresponding tile. In conclusion, the shortest path obtained in this way produces a buffered path with both good timing and low congestion cost.

One of the issue related to the use of analytical formula is that the upstream resistance $R_d$ is unknown in the bottom-up solution propagation process. However, the lower bound on the upstream resistance is $R = \min(R_d, R_b)$ and the upper bound $\overline{R}$ is $\max(R_d, R_b)$ plus the upstream wire resistance\(^2\). Then, we can sample a few values between $R$ and $\overline{R}$, and find the minimum cost buffered path for each value. Since the timing result is not sensitive to the upstream resistance, normally the sampling size is very limited.

\(^2\)The maximum upstream wire resistance can be derived from the length of maximum buffer-to-buffer interval. This is also mentioned in [14].
5.3 Layout Navigation

In order to estimate the congestion efficiently, the solution quality of plate based tree adjustment algorithms is restricted by the size of tile graph and the plate size. Since the complexity the algorithm in [13] (describe in Section 5.2.1) increases quadratically with plate size, using a fine tiling and a large plate size would be computationally prohibitive. More importantly, distinctions between critical and non-critical nets are missing in the algorithm. Practically, we may need to generate different solutions for critical and non-critical nets.

In order to speed up the tree adjustment process, at most one candidate per tile is allowed, which results in a maze routing based algorithm [17]. The right cost function is paramount so as to maintain the quality. Moreover, instead of performing plate to plate routing of a sequence of tile to tile routes, the entire optimization is performed in a single pass. This allows one to use as large a plate as necessary, for almost no runtime penalty. During the maze routing-like process, an immediate solution only contains the cost information of the subtree.

By parameterizing the cost function to trade-off critical and non-critical nets, which leads to the algorithm in [18]. We construct the cost function as follows, according to the criticality of the nets.

For Non-Critical Nets, some nets require buffering to fix electrical violations (such as slew, capacitance, or noise). Some other want the net to avoid highly dense areas or routing congestion. However, one still wants to minimize wirelength to some degree. So we set the cost to be $1 + e(g_i)$ and assume the total tile congestion cost$^3 e(g_i)$ is between 0 and 1, i.e., $0 \leq e(g_i) \leq 1$. This implies that a tile blocked for routing and/or density has cost twice that of a tile that uses no resources. The constant of one can be viewed as a “delay component”. A tile that corresponds to a Steiner point must merge the costs of the children into a single cost, by simply adding up the cost functions of all the children. Since these are non-critical nets, all sinks are treated equally by having initial cost zero.

---

$^3$The total congestion cost of a tile can be a linear expression of the squares of tile density and all its boundary densities.
For Critical Nets, the cost impact of the environment is immaterial. We seek the absolutely best possible slack. When a net is optimally buffered (assuming no obstacles), its delay is a linear function of its length [19]. Hence, in order to minimize delay, we simply minimize the number of tiles to the most critical sink, which results in a unit cost defined for each tile. When merging the branches, we pick the branch with worst slack, so the merged cost is the maximum of both costs. The costs at sinks are initialized based on the sink criticality. The more critical a sink, the higher its initial cost. Finally, the objective is to minimize cost at the source.

The algorithm is able to the trade-off between the critical and non-critical cost functions during the maze routing procedure. Let $0 \leq K \leq 1$ be the trade-off parameter, where $K = 1$ corresponds to a non-critical net and $K = 0$ corresponds to a critical net. Based on the previously defined cost functions for non-critical and critical nets, the cost function for a tile $g_i$ is then $1 + K \cdot e(g_i)$. For critical nets, merging branches is a maximization function, while it is an additive function for non-critical nets. These ideas can be combined while the merging cost of two children $g_i$ and $g_j$ becomes $\max(\text{cost}(g_i), \text{cost}(g_j)) + K \cdot \min(\text{cost}(g_i), \text{cost}(g_j))$.

It has been demonstrated that $K$ can be used to trade off the cost function, the merging operation, and even sink initialization. In practice, we can first optimize all nets that need buffering with $K = 1$, which limits the use of scarce resources. After performing a timing analysis, those nets that still have negative slack can be re-optimized with a smaller value of $K$, e.g., 0.7. This process of re-optimizing and gradually reducing can continue until, say, $K = 0.1$.

### 5.4 Relating Buffering Candidate Locations to Layout Environment

While the previous algorithms are considering the routing tree adjustment, the following algorithm focuses on buffer insertion candidate selection for congestion reduction.

Van Ginneken style algorithm assumes that a set of buffer insertion candidate locations are predetermined for the given topology. The most common method for selecting insertion points is to choose them at regular intervals. Alpert et al. [15] show how the quality of results is affected by the degree of wire segmenting that is performed on the topology. For example, Figure 28.15(a) shows
uniform segmenting for a Steiner tree with three sinks and a single blockage. For these regions for which buffer insertion is forbidden, one simply avoids inserting buffer candidate locations on top of the blockage. In (b), one can find the same uniform segmenting scheme, but with finer spacing. The additional buffer insertion locations could potentially improve the timing for the buffered net, for additional runtime cost. In (c), one can use roughly the same number of buffer insertion candidates as in uniform segmenting, but spacing them asymmetrically. The purpose is not to improve timing performance, but rather to bias van Ginneken style algorithm to insert buffers in regions of the design that are more favorable, such as areas with lower congestion cost.

In order to accomplish this buffer candidate selection, [18] applies a linear time and linear memory shortest path algorithm. The algorithm constructs a directed acyclic graph (DAG) over the set of potential candidate locations and chooses a subset by constructing a shortest path via a topological sort.

Let $L$ be the maximum allowable tiles in the tile graph (described in Section 4.1) between consecutive buffers which could be determined by a maximum allowable slew constraint. If buffers are placed at a distance greater than $L$ tiles away, then an electrical violation results or performance is significantly sacrificed. Based on $L$, edges are created by connecting the tiles which are no greater than $L$ tiles away from each other. The edge represents a pair of consecutive buffer candidates on the fixed routing tree.

Moreover, we define $S$ to be the desired number of tiles between consecutive buffer insertion
candidates, which is chosen by the user to obtain the desired timing performance/CPU trade-off. For example, Figure 28.15(a) has a value of that is twice that of Figure 28.15(b). For asymmetric spacing, a penalty is associated for spacing tiles either closer to or further from the desired spacing $S$. We define a function $pen(x, S, L) = \frac{(x-S)^2}{(L-S)^2}$ that assigns a penalty cost on an edge when the distance $x$ between tiles is not equal to $S$. Together with the congestion consideration, the total cost of a path is the summation over the penalty cost of all edges and the congestion cost of all tiles on the path. Hence, the problem can be solved by a topological sort, which finds the minimum cost path from the source to all sinks. By apply this pre-processing technique, buffers finally inserted significantly improve the overall design congestion with virtually no impact on either computation time or buffered net delays. In fact, since the pre-processing is more selective of the potential buffer insertion candidates, the final buffer insertion process can be speed up dramatically.
References


