Chapter 26 Buffer Insertion Basics

Jiang Hu, Zhuo Li and Shiyan Hu

1 Motivation

When the VLSI technology scales, gate delay and wire delay change in opposite directions. Smaller devices imply less gate switching delay. In contrast, thinner wire size leads to increased wire resistance and greater signal propagation delay along wires. As a result, wire delay has become a dominating factor for VLSI circuit performance. Further, it is becoming a limiting factor to the progress of VLSI technology. This is the well-known interconnect challenge [1–3]. Among many techniques addressing this challenge [4, 5], buffer (or repeater) insertion is such an effective one that it is an indispensable necessity for timing closure in submicron technology and beyond. Buffers can reduce wire delay by restoring signal strength, in particular, for long wires. Moreover, buffers can be applied to shield capacitive load from timing-critical paths such that the interconnect delay along critical paths are reduced.

As the ratio of wire delay to gate delay increases from one technology to the next, more and more buffers are required to achieve performance goals. The buffer scaling is studied by Intel

---

*Department of Electrical and Computer Engineering, Texas A&M University, College Station, TX 77843.  
IBM Austin Research Lab, Austin, TX 78758.  
 Department of Electrical and Computer Engineering, Texas A&M University, College Station, TX 77843.
and the results are reported in [6]. One metric that reveals the scaling is *critical buffer length* - the minimum distance beyond which inserting an optimally placed and sized buffer makes the interconnect delay less than that of the corresponding unbuffered wire. When wire delay increases due to the technology scaling, the critical buffer length becomes shorter, i.e., the distance that a buffer can comfortably drive shrinks. According to [6], the buffer critical length decreases by 68% when the VLSI technology migrates from 90nm to 45nm (for two generations). Please note that the critical buffer length scaling significantly outpaces the VLSI technology scaling which is roughly $0.5 \times$ for every two generations. If we look at the percentage of block level nets requiring buffers, it grows from 5.8% in 90nm technology to 19.6% in 45nm technology [6]. Perhaps the most alarming result is the scaling of buffer count [6] which predicts that 35% of cells will be buffers in 45nm technology as opposed to only 6% in 90nm technology.

The dramatic buffer scaling undoubtedly generates large and profound impact to VLSI circuit design. With millions of buffers required per chip, almost nobody can afford to neglect the importance of buffer insertion as compared to a decade ago when only a few thousands of buffers are needed for a chip [7]. Due to this importance, buffer insertion algorithms and methodologies need to be deeply studied on various aspects. First, a buffer insertion algorithm should deliver solutions of high quality since interconnect and circuit performance largely depend on the way that buffers are placed. Second, a buffer insertion algorithm needs to be sufficiently fast so that millions of nets can be optimized in reasonable time. Third, accurate delay models are necessary to ensure that buffer insertion solutions are reliable. Fourth, buffer insertion techniques are expected to simultaneously handle multiple objectives, such as timing, power and signal integrity, and their tradeoffs. Last but not the least, buffer insertion should interact with other layout steps, such as placement and routing, as the sheer number of buffers has already altered the landscape of circuit
layout design. Many of these issues will be discussed in subsequent sections and other chapters.

2 Optimization of Two-Pin Nets

For buffer insertion, perhaps the most simple case is a two-pin net, which is a wire segment with a driver (source) at one end and a sink at the other end. The simplicity allows closed form solutions to buffer insertion in two-pin nets.

If the delay of a two-pin net is to be minimized by using a single buffer type \( b \), one needs to decide the number of buffers \( k \) and the spacing between the buffers, the source and the sink. First, let us look at a very simple case in order to attain an intuitive understanding of the problem. In this case, the length of the two-pin net is \( l \) and the wire resistance and capacitance per unit length are \( r \) and \( c \), respectively. The number of buffers \( k \) has been given and is fixed. The driver resistance is the same as the buffer output resistance \( R_b \). The load capacitance of the sink is identical to buffer input capacitance \( C_b \). The buffer has an intrinsic delay of \( t_b \). The \( k \) buffers separates the net into \( k + 1 \) segments, with length of \( \vec{l} = (l_0, l_1, ..., l_k)^T \) (see Figure 1). Then, the Elmore delay of this net can be expressed as:

\[
t(\vec{l}) = \sum_{i=0}^{k} (\alpha l_i^2 + \beta l_i + \gamma)
\]

(1)

where \( \alpha = \frac{1}{2}rc \), \( \beta = R_b c + rC_b \) and \( \gamma = R_b C_b + t_b \). A formal problem formulation is

Minimize \[ t(\vec{l}) \]  

Subject to \[ g(\vec{l}) = l - \sum_{i=0}^{k} l_i = 0 \]

(3)

According to the Kuhn-Tucker condition [8], the following equation is the necessary condition for
the optimal solution.

\[ \vec{\nabla} t(\vec{l}) + \lambda \vec{\nabla} g(\vec{l}) = 0 \]  \hspace{1cm} (4)

where \( \lambda \) is the Lagrangian multiplier. According to the above condition, it can be easily derived that

\[ l_i = \frac{\beta}{\lambda - 2\alpha}, \quad i = 0, 1, \ldots, k \]  \hspace{1cm} (5)

Since \( \alpha, \beta \) and \( \lambda \) are all constants, it can be seen that the buffers need to be equally spaced in order to minimize the delay. This is an important conclusion that can be treated as a rule of thumb. The value of the Lagrangian multiplier \( \lambda \) can be found by plugging (5) into (3).

\[ k = \left\lfloor -\frac{1}{2} + \sqrt{1 + \frac{2(rcl + r(C_b - C_L) - c(R_b - R_d))^2}{r_c(R_bC_b + t_b)}} \right\rfloor \]  \hspace{1cm} (6)

In more general cases, the driver resistance \( R_d \) may be different from that of buffer output resistance and so is the sink capacitance \( C_L \). For such cases, the optimum number of buffers minimizing the delay is given by [9]:

\[ k = \left\lfloor -\frac{1}{2} + \sqrt{1 + \frac{2(rcl + r(C_b - C_L) - c(R_b - R_d))^2}{r_c(R_bC_b + t_b)}} \right\rfloor \]  \hspace{1cm} (6)
The length of each segment can be obtained through [9]:

\[
\begin{align*}
l_0 &= \frac{1}{k+1} \left( l + \frac{k(R_b - R_d)}{r} + \frac{C_L - C_b}{c} \right) \\
l_1 &= \ldots = l_{k-1} = \frac{1}{k+1} \left( l - \frac{R_b - R_d}{r} - \frac{k(C_L - C_b)}{c} \right) \\
l_k &= \frac{1}{k+1} \left( l - \frac{R_b - R_d}{r} - \frac{k(C_L - C_b)}{c} \right)
\end{align*}
\]  
(7)

A closed form solution to simultaneous buffer insertion/sizing and wire sizing is reported in [10]. Figure 2 shows an example of this simultaneous optimization. The wire is segmented into \( m \) pieces. The length \( l_i \) and width \( h_i \) of each wire piece \( i \) are the variables to be optimized. There are \( k \) buffers inserted between these pieces. The size \( b_i \) of each buffer \( i \) is also a decision variable. A buffer location is indicated by its surrounding wire pieces. For example, if the set of wire pieces between buffer \( i-1 \) and \( i \) is \( P_{i-1} \), the distance between the two buffers is equal to \( \sum_{j \in P_{i-1}} l_j \). There are two important conclusions [10] for the optimal solution that minimizes the delay. First, all wire pieces have the same length, i.e., \( l_i = \frac{l}{m}, i = 1, 2, ..., m \). Second, for wire pieces \( P_{i-1} = \{ p_{i-1,1}, p_{i-1,2}, ..., p_{i-1,m_{i-1}} \} \) between buffer \( i-1 \) and \( i \), their widths satisfy \( h_{i-1,1} > h_{i-1,2} > \ldots > h_{i-1,m_{i-1}} \) and form a geometric progression.

![Figure 2: An example of simultaneous buffer insertion/sizing and wire sizing.](image)

5
Van Ginneken’s algorithm

For a general case of signal nets, which may have multiple sinks, van Ginneken’s algorithm [11] is perhaps the first systematic approach on buffer insertion. For a fixed signal routing tree and given candidate buffer locations, van Ginneken’s algorithm can find the optimal buffering solution that maximizes timing slack according to the Elmore delay model. If there are \( n \) candidate buffer locations, its computation complexity is \( O(n^2) \). Based on van Ginneken’s algorithm, numerous extensions have been made, such as handling of multiple buffer types, tradeoff with power and cost, addressing slew rate and crosstalk noise, using accurate delay models and speedup techniques. These extensions will be covered in subsequent sections.

At a high level, van Ginneken’s algorithm [11] proceeds bottom-up from the leaf nodes toward the driver along a given routing tree. A set of candidate solutions keep updated during the process, where three operations adding wire, inserting buffers and branch merging may be performed. Meanwhile, the inferior solutions are pruned to accelerate the algorithm. After a set of candidate solutions are propagated to the source, the solution with the maximum required arrival time is selected as the final solution. For a routing tree with \( n \) buffer positions, the algorithm computes the optimal buffering solution in \( O(n^2) \) time.

A net is given as a binary routing tree \( T = (V, E) \), where \( V = \{s_0\} \cup V_s \cup V_n \), and \( E \subseteq V \times V \). Vertex \( s_0 \) is the source vertex and also the root of \( T \), \( V_s \) is the set of sink vertices, and \( V_n \) is the set of internal vertices. In the existing literatures, \( s_0 \) is also referred as driver. Denote by \( T(v) \) the subtree of \( T \) rooted at \( v \). Each sink vertex \( s \in V_s \) is associated with a sink capacitance \( C(s) \) and a required arrival time (RAT). Each edge \( e \in E \) is associated with lumped resistance \( R(e) \) and capacitance \( C(e) \). A buffer library \( B \) containing all the possible buffer types which can be
assigned to a buffer position is also given. In this section, $B$ contains only one buffer type. Delay estimation is obtained using the Elmore delay model, which is described in Chapter 3. A buffer assignment $\gamma$ is a mapping $\gamma : V_n \rightarrow B \cup \{\emptyset\}$ where $\emptyset$ denotes that no buffer is inserted. The timing buffering problem is defined as follows.

**Timing Driven Buffer Insertion Problem**: Given a binary routing tree $T = (V, E)$, possible buffer positions, and a buffer library $B$, compute a buffer assignment $\gamma$ such that the RAT at driver is maximized.

### 3.1 Concept of Candidate Solution

A buffer assignment $\gamma$ is also called a *candidate solution* for the timing buffering problem. A *partial solution*, denoted by $\gamma_v$, refers to an incomplete solution where the buffer assignment in $T(v)$ has been determined.

The Elmore delay from $v$ to any sink $s$ in $T(v)$ under $\gamma_v$ is computed by

$$D(s, \gamma_v) = \sum_{e=(v_i, v_j)} (D(v_i) + D(e)),$$

where the sum is taken over all edges along the path from $v$ to $s$. The slack of vertex $v$ under $\gamma_v$ is defined as

$$Q(\gamma_v) = \min_{s \in T(v)} \{RAT(s) - D(s, \gamma_v)\}.$$

At any vertex $v$, the effect of a partial solution $\gamma_v$ to its upstream part is characterized by a $(Q(\gamma_v), C(\gamma_v))$ pair, where $Q$ is the slack at $v$ under $\gamma_v$ and $C$ is the downstream capacitance viewing at $v$ under $\gamma_v$. 
3.2 Generating Candidate Solutions

Van Ginneken’s algorithm proceeds bottom-up from the leaf nodes toward the driver along $T$. A set of candidate solutions, denoted by $\Gamma$, keep updated during this process. There are three operations through solution propagation, namely, wire insertion, buffer insertion and branch merging (see Figure 3). We are to describe them in turn.

3.2.1 Wire insertion

Suppose that a partial solution $\gamma_v$ at position $v$ propagates to an upstream position $u$ and there is no branching point in between. If no buffer is placed at $u$, then only wire delay needs to be considered. Therefore, the new solution $\gamma_u$ can be computed as

$$Q(\gamma_u) = Q(\gamma_v) - D(e),$$
$$C(\gamma_u) = C(\gamma_v) + C(e),$$  \hspace{1cm} (8)$$

where $e = (u, v)$ and $D(e) = R(e)(\frac{C(e)}{2} + C(\gamma_v))$. 

![Figure 3: Operations in van Ginneken’s algorithm.](image-url)
3.2.2 Buffer insertion

Suppose that we add a buffer $b$ at $u$. $\gamma_u$ is then updated to $\gamma'_u$ where

\[
Q(\gamma'_u) = Q(\gamma_u) - (R(b) \cdot C(\gamma_u) + K(b)), \\
C(\gamma'_u) = C(b).
\]

(9)

3.2.3 Branch merging

When two branches $T_l$ and $T_r$ meet at a branching point $v$, $\Gamma_l$ and $\Gamma_r$, which correspond to $T_l$ and $T_r$, respectively, are to be merged. The merging process is performed as follows. For each solution $\gamma_l \in \Gamma_l$ and each solution $\gamma_r \in \Gamma_r$, generate a new solution $\gamma'$ according to:

\[
C(\gamma') = C(\gamma_l) + C(\gamma_r), \\
Q(\gamma') = \min\{Q(\gamma_l), Q(\gamma_r)\}.
\]

(10)

The smaller $Q$ is picked since the worst-case circuit performance needs to be considered.

3.3 Inferiority and Pruning Identification

Simply propagating all solutions by the above three operations makes the solution set grow exponentially in the number of buffer positions processed. An effective and efficient pruning technique is necessary to reduce the size of the solution set. This motivates an important concept - inferior solution - in van Ginneken’s algorithm. For any two partial solutions $\gamma_1, \gamma_2$ at the same vertex $v$, $\gamma_2$ is inferior to $\gamma_1$ if $C(\gamma_1) \leq C(\gamma_2)$ and $Q(\gamma_1) \geq Q(\gamma_2)$. Whenever a solution becomes inferior, it is pruned from the solution set. Therefore, only solutions excel in at least one aspect of downstream capacitance and slack can survive.
For an efficient pruning implementation and thus an efficient buffering algorithm, a sorted list is used to maintain the solution set. The solution set $\Gamma$ is increasingly sorted according to $C$, and thus $Q$ is also increasingly sorted if $\Gamma$ does not contain any inferior solutions.

By a straightforward implementation, when adding a wire, the number of candidate solutions will not change; when inserting a buffer, only one new candidate solution will be introduced. More efforts are needed to merge two branches $T_l$ and $T_r$ at $v$. For each partial solution in $\Gamma_l$, find the first solution with larger $Q$ value in $\Gamma_r$. If such a solution does not exist, the last solution in $\Gamma_r$ will be taken. Since $\Gamma_l$ and $\Gamma_r$ are sorted, we only need to traverse them once. Partial solutions in $\Gamma_r$ are similarly treated. It is easy to see that after merging, the number of solutions is at most $|\Gamma_l| + |\Gamma_r|$. As such, given $n$ buffer positions, at most $n$ solutions can be generated at any time. Consequently, the pruning procedure at any vertex in $T$ runs in $O(n)$ time.

3.4 Pseudo-code

In van Ginneken’s algorithm, a set of candidate solutions are propagated from sinks to driver. Along a branch, after a candidate buffer location $v$ is processed, all solutions are propagated to its upstream buffer location $u$ through wire insertion. A buffer is then inserted to each solution to obtain a new solution. Meanwhile, inferior solutions are pruned. At a branching point, solution sets from all branches are merged by merging process. In this way, the algorithm proceeds in the bottom-up fashion and the solution with maximum required arrival time at driver is returned. Given $n$ buffer positions in $T$, van Ginneken’s algorithm can compute a buffer assignment with maximum slack at driver in $O(n^2)$ time since any operation at any node can be performed in $O(n)$ time. Refer to Figure 4 for the pseudo-code of van Ginneken’s algorithm.
Algorithm: van Ginneken’s algorithm.

**Input:** $T$: routing tree, $B$: buffer library

**Output:** $\gamma$ which maximizes slack at driver

1. for each sink $s$, build a solution set $\{\gamma_s\}$, where $Q(\gamma_s) = \text{RAT}(s)$ and $C(\gamma_s) = C(s)$
2. for each branching point/driver $v_t$ in the order given by a postorder traversal of $T$, let $T'$ be each of the branches $T_1$, $T_2$ of $v_t$ and $\Gamma'$ be the solution set corresponding to $T'$, do
3. for each wire $e$ in $T'$, in a bottom-up order, do
4. for each $\gamma \in \Gamma'$, do
5. $C(\gamma) = C(\gamma) + C(e)$
6. $Q(\gamma) = Q(\gamma) - D(e)$
7. prune inferior solutions in $\Gamma'$
8. if the current position allows buffer insertion, then
9. for each $\gamma \in \Gamma'$, generate a new solution $\gamma'$
10. set $C(\gamma') = C(b)$
11. set $Q(\gamma') = Q(\gamma) - R(b) \cdot C(e) - K(b)$
12. $\Gamma' = \Gamma' \cup \{\gamma'\}$ and prune inferior solutions
13. // merge $\Gamma_1$ and $\Gamma_2$ to $\Gamma_v$
14. set $\Gamma_v = \emptyset$
15. for each $\gamma_1 \in \Gamma_1$ and $\gamma_2 \in \Gamma_2$, generate a new solution $\gamma'$
16. set $C(\gamma') = C(\gamma_1) + C(\gamma_2)$
17. set $Q(\gamma') = \min\{Q(\gamma_1), Q(\gamma_2)\}$
18. $\Gamma_v = \Gamma_v \cup \{\gamma'\}$ and prune inferior solutions
19. return $\gamma$ with the largest slack

Figure 4: Van Ginneken’s algorithm.

### 3.5 Example

Let us look at a simple example to illustrate the work flow of van Ginneken’s algorithm. Refer to Figure 5. Assume that there are three non-dominated solutions at $v_3$ whose $(Q, C)$ pairs are

$$(200, 10), (300, 30), (500, 50),$$

and there are two non-dominated solutions at $v_2$ whose $(Q, C)$ pairs are

$$(290, 5), (350, 20).$$
Figure 5: An example for performing van Ginneken’s algorithm.

We first propagate them to $v_1$ through wire insertion. Assume that $R(v_1, v_3) = 3$ and $C(v_1, v_3) = 2$. Solution (200,10) at $v_3$ becomes $(200 - 3 \cdot (2/2 + 10), 10 + 2) = (167, 12)$ at $v_1$. Similarly, the other two solutions become (207, 32), (347, 52). Assume that $R(v_2, v_3) = 2$ and $C(v_2, v_3) = 2$, solutions at $v_2$ become (278, 7), (308, 22) at $v_1$.

We are now to merge these solutions at $v_1$. Denote by $\Gamma_l$ the solutions propagated from $v_3$ and by $\Gamma_r$ the solutions propagated from $v_2$. Before merging, partial solutions in $\Gamma_l$ are

$$(167, 12), (207, 32), (347, 52),$$

and partial solutions in $\Gamma_r$ are

$$(278, 7), (308, 22).$$

After branch merging, the new candidate partial solutions whose $Q$ are dictated by solutions in $\Gamma_l$ are

$$(167, 19), (207, 39), (308, 74),$$
and those dictated by solutions in $\Gamma_r$ are

$$(278, 59), (308, 74).$$

After pruning inferior solutions, the solution set at $v_1$ is

$$\{(167, 19), (207, 39), (278, 59), (308, 74)\}.$$ 

4 Van Ginneken Extensions

4.1 Handling Library with Multiple Buffers

We extend the standard van Ginneken’s algorithm to handle multiple buffers and buffer cost [12]. The buffer library $B$ now contains various types of buffers. Each buffer $b$ in the buffer library has a cost $W(b)$, which can be measured by area or any other metric, depending on the optimization objective. A function $f : V_n \rightarrow 2^B$ specifies the types of buffers allowed at each internal vertex in $T$. The cost of a solution $\gamma$, denoted by $W(\gamma)$, is defined as $W(\gamma) = \sum_{b \in \gamma} W_b$. With the above notations, our new problem can be formulated as follows.

**Minimum Cost Timing Constrained Buffer Insertion Problem:** Given a binary routing tree $T = (V, E)$, possible buffer positions defined using $f$, and a buffer library $B$, to compute a minimal-cost buffer assignment $\gamma$ such that the RAT at driver is smaller than a timing constraint $\alpha$.

In contrast to the single buffer type case, $W$ is introduced into the $(Q, C)$ pair to handle buffer cost, i.e., each solution is now associated with a $(Q, C, W)$-triple. As such, during the process of bottom-up computation, additional efforts need to be made in updating $W$: if $\gamma'$ is generated by
inserting a wire into $\gamma$, then $W(\gamma') = W(\gamma)$; if $\gamma'$ is generated by inserting a buffer $b$ into $\gamma$, then $W(\gamma') = W(\gamma) + W(b)$; if $\gamma'$ is generated by merging $\gamma_l$ with $\gamma_r$, then $W(\gamma') = W(\gamma_l) + W(\gamma_r)$.

The definition of inferior solutions needs to be revised as well. For any two solutions $\gamma_1, \gamma_2$ at the same node, $\gamma_1$ dominates $\gamma_2$ if $C(\gamma_1) \leq C(\gamma_2)$, $W(\gamma_1) \leq W(\gamma_2)$ and $Q(\gamma_1) \geq Q(\gamma_2)$. Whenever a solution becomes dominated, it is pruned from the solution set. Therefore, only solutions excel in at least one aspect of downstream capacitance, buffer cost and RAT can survive.

With the above modification, van Ginneken’s algorithm can easily adapt to the new problem setup. However, since the domination is defined on a $(Q, C, W)$ triple rather than a $(Q, C)$ pair, more efficient pruning technique is necessary to maintain the efficiency of the algorithm. As such, range search tree technique is incorporated [12]. This technique will be described in details in Section 5.2.

### 4.2 Library with Inverters

So far, all buffers in the buffer library are non-inverting buffers. There can also have inverting buffers, or simply inverters. In terms of buffer cost and delay, inverter would provide cheaper buffer assignment and better delay over non-inverting buffers. As regard to algorithmic design, it is worth noting that introducing inverters into the buffer library brings the polarity issue to the problem, as the output polarity of a buffer will be negated after inserting an inverter.

### 4.3 Polarity Constraints

When output polarity for driver is required to be positive or negative, we impose a polarity constraint to the buffering problem. To handle polarity constraints, during the bottom-up computation,
the algorithm maintains two solution sets, one for positive and one for negative buffer input polarity. After choosing the best solution at driver, the buffer assignment can be then determined by a top-down traversal. The details of the new algorithm are elaborated as follows.

Denote the two solution sets at vertex $v$ by $\Gamma_v^+$ and $\Gamma_v^-$ corresponding to positive polarity and negative polarity, respectively. Supposed that an inverter $b^-$ is inserted to a solution $\gamma_v^+ \in \Gamma_v^+$, a new solution $\gamma'_v$ is generated in the same way as before except that it will be placed into $\Gamma_v^-$. Similarly, the new solution generated by inserting $b^-$ to a solution $\gamma_v^- \in \Gamma_v^-$ will be placed into $\Gamma_v^+$. For inserting a non-inverting buffer, the new solution is placed in the same set as its origin.

The other operations are easier to handle. The wire insertion goes the same as before and two solution sets are handled separately. Merging is carried out only among the solutions with the same polarity, e.g., the positive-polarity solution set of left branch is merged with that of the right branch. For inferiority check and solution pruning, only the solutions in the same set can be compared.

### 4.4 Slew and Capacitance Constraints

The **slew rate** of a signal refers to the rising or falling time of a signal switching. Sometimes the slew rate is referred as signal transition time. The slew rate of almost every signal has to be sufficiently small since a large slew rate implies large delay, large short circuit power dissipation and large vulnerability to crosstalk noise. In practice, a maximal slew rate constraint is required at the input of each gate/buffer. Therefore, this constraint needs to be obeyed in a buffering algorithm [12–15].

A simple slew model is essentially equivalent to the Elmore model for delay. It can be explained using a generic example which is a path $p$ from node $v_i$ (upstream) to $v_j$ (downstream) in a buffered
tree. There is a buffer (or the driver) $b_u$ at $v_i$, and there is no buffer between $v_i$ and $v_j$. The slew rate $S(v_j)$ at $v_j$ depends on both the output slew $S_{b_u,\text{out}}(v_i)$ at buffer $b_u$ and the slew degradation $S_w(p)$ along path $p$ (or wire slew), and is given by [16]:

$$S(v_j) = \sqrt{S_{b_u,\text{out}}(v_i)^2 + S_w(p)^2}. \quad (11)$$

The slew degradation $S_w(p)$ can be computed with Bakoglu’s metric [17] as

$$S_w(p) = \ln 9 \cdot D(p), \quad (12)$$

where $D(p)$ is the Elmore delay from $v_i$ to $v_j$.

The output slew of a buffer, such as $b_u$ at $v_i$, depends on the input slew at this buffer and the load capacitance seen from the output of the buffer. Usually, the dependence is described as a 2-D lookup table. As a simplified alternative, one can assume a fixed input slew at each gate/buffer. This fixed slew is equal to the maximum slew constraint and therefore is always satisfied but is a conservative estimation. For fixed input slew, the output slew of buffer $b$ at vertex $v$ is then given by

$$S_{b,\text{out}}(v) = R_b \cdot C(v) + K_b, \quad (13)$$

where $C(v)$ is the downstream capacitance at $v$, $R_b$ and $K_b$ are empirical fitting parameters. This is similar to empirically derived K-factor equations [18]. We call $R_b$ the slew resistance and $K_b$ the intrinsic slew of buffer $b$.

In a van Ginneken style buffering algorithm, if a candidate solution has a slew rate greater than
given slew constraint, it is pruned out and will not be propagated any more. Similar as the slew constraint, circuit designs also limit the maximum capacitive load a gate/buffer can drive [15]. For timing non-critical nets, buffer insertion is still necessary for the sake of satisfying the slew and capacitance constraints. For this case, fast slew buffering techniques are introduced in [19].

4.5 Integration with Wire Sizing

In addition to buffer insertion, wire sizing is an effective technique for improving interconnect performance [20–24]. If wire size can take only discrete options, which is often the case in practice, wire sizing can be directly integrated with van Ginneken style buffer insertion algorithm [12]. In the bottom-up dynamic programming procedure, multiple wire width options need to be considered when a wire is added (see Section 3.2.1). If there are \( k \) options of wire size, then \( k \) new candidate solutions are generated, one corresponding each wire size. However, including the wire sizing in van Ginneken’s algorithm makes the complexity pseudo-polynomial [12].

In [25], layer assignment and wire spacing are considered in conjunction with wire sizing. A combination of layer, width and spacing is called a wire code. All wires in a net have to use an identical wire code. If each wire code is treated as a polarity, the wire code assignment can be integrated with buffer insertion in the same way as handling polarity constraint (see Section 4.3). In contrast to simultaneous wire sizing and buffer insertion [12], the algorithm complexity stays polynomial after integrating wire code assignment [25] with van Ginneken’s algorithm.

Another important conclusion in [25] is about wire tapering. Wire tapering means that a wire segment is divided into multiple pieces and each piece can be sized individually. In contrast, uniform wire sizing does not make such division and maintain the same wire width for the entire
Tapered Wire Sizing

Uniform Wire Sizing

Figure 6: Wire sizing with tapering and uniform wire sizing.

segment. These two cases are illustrated in Figure 6. It is shown in [25] that the benefit of wire tapering versus uniform wire sizing is very limited when combined with buffer insertion. It is theoretically proved [25] that the signal velocity from simultaneous buffering with wire tapering is at most 1.0354 times of that from buffering and uniform wire sizing. In short, wire tapering improves signal speed by at most 3.54% over uniform wire sizing.

4.6 Noise Constraints with Devgan Metric

The shrinking of minimum distance between adjacent wires has caused an increase in the coupling capacitance of a net to its neighbors. A large coupling capacitance can cause a switching net to induce significant noise onto a neighboring net, resulting in an incorrect functional response. Therefore, noise avoidance techniques must become an integral part of the performance optimization environment.

The amount of coupling capacitance from one net to another is proportional to the distance that the two nets run parallel to each other. The coupling capacitance may cause an input signal on the aggressor net to induce a noise pulse on the victim net. If the resulting noise is greater than
the tolerable noise margin (NM) of the sink, then an electrical fault results. Inserting buffers in the victim net can separate the capacitive coupling into several independent and smaller portions, resulting in smaller noise pulse on the sink and the input of the inserted buffers.

Before describing the noise-aware buffering algorithms, we first introduce the coupling noise metric in Section 4.6.1.

### 4.6.1 Devgan’s coupling noise metric

Among many coupling noise models, Devgan’s metric [26] is particularly amenable for noise avoidance in buffer insertion, because its computational complexity, structure, and incremental nature is the same as the famous Elmore delay metric. Further, like the Elmore delay model, the noise metric is a provable upper bound on coupled noise. Other advantages of the noise metric include the ability to incorporate multiple aggressor nets and handle general victim and aggressor net topologies. A disadvantage of the Devgan metric is that it becomes more pessimistic as the ratio of the aggressor net’s transition time (at the output of the driver) to its delay decreases. However, cases in which this ratio becomes very small are rare since a long net delay generally corresponds to a large load on the driver, which in turn causes a slower transition time. The metric does not consider the duration of the noise pulse either. In general, the noise margin of a gate is dependent on both the peak noise amplitude and the noise pulse width. However, when considering failure at a gate, peak amplitude dominates pulse width.

If a wire segment $e$ in the victim net is adjacent with $t$ aggressor nets, let $\lambda_1, ..., \lambda_t$ be the ratios of coupling to wire capacitance from each aggressor net to $e$, and let $\mu_1, ..., \mu_t$ be the slopes of the aggressor signals. The impact of a coupling from aggressor $j$ can be treated as a current source $I_{e,j} = C_e \cdot \lambda_j \cdot \mu_j$ where $C_e$ is the wire capacitance of wire segment $e$. This is illustrated in Figure 7.
The total current induced by the aggressors on \( e \) is

\[
I_e = C_e \sum_{j=1}^{t} (\lambda_j \cdot \mu_j) \tag{14}
\]

Often, information about neighboring aggressor nets is unavailable, especially if buffer insertion is performed before routing. In this case, a designer may wish to perform buffer insertion to improve performance while also avoiding future potential noise problems. When performing buffer insertion in estimation mode, one might assume that: (1) there is a single aggressor net which couples with each wire in the routing tree, (2) the slope of all aggressors is \( \mu \), and (3) some fixed ratio \( \lambda \) of the total capacitance of each wire is due to coupling capacitance.

Let \( I_{T(v)} \) be defined as the total downstream current seen at node \( v \), i.e.,

\[
I_{T(v)} = \sum_{e \in E_{T(v)}} I_e,
\]

where \( E_{T(v)} \) is the set of wire edges downstream of node \( v \). Each wire adds to the noise induced on the victim net. The amount of additional noise induced from a wire \( e = (u, v) \) is given by

\[
\text{Noise}(e) = R_e \left( \frac{I_e}{2} + I_{T(v)} \right) \tag{15}
\]

20
where $R_e$ is the wire resistance. The total noise seen at sink $si$ starting at some upstream node $v$ is

$$\text{Noise}(v - si) = R_e I_{T(v)} + \sum_{e \in \text{path}(v - si)} \text{Noise}(e)$$

(16)

where $R_e = 0$ if there is no gate at node $v$. The path from $v$ to $si$ has no intermediate buffers.

Each node $v$ has a predetermined noise margin $NM(v)$. If the circuit is to have no electrical faults, the total noise propagated from each driver/buffer to each its sink $si$ must be less than the noise margin for $si$. We define the noise slack for every node $v$ as

$$NS(v) = \min_{si \in SI_{T(v)}} NM(si) - \text{Noise}(v - si)$$

(17)

where $SI_{T(v)}$ is the set of sink nodes for the subtree rooted at node $v$. Observe that $NS(si) = NM(si)$ for each sink $si$.

### 4.6.2 Algorithm of buffer insertion with noise avoidance

We begin with the simplest case of a single wire with uniform width and neighboring coupling capacitance. Let us consider a wire $e = (u, v)$. First, we need to ensure $NS(v) \geq R_b I_{T(v)}$ where $R_b$ is the buffer output resistance. If this condition is not satisfied, inserting a buffer even at node $v$ cannot satisfy the constraint of noise margin, i.e., buffer insertion is needed within subtree $T(v)$.

If $NS(v) \geq R_b I_{T(v)}$, we next search for the maximum wirelength $l_{e,max}$ of $e$ such that inserting a buffer at $u$ always satisfies noise constraints. The value of $l_{e,max}$ tells us the maximum unbuffered length or the minimum buffer usage for satisfying noise constraints. Let $R = R_e/l_e$ be the wire resistance per unit length and $I = I_e/l_e$ be the current per unit length. According to [27], this value
can be determined by

\[ l_{e,\text{max}} = \frac{R_b}{R} \cdot \frac{I_{T(v)}}{I} + \sqrt{\left(\frac{R_b}{R}\right)^2 + \left(\frac{I_{T(v)}}{I}\right)^2 + \frac{2NS(v)}{I \cdot R}} \]  

(18)

Depending on the timing criticality of the net, the noise-aware buffer insertion problem can be formulated in two different ways: (A) minimize total buffer cost subject to noise constraints; (B) maximize timing slack subject to noise constraints.

The algorithm for (A) is a bottom-up dynamic programming procedure which inserts buffers greedily as far apart as possible [27]. Each partial solution at node \( v \) is characterized by a 3-tuple of downstream noise current \( I_{T(v)} \), noise slack \( NS(v) \) and buffer assignment \( M \). In the solution propagation, the noise current is accumulated in the same way as the downstream capacitance in van Ginneken’s algorithm. Likewise, noise slack is treated like the timing slack (or required arrival time). This algorithm can return an optimal solution for a multi-sink tree \( T = (V, E) \) in \( O(|V|^2) \) time.

The core algorithm of noise constrained timing slack maximization is similar as van Ginneken’s algorithm except that the noise constraint is considered. Each candidate solution at node \( v \) is represented by a 5-tuple of downstream capacitance \( C_v \), required arrival time \( q(v) \), downstream noise current \( I_{T(v)} \), noise slack \( NS(v) \) and buffer assignment \( M \). In addition to pruning inferior solutions according to the \((C, q)\) pair, the algorithm eliminates candidate solutions that violate the noise constraint. At the source, the buffering solution not only has optimized timing performance but also satisfies the noise constraint.
4.7 Higher Order Delay Modeling

Many buffer insertion methods [11, 12, 28] are based on the Elmore wire delay model [29] and a linear gate delay model for the sake of simplicity. However, the Elmore delay model often overestimates interconnect delay. It is observed in [30] that Elmore delay sometimes has over 100% overestimation error when compared to SPICE. A critical reason of the overestimation is due to the neglect of the resistive shielding effect. In the example of Figure 8, the Elmore delay from node $A$ to $B$ is equal to $R_1(C_1 + C_2)$ assuming that $R_1$ can see the entire capacitance of $C_2$ despite the fact that $C_2$ is somewhat shielded by $R_2$. Consider an extreme scenario where $R_2 = \infty$ or there is open circuit between node $B$ and $C$. Obviously, the delay from $A$ to $B$ should be $R_1C_1$ instead of the Elmore delay $R_1(C_1 + C_2)$. The linear gate delay model is inaccurate due to its neglect of nonlinear behavior of gate delay in addition to resistive shielding effect. In other words, a gate delay is not a strictly linear function of load capacitance.

![Figure 8: Example of resistive shielding effect.](image)

The simple and relatively inaccurate delay models are suitable only for early design stages such as buffer planning. In post-placement stages, more accurate models are needed because (1) optimal buffering solutions based on simple models may be inferior since actual delay is not being optimized; (2) simplified delay modeling can cause a poor evaluation of the trade-off between total buffer cost and timing improvement. In more accurate delay models, the resistive shielding effect is considered by replacing lumped load capacitance with higher order load admittance estimation.
The accuracy of wire delay can be improved by including higher order moments of transfer function. An accurate and popular gate delay model is usually a lookup table employed together with effective capacitance [31, 32] which is obtained based on the higher order load admittance. These techniques will be described in more details as follows.

4.7.1 Higher order point admittance model

For an RC tree, which is a typical circuit topology in buffer insertion, the frequency domain point admittance at a node \( v \) is denoted as \( Y_v(s) \). It can be approximated by the third order Taylor expansion

\[
Y_v(s) = y_{v,0} + y_{v,1}s + y_{v,2}s^2 + y_{v,3}s^3 + O(s^4)
\]

where \( y_{v,0}, y_{v,1}, y_{v,2} \) and \( y_{v,3} \) are expansion coefficients. The third order approximation usually provides satisfactory accuracy in practice. Its computation is a bottom-up procedure starting from the leaf nodes of an RC tree, or the ground capacitors. For a capacitance \( C \) connected to ground, the admittance at its upstream end is simply \( Cs \). Please note that the zeroth order coefficient is equal to 0 in an RC tree since there is no DC path connected to ground. Therefore, we only need to propagate \( y_1, y_2 \) and \( y_3 \) in the bottom-up computation. There are two cases we need to consider:

- Case 1: For a resistance \( R \), given the admittance \( Y_{d}(s) \) of its downstream node, compute the admittance \( Y_{u}(s) \) of its upstream node (Figure 9(a)).

\[
\begin{align*}
y_{u,1} &= y_{d,1} \\
y_{u,2} &= y_{d,2} - Ry_{d,1}^2 \\
y_{u,3} &= y_{d,3} - 2Ry_{d,1}y_{d,2} + R^2y_{d,1}^3
\end{align*}
\] (19)

- Case 2: Given admittance \( Y_{d1}(s) \) and \( Y_{d2}(s) \) corresponding to two branches, compute the
admittance $Y_u(s)$ after merging them (Figure 9(b)).

$$y_{u,1} = y_{d1,1} + y_{d2,1} \quad y_{u,2} = y_{d1,2} + y_{d2,2} \quad y_{u,3} = y_{d1,3} + y_{d2,3}$$  \hspace{1cm} (20)$$

(a) \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} (b)

Figure 9: Two scenarios of admittance propagation.

The third order approximation $(y_1, y_2, y_3)$ of an admittance can be realized as an RC $\pi$-model $(C_u, R_\pi, C_d)$ (Figure 10) where

$$C_u = y_1 - \frac{y_2^2}{y_3} \quad R_\pi = -\frac{y_2^2}{y_3^2} \quad C_d = \frac{y_2^2}{y_3}$$  \hspace{1cm} (21)$$

Figure 10: Illustration of $\pi$-model.

4.7.2 Higher order wire delay model

While the Elmore delay is equal to the first order moment of transfer function, the accuracy of delay estimation can be remarkably improved by including higher order moments. For example,
the wire delay model [33] based on the first three moments and the closed-form model [34] using the first two moments.

Since van Ginneken style buffering algorithms proceed in a bottom-up manner, bottom-up moment computations are required. Figure 11(a) shows a wire \( e \) connected to a subtree rooted at node \( B \). Assume that the first \( k \) moments \( m_{BC}^{(1)}, m_{BC}^{(2)}, ..., m_{BC}^{(k)} \) have already been computed for the path from \( B \) to \( C \). We wish to compute the moments \( m_{AC}^{(1)}, m_{AC}^{(2)}, ..., m_{AC}^{(k)} \) so that the \( A \leadsto C \) delay can be derived.

![Figure 11: Illustration of bottom-up moment computation.](image)

The techniques in Section 4.7.1 are used to reduce the subtree at \( B \) to a \( \pi - \) model \((C_j, R_\pi, C_f)\) (Figure 11(b)). Node \( D \) just denotes the point on the far side of the resistor connected to \( B \) and is not an actual physical location. The RC tree can be further simplified to the network shown in Figure 11(c). The capacitance \( C_j \) and \( C_e/2 \) at \( B \) are merged to form a capacitor with value \( C_n \).

The moments from \( A \) to \( B \) can be recursively computed by the equation

\[
m_{AB}^{(i)} = -R_e(m_{AB}^{(i-1)} + m_{AD}^{(i-1)} C_f)
\]  

(22)

where the moments from \( A \) to \( D \) are given by

\[
m_{AD}^{(i)} = m_{AB}^{(i)} - m_{AD}^{(i-1)} R_\pi C_f
\]  

(23)
and \( m_{AB}^{(0)} = m_{AD}^{(0)} = 1 \). Now the moments from \( A \) to \( C \) can be computed via moment multiplication as follows.

\[
m_{AC}^{(i)} = \sum_{j=0}^{i} (m_{AB}^{(j)} \cdot m_{BC}^{(i-j)})
\]  

(24)

One property of Elmore delay that makes it attractive for timing optimization is that the delays are additive. This property does not hold for higher order delay models. Consequently, a non-critical sink in a subtree may become a critical sink depending the value of upstream resistance [35]. Therefore, one must store the moments for all the paths to downstream sinks during the bottom-up candidate solution propagation.

4.7.3 Accurate gate delay

A popular gate delay model with decent accuracy consists of the following three steps:

1. Compute a \( \pi \)-model of the driving point admittance for the RC interconnect using the techniques introduced in Section 4.7.1.
2. Given the \( \pi \)-model and the characteristics of the driver, compute an effective capacitance \( C_{eff} \) [31, 32].
3. Based on \( C_{eff} \), compute the gate delay using \( k \)-factor equations or lookup table [36].

4.8 Flip-flop Insertion

The technology scaling leads to decreasing clock period, increasing wire delay and growing chip size. Consequently, it often takes multiple clock cycles for signals to reach their destinations along global wires. Traditional interconnect optimization techniques such as buffer insertion are inade-
quate in handling this scenario and flip-flop/latch insertion (or interconnect pipelining) becomes a necessity.

In pipelined interconnect design, flip-flops and buffers are inserted simultaneously in a given Steiner tree $T = (V, E)$ [37, 38]. The simultaneous insertion algorithm is similar to van Ginneken’s dynamic programming method except that a new criterion - latency, needs to be considered. The latency from the signal source to a sink is the number of flip-flops in-between. Therefore, a candidate solution at node $v \in V$ is characterized by its latency $\lambda_v$ in addition to downstream capacitance $C_v$, required arrival time (RAT) $q_v$. Obviously, a small latency is preferred.

The inclusion of flip-flop and latency also requests other changes in a van Ginneken style algorithm. When a flip-flop is inserted in the bottom-up candidate propagation, the RAT at the input of this flip-flop is reset to clock period time $T_\phi$. The latency of corresponding candidate solution is also increased by 1. For the ease of presentation, clock skew and setup/hold time are neglected without loss of generality. Then, the delay between two adjacent flip-flops cannot be greater than the clock period time $T_\phi$, i.e., the RAT cannot be negative. During the candidate solution propagation, if a candidate solution has negative RAT, it should be pruned without further propagation. When merge two candidate solutions from two child branches, the latency of the merged solution is the maximum of the two branch solutions.

There are two formulations for the simultaneous flip-flop and buffer insertion problem. MiLa: find the minimum latency that can be obtained. GiLa: find a flip-flop/buffer insertion implementation that satisfies given latency constraint. MiLa can be used for the estimation of interconnect latency at the micro architectural level. After the micro-architecture design is completed, all interconnect must be designed so as to abide to given latency requirements by using GiLa.

The algorithm of MiLa [38] and GiLa [38] are shown in Figure 12 and Figure 13, respectively.
In GiLa, the $\lambda_u$ for a leaf node $u$ is the latency constraint at that node. Usually, $\lambda_u$ at a leaf is a non-positive number. For example, $\lambda_u = -3$ requires that the latency from the source to node $u$ is 3. During the bottom-up solution propagation, $\lambda$ is increased by 1 if a flip-flop is inserted. Therefore, $\lambda = 0$ at the source implies that the latency constraint is satisfied. If the latency at the source is greater than zero, then the corresponding solution is not feasible (line 2.6.1 of Figure 13). If the latency at the source is less than zero, the latency constraint can be satisfied by padding extra flip-flops in the corresponding solution (line 2.6.2.1 of Figure 13). The padding procedure is called $ReFlop(T_u, k)$ which inserts $k$ flip-flops in the root path of $T_u$. The root path is from $u$ to either a leaf node or a branch node $v$ and there is no other branch node in-between. The flip-flops previously inserted on the root path and the newly inserted $k$ flip-flops are redistributed evenly along the path. When merge solutions from two branches in GiLa, $ReFlop$ is performed (line 3.3-3.4.1 of Figure 13) for the solutions with smaller latency to ensure that there is at least one merged solution matching the latency of both branches.

5 Speedup Techniques

Due to dramatically increasing number of buffers inserted in the circuits, algorithms that can efficiently insert buffers are essential for the design automation tools. In this chapter, several recent proposed speedup results are introduced and the key techniques are described.

5.1 Recent Speedup Results

This chapter studies buffer insertion in interconnect with a set of possible buffer positions and a discrete buffer library. In 1990, van Ginneken [11] proposed an $O(n^2)$ time dynamic programming
**Algorithm: MiLa(T_u)/MiLa(T_{u,v})**

**Input:** Subtree rooted at node u or edge (u, v)

**Output:** A set of candidate solutions Γ_u

**Global:** Routing tree T and buffer library B

1. if u is a leaf, Γ_u = (C_u, q_u, 0, 0) // q is required arrival time
2. else if u has one child node v or the input is T_{u,v}
   2.1 Γ_v = MiLa(v)
   2.2 Γ_u = ∪γ∈Γ_v(addWire((u, v), γ))
   2.3 Γ_b = ∅
   2.4 for each b in B
      2.4.1 Γ = ∪γ∈Γ_u(addBuffer(γ, b))
      2.4.2 prune Γ
      2.4.3 Γ_b = Γ_b ∪ Γ
   2.5 Γ_u = Γ_u ∪ Γ_b
3. else if u has two child edges (u, v) and (u, z)
   3.1 Γ_{u,v} = MiLa(T_{u,v}), Γ_{u,z} = MiLa(T_{u,z})
   3.2 Γ_u = Γ_u ∪ merge(Γ_{u,v}, Γ_{u,z})
4. prune Γ_u
5. return Γ_u

Figure 12: The MiLa algorithm.

algorithm for buffer insertion with one buffer type, where n is the number of possible buffer positions. His algorithm finds a buffer insertion solution that maximizes the slack at the source. In 1996, Lillis, Cheng and Lin [12] extended van Ginneken’s algorithm to allow b buffer types in time O(b^2 n^2).

Recently, many efforts are taken to speedup the van Ginneken’s algorithm and its extension. Shi and Li [39] improved the time complexity of van Ginneken’s algorithm to O(b^2 n log n) for 2-pin nets, and O(b^2 n log^2 n) for multi-pin nets. The speedup is achieved by four novel techniques: predictive pruning, candidate tree, fast redundancy check, and fast merging. To reduce the quadratic effect of b, Li and Shi [40] proposed an algorithm with time complexity O(b n^2). The speedup is achieved by the observation that the best candidate to be associated with any buffer must lie on the convex hull of the (Q, C) plane and convex pruning. To utilize the fact that in real applications most nets have small numbers of pins and large number of buffer positions, Li and
Algorithm: \textit{GiLa}(T_u)/\textit{GiLa}(T_{u,v})

\textbf{Input}: Subtree \(T_u\) rooted at node \(u\) or edge \((u,v)\)

\textbf{Output}: A set of candidate solutions \(\Gamma_u\)

\textbf{Global}: Routing tree \(T\) and buffer library \(B\)

1. \textbf{if} \(u\) is a leaf, \(\Gamma_u = (C_u, q_u, \lambda_u, 0)\)
2. \textbf{else if} node \(u\) has one child node \(v\) or the input is \(T_{u,v}\)
   \begin{enumerate}
   \item \(\Gamma_v = \text{GiLa}(T_v)\)
   \item \(\Gamma_u = \bigcup_{\gamma \in \Gamma_v} (\text{addWire}((u,v), \gamma))\)
   \item \(\Gamma_b = \emptyset\)
   \item \textbf{for} each \(b\) \textbf{in} \(B\)
     \begin{enumerate}
     \item \(\Gamma = \bigcup_{\gamma \in \Gamma_u} (\text{addBuffer}((\gamma, b))\)
     \item \text{prune} \(\Gamma\)
     \item \(\Gamma_b = \Gamma_b \cup \Gamma\)
     \end{enumerate}
   \end{enumerate}
\textbf{Global}: \(\Gamma_u \equiv \{\Gamma^x, ..., \Gamma^y\}\), \(x, y\) indicate latency
\textbf{else if} \(u\) is source
\textbf{Global}: \(\Gamma_u = \text{Reflop}(T_u, -y)\)
3. \textbf{else if} \(u\) has two child edges \((u,v)\) and \((u,z)\)
   \begin{enumerate}
   \item \(\Gamma_{u,v} = \text{GiLa}(T_{u,v}), \Gamma_{u,z} = \text{GiLa}(T_{u,z})\)
   \item \(\| \Gamma_{u,v} = \{\Gamma^x, ..., \Gamma^y\}, \Gamma_{u,z} = \{\Gamma^m, ..., \Gamma^n\}\)
   \item \textbf{if} \(y < m\) \textbf{else} \(\text{insert} -y \text{ more flops in } \Gamma_{u,v}\)
   \begin{enumerate}
   \item \(\Gamma_{u,v} = \text{Reflop}(T_{u,v}, m - y)\)
   \item \textbf{if} \(n < x\) \textbf{else} \(\text{insert} x - n \text{ more flops in } \Gamma_{u,z}\)
   \begin{enumerate}
   \item \(\Gamma_{u,z} = \text{Reflop}(T_{u,z}, x - n)\)
   \end{enumerate}
   \end{enumerate}
   \end{enumerate}
\textbf{else if} \(u\) has two child edges \((u,v)\) and \((u,z)\)
\textbf{else if} \(u\) is source
\textbf{Global}: \(\Gamma_u = \text{Reflop}(T_u, -y)\)
4. \text{prune} \(\Gamma_u\)
5. \text{return} \(\Gamma_u\)

Figure 13: The GiLa algorithm.

Shi [41] proposed a simple \(O(mn)\) algorithms for \(m\)-pin nets. The speedup is achieved by the property explored in [40], convex pruning, a clever bookkeeping method and an innovative linked list that allow \(O(1)\) time update for adding a wire or a candidate.

In the following subsections, new pruning techniques, efficient way to find the best candidates when adding a buffer, and implicit data representations are presented. They are the basic component of many recent speedup algorithms.
5.2 Predictive Pruning

During the van Ginneken’s algorithm, a candidate is pruned out only if there is another candidate that is superior in terms of capacitance and slack. This pruning is based on the information at the current node being processed. However, all candidates at this node must be propagated further upstream toward the source. This means the load seen at this node must be driven by some minimal amount of upstream wire or gate resistance. By anticipating the upstream resistance ahead of time, one can prune out more potentially inferior candidates earlier rather than later, which reduces the total number of candidates generated. More specifically, assume that each candidate must be driven by an upstream resistance of at least $R_{\min}$. The pruning based on anticipated upstream resistance is called predictive pruning.

**Definition 1 (Predictive pruning)** Let $\alpha_1$ and $\alpha_2$ be two nonredundant candidates of $T(v)$ such that $C(\alpha_1) < C(\alpha_2)$ and $Q(\alpha_1) < Q(\alpha_2)$. If $Q(\alpha_2) - R_{\min} \cdot C(\alpha_2) \leq Q(\alpha_1) - R_{\min} \cdot C(\alpha_1)$, then $\alpha_2$ is pruned.

Predictive pruning preserves optimality. The general situation is shown in Fig. 14. Let $\alpha_1$ and $\alpha_2$ be candidates of $T(v_1)$ that satisfy the condition in Definition 1. Using $\alpha_1$ instead of $\alpha_2$ will not increase delay from $v$ to sinks in $v_2, \ldots, v_k$. It is easy to see $C(v, \alpha_1) < C(v, \alpha_2)$. If $Q$ at $v$ is determined by $T(v_1)$, we have

$$
Q(v, \alpha_1) - Q(v, \alpha_2) = Q(v_1, \alpha_1) - Q(v_1, \alpha_2) - R_{\min} \cdot (C(v_1, \alpha_1) - C(v_1, \alpha_2)) \\
\geq 0
$$

Therefore, $\alpha_2$ is redundant.
Predictive pruning technique prunes more redundant solutions while guarantees optimality. It is one of four key techniques of fast algorithms proposed in [39]. In [42], significant speedup is achieved by simply extending predictive pruning technique to buffer cost. Aggressive predictive pruning technique, which uses a resistance larger than $R_{\text{min}}$ to prune candidates, is proposed in [43] to achieve further speedup with a little degradation of solution quality.

5.3 Convex Pruning

The basic data structure of van Ginneken’s algorithms is a sorted list of non-dominated candidates. Both the pruning in van Ginneken’s algorithm and the predictive pruning are performed by comparing two neighboring candidates a time. However, more potentially inferior candidates can be pruned out by comparing three neighboring candidate solutions simultaneously. For three solutions in the sorted list, the middle one may be pruned according to convex pruning.

**Definition 2 (Convex pruning)** Let $\alpha_1$, $\alpha_2$ and $\alpha_3$ be three nonredundant candidates of $T(v)$ such that $C(\alpha_1) < C(\alpha_2) < C(\alpha_3)$ and $Q(\alpha_1) < Q(\alpha_2) < Q(\alpha_3)$. If

$$\frac{Q(\alpha_2) - Q(\alpha_1)}{C(\alpha_2) - C(\alpha_1)} < \frac{Q(\alpha_3) - Q(\alpha_2)}{C(\alpha_3) - C(\alpha_2)},$$

then we call $\alpha_2$ non-convex, and prune it.
Convex pruning can be explained by Figure 15. Consider $Q$ as the $Y$-axis and $C$ as the $X$-axis. Then candidates are points in the two-dimensional plane. It is easy to see that the set of non-redundant candidates $N(v)$ is a monotonically increasing sequence. Candidate $\alpha_2 = (Q_2, C_2)$ in the above definition is shown in Figure 15(a), and is pruned in Figure 15(b). The set of nonredundant candidates after convex pruning $M(v)$ is a convex hull.

![Figure 15](image)

Figure 15: (a) Nonredundant candidates $N(v)$. (b) Nonredundant candidates $M(v)$ after convex pruning.

For 2-pin nets, convex pruning preserves optimality. Let $\alpha_1, \alpha_2$ and $\alpha_3$ be candidates of $T(v)$ that satisfy the condition in Definition 2. In Figure 15, let the slope between $\alpha_1$ and $\alpha_2$ ($\alpha_2$ and $\alpha_3$) be $\rho_{1,2}$ ($\rho_{2,3}$). If candidate $\alpha_2$ is not on the convex hull of the solution set, then $\rho_{1,2} < \rho_{2,3}$. These candidates must have certain upstream resistance $R$ including wire resistance and buffer/driver resistance. If $R < \rho_{2,3}$, $\alpha_2$ must become inferior to $\alpha_3$ when both candidates are propagated to the upstream node. Otherwise, $R > \rho_{2,3}$ which implies $R > \rho_{1,2}$, and therefore $\alpha_2$ must become inferior to $\alpha_1$. In other words, if a candidate is not on the convex hull, it will be pruned either by the solution ahead of it or the solution behind it. Please note that this conclusion only applies to 2-pin nets. For multi-pin nets when the upstream could be a merging vertex, nonredundant candidates that are pruned by convex pruning could still be useful.

Convex pruning of a list of non-redundant candidates sorted in increasing $(Q, C)$ order can be performed in linear time by Graham’s scan. Furthermore, when a new candidate is inserted to the
list, we only need to check its neighbors to decide if any candidate should be pruned under convex pruning. The time is $O(1)$, amortized over all candidates.

In [40, 41], the convex pruning is used to form the convex hull of non-redundant candidates, which is the key component of the $O(n^2)$ algorithm and $O(mn)$ algorithm. In [43], convex pruning (called squeeze pruning) is performed on both 2-pin and multi-pin nets to prune more solutions with a little degradation of solution quality.

### 5.4 Efficient Way to Find Best Candidates

Assume $v$ is a buffer position, and we have computed the set of nonredundant candidates $N'(v)$ for $T(v)$, where $N'(v)$ does not include candidates with buffers inserted at $v$. Now we want to insert buffers at $v$ and compute $N(v)$. Define $P_i(v, \alpha)$ as the slack at $v$ if we add a buffer of type $B_i$ for any candidate $\alpha$:

$$P_i(v, \alpha) = Q(v, \alpha) - R(B_i) \cdot C(v, \alpha) - K(B_i).$$  \hfill (26)

If we do not insert any buffer, then every candidate in $N'(v)$ is a candidate in $N(v)$. If we insert a buffer, then for every buffer type $B_i$, $i = 1, 2, \ldots, b$, there will be a new candidate $\beta_i$:

$$Q(v, \beta_i) = \max_{\alpha \in N'(v)} \{P_i(v, \alpha)\},$$

$$C(v, \beta_i) = C(B_i).$$

Define the best candidate for $B_i$ as the candidate $\alpha \in N'(v)$ such that $\alpha$ maximizes $P_i(v, \alpha)$ among all candidates in $N'(v)$. If there are multiple $\alpha$’s that maximize $P_i(v, \alpha)$, choose the one
with minimum $C$. In van Ginneken’s algorithm, it takes $O(bn)$ to find one best candidate at each buffer position.

According to convex pruning, it is easy to see that all best candidates are on the convex hull. The following lemma says that if we sort candidates in increasing $Q$ and $C$ order from left to right, then as we add wires to the candidates, we always move to the left to find the best candidates.

**Lemma 1** For any $T(v)$, let nonredundant candidates after convex pruning be $\alpha_1, \alpha_2, \ldots, \alpha_k$, in increasing $Q$ and $C$ order. Now add wire $e$ to each candidate $\alpha_j$ and denote it as $\alpha_j + e$. For any buffer type $B_i$, if $\alpha_j$ gives the maximum $P_i(\alpha_j)$ and $\alpha_k$ gives the maximum $P_i(\alpha_k + e)$, then $k \leq j$.

The following lemma says the best candidate can be found by local search, if all candidates are convex.

**Lemma 2** For any $T(v)$, let nonredundant candidates after convex pruning be $\alpha_1, \alpha_2, \ldots, \alpha_k$, in increasing $Q$ and $C$ order. If $P_i(\alpha_{j-1}) \leq P_i(\alpha_j)$, $P_i(\alpha_j) \geq P_i(\alpha_{j+1})$, then $\alpha_j$ is the best candidate for buffer type $B_i$ and

$$P_i(\alpha_1) \leq \cdots \leq P_i(\alpha_{j-1}) \leq P_i(\alpha_j),$$

$$P_i(\alpha_j) \geq P_i(\alpha_{j+1}) \geq \cdots \geq P_i(\alpha_k).$$

With the above two lemmas and convex pruning, best candidates are founded in amortized $O(n)$ time in [40] and $O(b)$ time in [41], which are more efficient than van Ginneken’s algorithm.

---

1In [40], Lemma 1 is presented differently. It says if all buffers are sorted decreasingly according to driving resistance, then the best candidates for each buffer type in such order is from left to right.
5.5 Implicit Representation

Van Ginnken’s algorithm uses explicit representation to store slack and capacitance values, and therefore it takes $O(bn)$ time when adding a wire. It is possible to use implicit representation to avoid explicit updating of candidates.

In the implicit representation, $C(v, \alpha)$ and $Q(v, \alpha)$ are not explicitly stored for each candidate. Instead, each candidate contains 5 fields: $q$, $c$, $qa$, $ca$ and $ra$. When $q$, $ca$ and $ra$ are all 0, $q$ and $c$ give $Q(v, \alpha)$ and $C(v, \alpha)$, respectively. When a wire is added, only $qa$, $ca$ and $ra$ in the root of the tree ([39]) or as global variables themselves ([41]) are updated. Intuitively, $qa$ represents extra wire delay, $ca$ represents extra wire capacitance and $ra$ represents extra wire resistance.

It takes only $O(1)$ time to add a wire with the implicit representation [39, 41]. For example, in [41], when we reach an edge $e$ with resistance $R(e)$ and $C(e)$, $qa$, $ra$ and $ca$ are updated to reflect new values of $Q$ and $C$ of all previous candidates in $O(1)$ time, without actually touching any candidate:

\[
qa = qa + R(e) \cdot C(e)/2 + R(e) \cdot ca,
\]
\[
ca = ca + C(e),
\]
\[
ra = ra + R(e).
\]

\footnote{In [41], only 2 fields, $q$ and $c$, are necessary for each candidate. $qa$, $ca$ and $ra$ are global variables for each 2-pin segment.}
The actual value of $Q$ and $C$ of each candidate $\alpha$, are decided as follows

$$Q(\alpha) = q - qa - ra \cdot c,$$

$$C(\alpha) = c + ca. \quad (27)$$

Implicit representation is applied on balance tree in [39], where the operation of adding a wire takes $O(b \log n)$ time. It is applied on a sorted linked list in [41], where the operation of adding a wire takes $O(1)$ time.

References


