A Fast Algorithm for Optimal Buffer Insertion

Weiping Shi, *Senior Member, IEEE,* and Zhuo Li, *Student Member, IEEE*

***Abstract*—The classic buffer insertion algorithm of van Ginneken has time and space complexity $O(n^2)$, where n is the number of possible buffer positions. For more than a decade, van Ginneken's algorithm has been the foundation of buffer insertion. In this paper, we present a new algorithm that computes the same optimal buffer insertion, but runs much faster. For 2-pin nets, our time complexity is $O(n \log n)$ and space complexity is $O(n)$. For multipin nets, our time complexity is $O(n \log^2 n)$ and space complexity is $O(n \log n)$. The speedup is achieved by four novel techniques: predictive pruning, candidate tree, fast redundancy check, and fast merging. On industrial test cases, the new algorithms is 2–80 times faster than van Ginneken's algorithm and uses $1/4$–$1/500$ of the memory. Since van Ginneken's algorithm and its variations are used by most existing algorithms on buffer insertion and buffer sizing, our new algorithm significantly improves the performance of all these algorithms. The predictive pruning technique has been applied to buffer cost minimization (Shi *et al.*, 2004), and significantly improved the running time.**

***Index Terms*—Buffer insertion, data structure, Elmore delay, interconnect, routing.**

I. Introduction

As the feature size continues to shrink, delay optimization of interconnect becomes increasingly important. One popular technique for reducing interconnect delay is buffer insertion [1], [2], [7]–[9], [14], [15]. The objective of the optimal buffer insertion problem is to find where to insert buffers in the interconnect so that the timing requirements are met.

For buffer insertion under a given routing tree, van Ginneken [14] in 1990 proposed a dynamic programming algorithm. His algorithm gives the optimal slack and has time and space complexity $O(n^2)$, where n is the number of possible buffer positions. Lillis *et al.* [8] extended van Ginneken's algorithm to allow $|B|$ buffer types in time $O(|B|^2 n^2)$.

Some researchers consider simultaneous routing tree construction and buffer insertion, which is an NP-hard problem. Okamoto and Cong [9] combined A-tree construction with van Ginneken's algorithm. Kang *et al.* [7] constructed a bounded delay tree, and then used van Ginneken's algorithm to optimize buffers. Zhou *et al.* [15] combined the shortest path algorithm with van Ginneken's algorithm.

For buffer insertion on a single line allowing continuous buffer positions and continuous buffer sizes, Dhar and Franklin [6] proposed a closed form solution, and Chu and Wong [4] proposed a quadratic programming approach. It should be pointed out that the discrete version of the buffer insertion problem, which is studied by van Ginneken and us, is more difficult than the continuous version of the problem. In addition, the continuous methods can not be applied to trees.

The performance of the above algorithms are limited by the quadratic time complexity of van Ginneken's algorithm, as pointed out by the researchers [1], [8], [15]. For large nets or large number of wire segments, van Ginneken's algorithm becomes the bottleneck.

Van Ginneken's algorithm consists of three major operations: 1) adding a buffer in $O(n)$ time; 2) adding a wire in $O(n)$ time; and 3) merging two branches in $O(n_1 + n_2)$ time, where n_1 and n_2 are the numbers of buffer positions in the two branches. As a result, van Ginneken's algorithm has time complexity $O(n^2)$. Our algorithm performs the three operations in much less time: 1) adding a buffer in $O(\log n)$ time; 2) adding a wire in $O(1)$ time; and 3) merging two branches in $O(n_2 \log n_1)$ time, where $n_1 \geq n_2$.

As a result, our algorithm has time complexity $O(n \log n)$ for 2-pin nets and $O(n \log^2 n)$ for multipin nets. Our speedup is achieved by four novel ideas: 1) predictive pruning; 2) candidate tree; 3) fast redundancy check; and 4) fast merging. Together, the four ideas make the improvement possible.

We also extend the basic algorithm to buffer sizing by allowing $|B|$ buffer types in time $O(|B|^2 n \log n)$ for 2-pin nets and $O(|B|^2 n \log^2 n)$ for multipin nets, an improvement of $O(|B|^2 n^2)$ time algorithms of Lillis *et al.* [8]. Furthermore, our algorithm allows the user to specify for each vertex the set of possible buffer sizes, which immediately implies an $O(n \log n)$ time solution for the driver selection problem studied by Alpert *et al.* [2]. For algorithms that use van Ginneken's algorithm as an inner subroutine, the new algorithm may significantly improve the running time of these algorithms.

It is interesting to note that van Ginneken's algorithm is similar to the $O(n^2)$ time floorplan minimization algorithms of Stockmeyer [13] and Otten [10]. Using a similar data structure and a fast merging scheme, Shi improved the time complexity of floorplanning to $O(n \log n)$ [11]. However, the buffer insertion problem is much more difficult due to the delay calculation.

Section II formulates the problem. Section III describes the techniques to speedup the three operations in van Ginneken's algorithm. The complete algorithm is in Section IV, the time and space complexity analysis is in Section V, and extension to multiple buffer types is in Section VI. Simulation results are given in Section VII and the conclusion is given in Section VIII.

II. Preliminary

A net is given as a routing tree $\mathbf{T} = (V, E)$, where $V = \{s_0\} \cup V_s \cup V_n$, and $E \subseteq V \times V$. Vertex s_0 is the *source* vertex

Manuscript received February 5, 2004; revised June 18, 2004. This work was supported in part by the National Science Foundation under Grant CCR-0098329, Grant CCR-0113668, Grant EIA-0223785, and ATP Grant 512-0266-2001, and in part by a fellowship from Applied Materials. This paper was recommended by Associate Editor J. Lillis.

The authors are with Department of Electrical Engineering, Texas A&M University, College Station, TX 77843 USA (e-mail: wshi@ee.tamu.edu; zhuoli@ee.tamu.edu).

Digital Object Identifier 10.1109/TCAD.2005.847942

and also the root of $\mathbf{T}$, V_s is the set of *sink* vertices, and V_n is the set of *internal* vertices. Each sink vertex $s \in V_s$ is associated with sink capacitance $C(s)$ and required arrival time $\mathrm{RAT}(s)$. A buffer library B contains different types of buffers. For each buffer type $b \in B$, the intrinsic delay is $K(b)$, driving resistance is $R(b)$, and input capacitance is $C(b)$. A function $f : V_n \to 2^B$ specifies the types of buffers allowed at each internal vertex. Each edge $e \in E$ is associated with lumped resistance $R(e)$ and capacitance $C(e)$.

Following previous researchers [1]–[2], [4], [7], [8], [9], [14], [15], we use the Elmore delay for the interconnect and the linear delay for buffers. For each edge $e = (v_i, v_j)$, signals travel from v_i to v_j. The Elmore delay of e is

$$D(e) = R(e)\left(\frac{C(e)}{2} + C(v_j)\right)$$

where $C(v_j)$ is the downstream capacitance at v_j. For any buffer b at vertex v_j, the buffer delay is

$$D(v_j) = R(b) \cdot C(v_j) + K(b)$$

where $C(v_j)$ is the downstream capacitance at v_j. When a buffer b is inserted, the capacitance viewed from the upper stream is $C(b)$.

For any vertex $v \in V$, let $T(v)$ be the subtree downstream from v, and with v being the root. Once we decide where to insert buffers in $T(v)$, we have a *candidate* α for $T(v)$. The delay from v to sink $s \in T(v)$ under α is

$$D(v, s, \alpha) = \sum_{e=(v_i, v_j)} \big(D(v_i) + D(e)\big)$$

where the sum is over all edges e in the path from v to s. If v_i is a buffer in α, then $D(v_i)$ is the buffer delay. If v_i is not a buffer in α, then $D(v_i) = 0$. The slack of v under α is

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

Buffer Insertion Problem: Given a routing tree $\mathbf{T} = (V, E)$, sink capacitance $C(s)$ and $\mathrm{RAT}(s)$ for each sink s, capacitance $C(e)$ and resistance $R(e)$ for each edge e, possible buffer position f, and buffer library B, find a candidate α for $\mathbf{T}$ that maximizes $Q(s_0, \alpha)$.

An example of the buffer insertion problem is shown in Fig. 1 and one of its candidate solutions is shown in Fig. 2.

The effect of a candidate to the upstream is traditionally described by slack Q and downstream capacitance C [14]. Define $C(v, \alpha)$ as the downstream capacitance at node v under candidate α. For any two candidates α_1 and α_2 of $T(v)$, we say α_1 *dominates* α_2, if $Q(v, \alpha_1) \geq Q(v, \alpha_2)$ and $C(v, \alpha_1) \leq C(v, \alpha_2)$. The set of *nonredundant candidates* of $T(v)$, which we denote as $N(v)$, is the set of candidates such that no candidate in $N(v)$ dominates any other candidate in $N(v)$, and every candidate of $T(v)$ is dominated by some candidates in $N(v)$. Once we have $N(s_0)$, the candidate that gives the maximum $Q(s_0, \alpha)$ can be found easily.

Finally, we briefly review the three major operations in van Ginneken's dynamic programming algorithm.

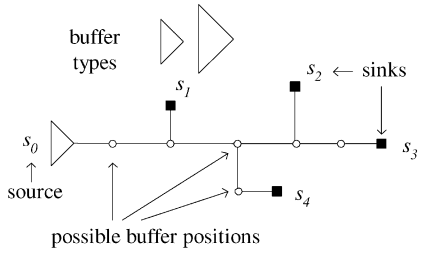


Fig. 1. Example of buffer insertion problem.

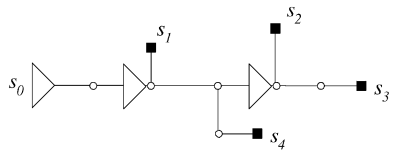


Fig. 2. One candidate solution for Fig. 1.

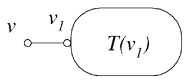


Fig. 3. $T(v)$ consists of buffer position v and $T(v_1)$.

Assume we have computed nonredundant candidates for $T(v_1)$, and now reach a buffer position v, see Fig. 3. Wire (v, v_1) has 0 resistance and capacitance. If we do not insert a buffer at v, then every candidate for $T(v_1)$ is a candidate for $T(v)$. If we insert a buffer at v, then there will be a new candidate β

$$Q(v, \beta) = \max_{\alpha}\{Q(v_1, \alpha) - R(b) \cdot C(v_1, \alpha) - K(b)\}$$
$$C(v, \beta) = C(b)$$

where $\max$ is taken over all nonredundant candidates α of $T(v_1)$. The new candidate β may make other candidates redundant, or may be redundant itself. Using a linked list to store nonredundant candidates, van Ginneken's algorithm takes $O(n)$ time to generate β, insert β into the list of nonredundant candidates, and delete redundancy.

Example 1: Assume there are three nonredundant candidates α_1, α_2, and α_3 for $T(v_1)$ with their (Q, C) values being $(200, 8), (300, 20)$, and $(400, 70)$, respectively. Further, assume a buffer with $R(b) = 8, K(b) = 5$ and $C(b) = 3$. The Q values of the three candidates after inserting the buffer will be as follows:

$$\alpha_1 \text{ with buffer: } 200 - 8 \cdot 8 - 5 = 131$$
$$\alpha_2 \text{ with buffer: } 300 - 8 \cdot 20 - 5 = 135$$
$$\alpha_3 \text{ with buffer: } 400 - 8 \cdot 70 - 5 = -165.$$

Therefore, the best candidate to insert a buffer is α_2, and the (Q, C) value of the new candidate β is $(135, 3)$. By inserting β into the original list of nonredundant candidates, we have

$$(135, 3), (200, 8), (300, 20), (400, 70).$$

In this case, all candidates are nonredundant.

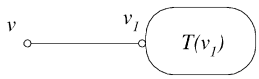


Fig. 4. $T(v)$ consists of wire (v, v_1) and $T(v_1)$.

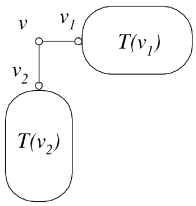


Fig. 5. $T(v)$ consists of $T(v_1)$ and $T(v_2)$.

When a wire $e = (v, v_1)$ is added as shown in Fig. 4, every candidate α for $T(v_1)$ becomes a candidate for $T(v)$ where

$$Q(v,\alpha) = Q(v_1,\alpha) - R(e)C(e)/2 - R(e)C(v_1,\alpha)$$
$$C(v,\alpha) = C(v_1,\alpha) + C(e).$$

Using a linked list, it takes $O(n)$ time to update candidates and check redundancy.

Example 2: Assume the (Q, C) values of nonredundant candidates for $T(v_1)$ are

$$(135, 3), (200, 8), (300, 20), (400, 70).$$

If we add a wire with $R(e) = 2$ and $C(e) = 2$, then these candidates become

$$(127, 5), (182, 10), (258, 22), (258, 72).$$

Clearly, the last candidate is redundant and should be deleted.

Finally, when two subtrees are merged as shown in Fig. 5, things are more complicated. Both edges (v, v_1) and (v, v_2) have zero resistance and capacitance. For each candidate α_1 in $T(v_1)$, we find the best candidate α_2 in $T(v_2)$ to form a new candidate β for $T(v)$

$$Q(v,\beta) = \min\{Q(v_1,\alpha_1), Q(v_2,\alpha_2)\}$$
$$C(v,\beta) = C(v_1,\alpha_1) + C(v_2,\alpha_2).$$

Do the same for each candidate in $T(v_2)$. Then, take the union of all candidates and delete redundancy. Using linked list, the process takes $O(n_1+n_2)$ time, where n_1 and n_2 are the number of nonredundant candidates for $T(v_1)$ and $T(v_2)$.

Example 3: Let the (Q, C) values of the nonredundant candidates for $T(v_1)$ be

$$(135, 3), (200, 8), (300, 20), (400, 70)$$

and the candidates for $T(v_2)$ be

$$(350, 10), (400, 20).$$

Then, the above process will produce the following candidates of $T(v)$ whose Q is determined by candidates in $T(v_2)$:

$$(350, 80), (400, 90)$$

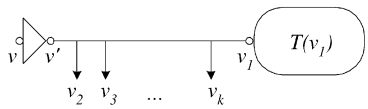


Fig. 6. If α_1 b-dominates α_2 at v_1, β_1 dominates β_2 at v.

and the following candidates of $T(v)$ whose Q is determined by candidates in $T(v_1)$:

$$(135, 13), (200, 18), (300, 30), (400, 90).$$

After deleting redundancy, the set of nonredundant candidates for $T(v)$ is

$$(135, 13), (200, 18), (300, 30), (350, 80), (400, 90).$$

III. Speedup Techniques

To illustrate the main ideas, we assume for now that there is only one noninverting buffer type b, and s_0 is also driven by a buffer of type b. Extensions to multiple buffer types are in Section VI.

A. Predictive Pruning

When we insert buffer b at v, we want to associate the buffer with a candidate α that maximizes slack

$$P(v,\alpha) = Q(v,\alpha) - R(b)\cdot C(v,\alpha) - K(b)$$

among all candidates. However, such a candidate is not necessarily the candidate with the maximum Q as shown in Example 1.

For any candidates α_1 and α_2 of $T(v)$, we say α_1 b-dominates α_2 if $P(v,\alpha_1) \geq P(v,\alpha_2)$ and $C(v,\alpha_1) \leq C(v,\alpha_2)$.

Lemma 1: If α_1 b-dominates α_2, then α_2 is redundant.

Proof: The general situation is shown in Fig. 6, where α_1 and α_2 are candidates for $T(v_1)$, β_1 and β_2 are candidates for $T(v)$ and v is the first buffer upstream from v_1 in βs. The only difference between β_1 and β_2 is that β_1 contains α_1 for $T(v_1)$, while β_2 contains α_2 for $T(v_1)$. It is sufficient to show if α_1 b-dominates α_2, then $Q(v,\beta_1) \geq Q(v,\beta_2)$

Using α_1 instead of α_2 will not increase delay from v to sinks in $v_2, \ldots, v_k$. If Q at v is determined by $T(v_1)$, let $R(v', v_1)$ be the resistance of the wire(s) from v' to v_1.

$$\begin{aligned} &Q(v,\beta_1) - Q(v,\beta_2) \\ &\quad = Q(v_1,\alpha_1) - \big(R(v',v_1) + R(b)\big)\cdot C(v_1,\alpha_1) \\ &\qquad -\Big(Q(v_1,\alpha_2) - \big(R(v',v_1) + R(b)\big)\cdot C(v_1,\alpha_2)\Big) \\ &\quad = P(v_1,\alpha_1) - P(v_1,\alpha_2) \\ &\qquad + R(v',v_1)\big(C(v_1,\alpha_2) - C(v_1,\alpha_1)\big) \\ &\quad \geq 0. \end{aligned}$$

■

It is easy to see if α_1 dominates α_2, then α_1 b-dominates α_2. From now on, we say a candidate is redundant if it is b-dominated by another candidate. We call this *predictive pruning* since it prunes the future redundant solutions. The nonredundant candidates after predictive pruning are in the same order as the traditional nonredundant candidates under (Q, C) pruning, except

that some candidates that are nonredundant under (Q, C) are redundant under (P, C). It is easy to show the following.

Lemma 2: If α_1 and α_2 do not b-dominate one another, then $P(v, \alpha_1) > P(v, \alpha_2)$ if and only if $Q(v, \alpha_1) > Q(v, \alpha_2)$.

Predictive pruning not only gives a better pruning criteria, but also allows us to find the candidate that gives the maximum P in $O(1)$ time. For the candidates in Example 1, we have

$$P(v_1, \alpha_1)\text{: } 200 - 8 \cdot 8 - 5 = 131$$
$$P(v_1, \alpha_2)\text{: } 300 - 8 \cdot 20 - 5 = 135$$
$$P(v_1, \alpha_3)\text{: } 400 - 8 \cdot 70 - 5 = -165.$$

Since the (P, C) values of the three candidates are

$$(131, 8), (135, 20), (-165, 70)$$

the last candidate α_3 is redundant under predictive pruning and should be deleted. So the remaining nonredundant candidates are α_1 and α_2, with their (P, C) values

$$(131, 8), (135, 20).$$

Therefore, the best candidate to insert the buffer is α_2.

Note that we assumed the source is driven by a buffer type of b. However, Lemma 1 and 2 are true for any source with driving resistance greater than $R(b)$. In general, if the upstream resistance of every node is at least R, then we can define a corresponding P and use it to prune.

B. Candidate Tree

We assume the readers are familiar with balanced binary search trees, such as red-black trees [5]. Given a balanced binary search tree of k keys, the search, insertion and deletion of any key can be done in $O(\log k)$ time. In practice, simple binary search trees that do not rebalance work almost as well as balanced search trees.

We will use a balanced binary search tree $A(v)$, which we call a *candidate tree*, to efficiently store nonredundant candidates of $T(v)$. Please do not confuse routing tree $T(v)$ with the candidate tree $A(v)$. The former is a topology while the latter is a data structure. For each candidate α of $T(v)$, there is a corresponding node $u(\alpha)$ in $A(v)$. $A(v)$ is organized in increasing C order and increasing Q order, and pruned by (P, C). This is possible because the candidates in $A(v)$ are nonredundant. For each routing tree, we have a candidate tree to store the nonredundant candidates for that routing tree. Since our dynamic programming algorithm is bottom up, initially there will be many candidate trees, one for each sink. As the sinks and branches are merged, the candidate trees are merged as explained later in the paper. Finally when we merge all the branches, there is only one candidate tree.

When an edge $e = (v, v_1)$ is inserted, see Fig. 4, the values of Q and C of each candidate α_i for $T(v_1)$ must be updated. Van Ginneken spends linear time to update each candidate, which is necessary for him since he stores Q and C explicitly.

The candidate tree is an implicit representation that allows $O(\log n)$ time insertion of wires and buffers. In the candidate tree, $C(v, \alpha)$ and $Q(v, \alpha)$ are not explicitly stored in the corresponding node $u(\alpha)$. Instead, the information is stored in the path from $u(\alpha)$ to the root of $A(v)$. Each node $u(\alpha)$ contains five fields: q, c, qa, ca and ra. When qa, ca and ra are all 0, q and c give $Q(v, \alpha)$ and $C(v, \alpha)$, respectively. Fig. 7 is an example candidate tree where qa, ca and ra fields are all 0.

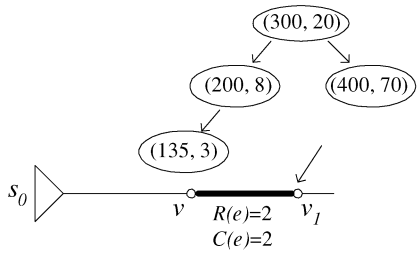


Fig. 7. Candidate tree $A(v_1)$ of four candidates. Fields qa, ca, and ra are 0 for all candidates.

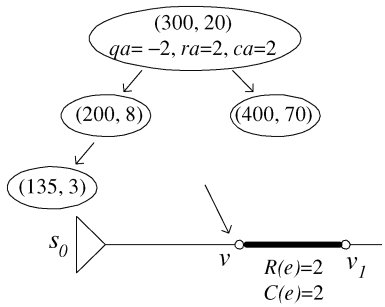


Fig. 8. Candidate tree $A(v)$ of four candidates after the wire is added.

Assume $qa = 0, ca = 0$, and $ra = 0$ for the root. When edge e is added, the following information is inserted to the fields of the root:

1) $ca = C(e)$, meaning that C of every candidate in the tree will be increased by $C(e)$;
2) $qa = -R(e)C(e)/2$, meaning that Q of every candidate in the tree will be decreased by $R(e)C(e)/2$;
3) $ra = R(e)$, meaning that Q of every candidate α in the tree will be decreased by $R(e) \cdot C(v, \alpha)$, where $C(v, \alpha)$ is the value before adding edge e.

The implicit representation is used recursively on each node in the candidate tree. The actual update of C and Q for each candidate will take place later, whenever that candidate is visited. This delayed update can save a great amount of computation time.

In general, let α be a candidate of $T(v)$, $u(\alpha)$ be the node for α in $A(v)$, u_1 be the root of $A(v)$, and $u_1, u_2, \ldots, u_k = u(\alpha)$, be the path from the root to $u(\alpha)$. Then

$$C(v, \alpha) = c(u_k) + \sum_{i=1}^{k} ca(u_i)$$
$$Q(v, \alpha) = q(u_k) - \sum_{i=1}^{k} ra(u_i) \left(c(u_k) + \sum_{j=i+1}^{k} ca(u_j) \right) + \sum_{i=1}^{k} qa(u_i).$$

Fig. 8 shows the candidate tree after adding a wire (v, v_1) in Fig. 7.

The following C code defines the data structure of each candidate tree node.

```
typedef struct A_node {
  float q, c, qa, ca, ra;
  struct TypeLoc *b, *ba;//type & location
  char dirty; // whether to update
  int size; // candidates in subtree
  struct A_node *left, *right;
  struct L_node *l;// to expiration list
  char color; // for red-black tree
} A_node;
```

Although the definition of C and Q is recursive, the values can be computed in $O(1)$ time for each candidate, whenever each candidate is visited. The search of a candidate tree is similar to the search of any binary search tree. The only difference is that when a node is dirty, fields c and q will be updated to give the current value of C and Q, and fields qa, ca and ra are propagated one level down to the children. The delayed propagation is crucial to the reduction of the running time. The following C code illustrates the update process. Function update (x) updates all fields of node x, and propagates information to the children. It reflects how (1) is evaluated.

```
void update(A_node *x) {
  // propagate to left subtree
  x->left->qa = x->left->qa + x->qa
      -(x->ra) * (x->left->ca);
  x->left->ca = x->left->ca + x->ca;
  x->left->ra = x->left->ra + x->ra;
  x->left->dirty = TRUE;
  // propagate to right subtree
  x->right->qa = x->right->qa + x->
      qa  - (x->ra) * (x->right->ca);
  x->right->ca = x->right->ca + x->
      ca;
  x->right->ra = x->right->ra + x->
      ra;
  x->right->dirty = TRUE;
  // update x
  x->q = x->q + x->qa - x->ra*x->c;
  x->c = x->c + x->ca;
  x->ca = x->qa = x->ra = 0;
  x->dirty = FALSE;
}
```

Fig. 9 is an example showing how the candidate tree is updated when node $(200, 8)$ is visited.

The following C code illustrates the search. Function search(x, y) searches a candidate tree with node x being the root, for a node $u(\alpha)$ such that $Q(v, \alpha) =$ y. For simplicity, we illustrate a recursive version, though the implemented algorithm is nonrecursive [5].

```
A_node * search(A_node * x, float y) {
  if (x == NIL)
    return NIL;
```

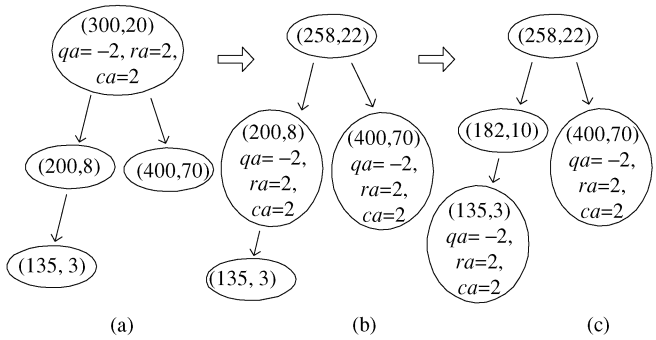


Fig. 9. Update of candidate tree $A(v)$ when some nodes are visited.

```
  if (x->dirty == TRUE)
    update(x);
  if (x->q == y)
    return x; // found
  else if (x->q >y)
    return search(x->left, y);
  else
    return search(x->right, y);
}
```

Note that whenever a node is visited, the path from root to that node is "cleaned up," meaning that every node on this path is not dirty.

C. Buffer Location and Type

In the original van Ginneken's algorithm [14], the (Q, C) lists are stored at each node in the bottom-up phase. After the best slack is found, the buffer locations and types for the best candidate are determined in the top-down phase by recomputing the partial solutions. Therefore, van Ginneken's algorithm uses $O(n^2)$ memory since each (Q, C) list may take $O(n)$ storage, and there are n such lists.

In our algorithm, we use the candidate trees to store buffer location and type information in memory $O(n)$ for 2-pin nets, and $O(n \log n)$ for multipin nets. This is a significant reduction over the traditional van Ginneken's algorithm that uses $O(n^2)$ memory.

Similar to the fields of Q and C, the location and type are implicitly stored. For each candidate α, the information is stored in the path from the root to $u(\alpha)$. In the above definition of A_node, there are two pointers b and ba of type TypeLoc, which is defined as follows.

```
typedef struct TypeLoc {//type & location
  int btype; // buffer type
  int bloc; // buffer location
  int used; // number of times used
  struct TypeLoc *left, *right;
} TypeLoc;
```

Assume we create a new candidate β from candidate α and a new buffer b_i at position v_j. Let x point to the candidate tree node for β and y point to the candidate tree node for α. Furthermore assume y->b contains the type and location of buffers in

α, and y−>ba is empty. Then, the following process will create the type and location information for β.

```
TypeLoc *p;
p = malloc (sizeof (TypeLoc));
p->btype = bi;
p->bloc = vj;
p->left = y->b;
p->right = NULL;
x->b = p;
x->ba = NULL;
p->used = 1;
y->b->used++;
```

Since α may contain $O(n)$ buffers, any explicit recording of the the types and locations of these buffers will require $O(n)$ memory. However in our algorithm, we simply use one pointer x−>b−>left to share the buffer information from α, thereby using only $O(1)$ memory. The p−>used field is to keep track how many candidates point to p. When a candidate that references p is deleted, p−>used will be decreased by 1. When p−>used equals 0, we delete p.

Now, assume we create a new candidate β by merging candidates α_1 and α_2. Let x point to candidate tree node for β and y1, y2 point to the candidate tree nodes for α_1 and α_2 respectively. Then, we do the following to store the buffer type and locations of β.

```
TypeLoc *p;
p = malloc (sizeof (TypeLoc));
p->btype = MERGE;
p->bloc = NULL;
p->left = y1->b;
p->right = y2->b;
x->b = p;
x->ba = NULL;
p->used = 1;
y1->b->used++;
y2->b->used++;
```

Field ba is used for more complicated merging. Let x point to a node $u(\beta)$ in the candidate tree and assume x−>ba field is nonempty. Then, every candidate in the subtree with $u(\beta)$ being the root is associated with the buffer types and locations of x−>ba. The following C code illustrates additional work of update(x) to update the buffer type and location. The omitted part was shown earlier.

```
void update(A_node*x) {
  TypeLoc *p;
  // propagate to left subtree
  ...
  p = malloc(sizeof(TypeLoc));
  p->left = x->ba;
  p->right = x->left->ba;
  p->btype = MERGE;
  x->ba->used++;
  x->left->ba->used++;
  x->left->ba = p;
  p->used = 1;
  // propagate to right subtree
  ...
  p = malloc(sizeof(TypeLoc));
  p->left = x->ba;
  p->right = x->right->ba;
  p->btype = MERGE;
  x->ba->used++;
  x->right->ba->used++;
  x->right->ba = p;
  p->used = 1;
  // update x
  ...
  p = malloc(sizeof(TypeLoc));
  p->left = x->ba;
  p->right = x->b;
  p->btype = MERGE;
  x->ba->used++;
  x->b->used++;
  x->b = p;
  p->used = 1;
  x->ba = NULL;
}
```

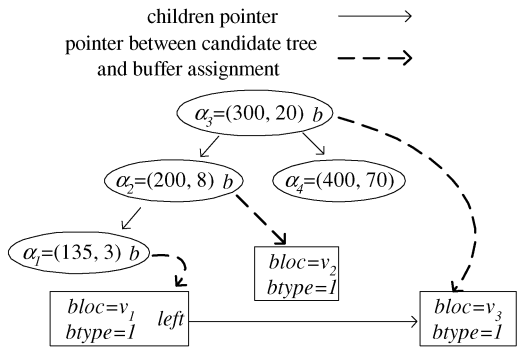


Fig. 10. Four candidates with their buffer types and locations: α_4 has no buffer, α_3 has one buffer at v_3, α_2 has one buffer at v_2, and α_1 consists of α_3 and a buffer at v_1 as shown in Fig. 7.

The following C code illustrates how the buffer assignment is retrieved. Function report(y) prints the buffer type and location information of TypeLoc pointer y.

```
void report(TypeLoc *y) {
  if (y == NULL)
    return;
  if (y->btype != MERGE)
    printf("buffer type %d location
      %d\n", y->btype, y->bloc);
  report(y->left);
  report(y->right);
}
```

Fig. 10 is an example showing how the buffer assignment are stored in the candidate tree.

D. Fast Redundancy Check

For every $A(v)$, we also maintain an *expiration list* $L(v)$ to tell if a candidate in $A(v)$ is redundant under predictive pruning when a wire is added to v. Let $A(v)$ contain nonredundant candidates $\alpha_1, \ldots, \alpha_n$ in increasing C and Q order. The expiration list $L(v)$ contains $l_2, \ldots, l_n$, where

$$l_i = \frac{Q(v,\alpha_i) - Q(v,\alpha_{i-1})}{C(v,\alpha_i) - C(v,\alpha_{i-1})} - R(b). \tag{2}$$

Intuitively, l_i is the threshold such that with such a resistance added, α_i is dominated by α_{i-1}.

Lemma 3: Let α_1 and α_2 be two nonredundant candidates of $T(v_1)$, where $Q(v_1,\alpha_1) < Q(v_1,\alpha_2)$ and $C(v_1,\alpha_1) < C(v_1,\alpha_2)$. Define l_2 according to (2). If we attach an edge $e = (v, v_1)$ at $T(v_1)$, then α_2 is b-dominated by α_1 for $T(v)$ if and only if $R(e) \geq l_2$.

Proof: For $i = 1$ or 2

$$\begin{aligned} P(v,\alpha_i) &= Q(v,\alpha_i) - K(b) - R(b)\cdot C(v,\alpha_i) \\ &= Q(v_1,\alpha_i) - R(e)\cdot\big(C(v_1,\alpha_i) + C(e)/2\big) \\ &\quad -K(b) - R(b)\cdot\big(C(v_1,\alpha_i) + C(e)\big). \end{aligned}$$

Therefore,

$$\begin{aligned} P(v,\alpha_1) - P(v,\alpha_2) &= Q(v_1,\alpha_1) - Q(v_1,\alpha_2) \\ &\quad +\big(R(e) + R(b)\big)\cdot\big(C(v_1,\alpha_2) - C(v_1,\alpha_1)\big). \end{aligned}$$

Hence, $P(v,\alpha_1) \geq P(v,\alpha_2)$ if and only if $R(e) \geq l_2$. On the other hand, we always have

$$C(v,\alpha_2) - C(v,\alpha_1) = C(v_1,\alpha_2) - C(v_1,\alpha_1) > 0.$$

Therefore, α_2 is b-dominated if and only if $R(e) \geq l_2$. ■

$L(v)$ is also organized as a balanced search tree in increasing l order. The following C code defines the data structure for each expiration list node.

```
typedef struct L_node {
  float l;      // threshold
  float la;     // additional info
  struct A_node*a;      // to candidate
tree
  char dirty;      //whether to update
  struct L_node *left, *right;
} L_node;
```

Using balanced search trees or priority queues, finding the minimum l_i, insertion and deletion of any l_i can be done in $O(\log n)$ time. Similar to the candidate tree, if a node is dirty, `la` is added to `l` and propagated to `la` of the two children. Note the cross reference with the candidate tree.

Figs. 11–13 are examples showing how the candidate tree and expiration list change when a wire is added.

E. Fast Merge

The case for merge in Fig. 5 is more involved. Assume we have computed all nonredundant candidates for $T(v_1)$ and $T(v_2)$, and stored the results in $A(v_1)$, $L(v_1)$, $A(v_2)$, and $L(v_2)$ respectively. Now we want to merge $T(v_1)$ and $T(v_2)$ to form $T(v)$. Let the number of candidates in $T(v_1)$ and $T(v_2)$ be n_1 and n_2, and assume without loss of generality $n_1 \geq n_2$.

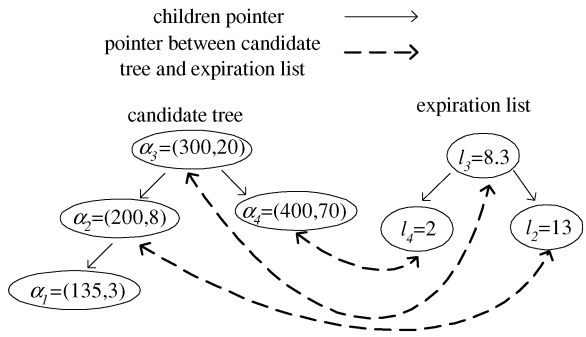


Fig. 11. Candidate tree and expiration list before adding a wire.

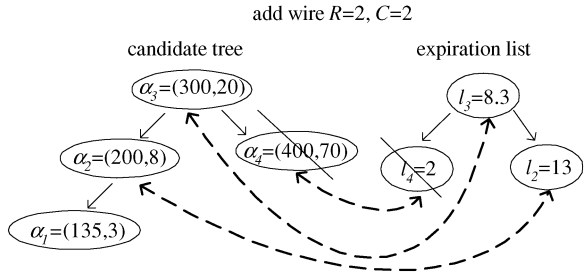


Fig. 12. After adding a wire with $R = 2, C = 2$, (400, 70) is redundant.

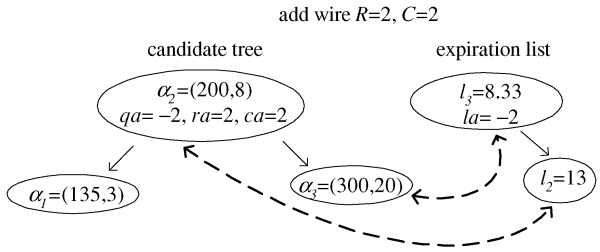


Fig. 13. Final candidate tree and expiration list.

First, we generate nonredundant candidates of $T(v)$ whose Q are decided by $T(v_2)$. For each candidate α_i in $A(v_2)$, we want to find a candidate β_j in $A(v_1)$ such that $Q(v_1,\beta_j) \geq Q(v_2,\alpha_i)$, and $C(v_1,\beta_j)$ is the minimum among all such β_js. This can be done by n_2 searches to $A(v_1)$ in total time $O(n_2 \log n_1)$. The result candidates are stored in a list Z.

Then, we generate nonredundant candidates of $T(v)$ whose Q are decided by $T(v_1)$. We will turn candidate tree $A(v_1)$ to store these new candidates, using field ca. For each candidate α_i in $A(v_2)$, the candidates that can be combined with α_i form an interval in $A(v_1)$. The interval boundaries can be found through two searches of $A(v_1)$, and updates can be made to the boundaries. The total time is also $O(n_2 \log n_1)$.

Finally, we insert list Z of size $O(n_2)$ into the modified candidate tree $A(v_1)$ of size $O(n_1)$. We also check redundancy, and update expiration list. When we finish, candidate tree $A(v_1)$ is $A(v)$. The total time is $O(n_2 \log n_1)$.

Figs. 14–17 are an example of the fast merge process. For simplicity, fields qa, ra, ca of all candidates are initially 0.

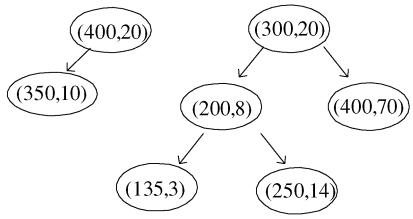


Fig. 14. Two candidates trees $A(v_1)$ (right) and $A(v_2)$.

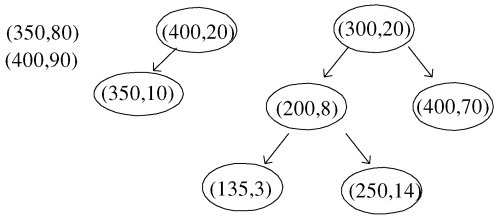


Fig. 15. List Z of candidates of $T(v)$ whose Q are decided by $T(v_2)$.

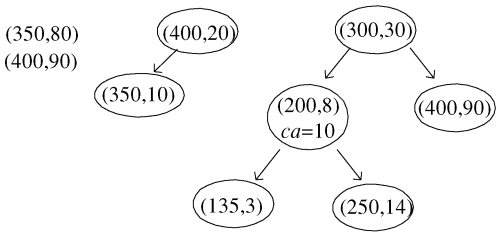


Fig. 16. Candidate tree $A(v_1)$ now stores candidates of $T(v)$ whose Q are decided by $T(v_1)$.

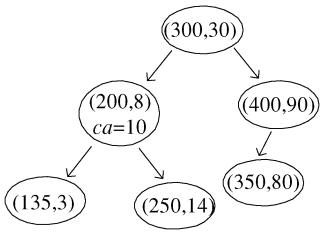


Fig. 17. Insert candidates in Z to the updated candidate tree and delete redundancy. Final candidate tree.

IV. Algorithm

We will compute all nonredundant candidates $N(s_0)$ for the given tree $\mathbf{T}$. Our fast buffer insertion (FBI) algorithm starts from the sinks, and builds nonredundant candidates bottom-up.

Algorithm FBI(v).
Input Routing tree $T(v)$ with root v.
Output Candidate tree $A(v)$ that contains all nonredundant candidates of $T(v)$.
Begin
1: **If** v is a sink **then**
  Create a candidate tree $A(v)$ to store the only candidate of $T(v)$;
 Return $A(v)$.
2: **Else if** $T(v)$ consists of edge (v, v_1) and $T(v_1)$ **then** $A(v_1) \leftarrow \mathrm{FBI}(v_1)$;
  Modify $A(v_1)$ to include delay due to wire (v, v_1);
  Delete redundancy;
 Return the modified $A(v_1)$.
3: **Else if** $T(v)$ consists of buffer position v and $T(v_1)$ **then** $A(v_1) \leftarrow \mathrm{FBI}(v_1)$;
  Find candidate α in $A(v_1)$ that has max $Q(v_1, \alpha)$;
  Form a new candidate and insert it into $A(v_1)$;
  Delete redundancy;
  Return the modified $A(v_1)$.
4: **Else** $T(v) = T(v_1) \cup T(v_2)$
  $A(v_1) \leftarrow \mathrm{FBI}(v_1); A(v_2) \leftarrow \mathrm{FBI}(v_2)$;
  Assume without loss of generality $|A(v_1)| \geq |A(v_2)|$;
4.1: $Z \leftarrow$ nonredundant candidates of $T(v)$ whose Q are determined by $T(v_2)$;
4.2: Compute nonredundant candidates of $T(v)$ whose Q are determined by $T(v_1)$;
  Change $A(v_1)$ to store the resulting candidates;
4.3: Insert Z into $A(v_1)$ and delete redundancy;
  Return the modified $A(v_1)$.
End of Algorithm.

We now explain the details.

A. Sink

If T is sink s_i, then we create a candidate tree $A(s_i)$ that contains only one node. Let x be the pointer point to the root, then the fields are set as follows.

```
x->c = C(si);
x->q = RAT(si);
x->qa = x->ca = x->ra = 0;
x->dirty = FALSE;
```

The expiration list $L(s_i)$ is empty.

B. Buffer

Consider the case in Fig. 3, where $f(v) = \{b\}$ and wire (v, v_1) has zero resistance and capacitance. Assume all n_1 nonredundant candidates for $T(v_1)$ have been computed and stored in candidate tree $A(v_1)$, and a corresponding expiration list $L(v_1)$ is created.

If we do not add a buffer at v, then all nonredundant candidates for $T(v_1)$ become nonredundant candidates for $T(v)$. If we add a buffer at v, then there is a new candidate β such that

$$Q(v, \beta) = \max_{1 \leq i \leq n_1} \{Q(v_1, \alpha_i) - R(b) \cdot C(v_1, \alpha_i) - K(b)\},$$

and $C(v, \beta) = C(b)$. From Lemma 2, β can be found in $O(1)$ time from $A(v)$. Once we form β, we search $A(v_1)$ for α_i and α_{i+1} such that $C(v, \alpha_i) \leq C(v, \beta) \leq C(v, \alpha_{i+1})$. Then check if β is b-dominated by α_i, and if β b-dominates α_{i+1}. If β is b-dominated by α_i, delete β. If β b-dominates α_{i+1}, insert β

into $A(v_1)$ in $O(\log n_1)$ time and delete α_{i+1}, and check α_{i+2}, etc. Each deletion can be done in $O(\log n_1)$ time. We will discuss time for deletion in Theorem 1.

The insertion of β between α_i and α_{i+1} will cause the following updates to $L(v)$: Delete old l_{i+1}, and insert two new ls corresponding to α_i, β and β, α_{i+1}, respectively. This can be done in $O(\log n_1)$ time.

C. Wire

Consider the case in Fig. 4, where $e = (v, v_1)$ is a wire. Assume all n_1 nonredundant candidates for $T(v_1)$ have been computed and stored in candidate tree $A(v_1)$, and a corresponding expiration list $L(v_1)$ is created.

Each candidate α_i of $T(v_1)$ with wire $e = (v, v_1)$ is a new candidate β_i for $T(v)$. We modify the root x of $A(v_1)$.

```
if (x->dirty == TRUE)
  update(x);
x->ca = C(e);
x->qa = -R(e)*C(e)/2;
x->ra = R(e);
x->dirty = TRUE;
```

Now, all candidates for $T(v_1)$ become candidates for $T(v)$. Call the new candidate tree $A(v)$.

However, we are not done yet. Wire e may make some βs redundant. We compare $R(e)$ with the minimum l_i in $L(v_1)$. If $R(e) \geq l_i$, according to Lemma 3, the corresponding candidate β_i is redundant and should be deleted from $A(v)$. Repeat the process, until $R(e) < l_i$. Each deletion from $A(v)$ and $L(v_1)$ takes $O(\log n_1)$ time. We will discuss the total deletion time in Theorem 1.

From (2), it can be seen that the addition of e decreases the value of all l_is by $R(e)$. Therefore, we add $-R(e)$ to the la field of the root of $L(v_1)$ in $O(1)$ time. The order of l_is in $L(v_1)$ does not change. This gives us the new expiration list $L(v)$.

D. Merge

Assume we have computed all nonredundant candidates for $T(v_1)$ and $T(v_2)$, and stored the results in $A(v_1), L(v_1), A(v_2)$, and $L(v_2)$ respectively. Now we want to merge $T(v_1)$ and $T(v_2)$ to form $T(v)$.

Let the number of candidates in $A(v_1)$ and $A(v_2)$ be n_1 and n_2 respectively. Assume without loss of generality $n_1 \geq n_2$, otherwise exchange $A(v_1)$ and $A(v_2)$. Field size tells us in $O(1)$ time which tree contains more candidates.

Step 1: Consider nonredundant candidates of $T(v)$ whose Q are decided by $T(v_2)$. We also include nonredundant candidates whose Q are decided by both $T(v_1)$ and $T(v_2)$ simultaneously. For each candidate α_i in $A(v_2)$, we want to find a candidate β_j in $A(v_1)$ such that $Q(v_1, \beta_j) \geq Q(v_2, \alpha_i)$, and $C(v_1, \beta_j)$ is the minimum among all such β_js. In other words, we want to find index j:

$$j = \min_{1 \leq k \leq n_1} \{k \mid \beta_k \in A(v_1), Q(v_1, \beta_k) \geq Q(v_2, \alpha_i)\}.$$

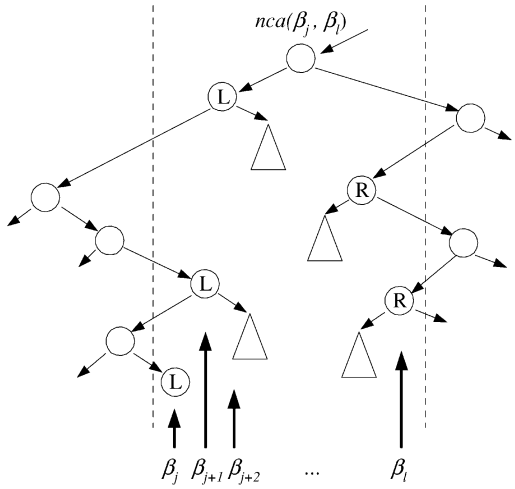


Fig. 18. Nodes $u(\beta_j), u(\beta_{j+1}), \ldots, u(\beta_l)$ in candidate tree $A(v_1)$ form an interval.

Given α_i, we can find the corresponding β_j by searching $A(v_1)$. Together, $\alpha_i \cup \beta_j$ is a candidate of $T(v)$ with slack $Q(v_2, \alpha_i)$ and capacitance $C(v_2, \alpha_i) + C(v_1, \beta_j)$.

To quickly generate all nonredundant candidates of $T(v)$ whose Qs are decided by $T(v_2)$, we traverse every α_i in $A(v_2)$ in increasing Q order, and search $A(v_1)$ for the corresponding β_j. The total time to traverse $A(v_2)$ is $O(n_2)$, and the total time to search $A(v_1)$ is $O(n_2 \log n_1)$. The newly generated candidates are stored in a temporary list Z in increasing Q order for Step 3. The size of Z is at most n_2. Expiration list $L(v_2)$ is freed.

Step 2: Now consider nonredundant candidates of $T(v)$ whose Q are decided by $T(v_1)$. For each candidate α_i in $A(v_2)$, we want to find candidates $\beta_j, \beta_{j+1}, \ldots, \beta_l$ in $A(v_1)$ such that

$$j = \min_{1 \leq k \leq n_1} \{k \mid \beta_k \in A(v_1), Q(v_1, \beta_k) > Q(v_2, \alpha_{i-1})\}$$
$$l = \max_{1 \leq k \leq n_1} \{k \mid \beta_k \in A(v_1), Q(v_1, \beta_k) < Q(v_2, \alpha_i)\}.$$

This can be done through two searches of $A(v_1)$ using $Q(v_2, \alpha_{i-1})$ and $Q(v_2, \alpha_i)$. If no such j and l are found, increment i by 1 and repeat. Otherwise, we form the following $l - j + 1$ candidates of $T(v)$:

$$\alpha_i \cup \beta_j : Q = Q(v_1, \beta_j), C = C(v_1, \beta_j) + C(v_2, \alpha_i)$$
$$\cdots$$
$$\alpha_i \cup \beta_l : Q = Q(v_1, \beta_l), C = C(v_1, \beta_l) + C(v_2, \alpha_i).$$

To store the newly generated candidates, we change the fields of nodes $u(\beta_j), \ldots, u(\beta_l)$ in $A(v_1)$. Step by step, we will turn $A(v_1)$ into an candidate tree of $T(v)$. However, we cannot afford $O(l - j)$ time to explicitly change the nodes. Instead, we change fields ca. Fig. 18 illustrates the general situation of nodes $u(\beta_j), u(\beta_{j+1}), \ldots, u(\beta_l)$. These nodes form a continuous interval in $A(v_1)$. Let $nca(\beta_j, \beta_l)$ be the *nearest common ancestor* of $u(\beta_j)$ and $u(\beta_l)$. Let the *left boundary* be the set of candidates γ such that $u(\gamma)$ is on the path from $u(\beta_j)$ to $nca(\beta_j, \beta_l)$ and $Q(v_1, \gamma) \geq Q(v_1, \beta_j)$. In Fig. 18, nodes with "L" are the left boundary. Let pointer x point to the node for

α_i. For every left boundary node pointed by u in $A(v_1)$, not including $nca(\beta_j, \beta_l)$, we make the following changes:

```
// c values
u->c = u->c + x->c;
u->right->ca = u->right->ca + x->
c;
u->right->dirty = TRUE;
// buffer type and location
TypeLoc *p;
p = malloc(sizeof(TypeLoc));
p->left = u->b;
p->right = x->b;
p->btype = MERGE;
u->b->used++;
x->b->used++;
u->b = p;
p->used = 1;
p = malloc(sizeof(TypeLoc));
p->left = u->right->ba;
p->right = x->b;
p->btype = MERGE;
u->right->ba->used++;
x->b->used++;
u->right->ba = p;
p->used = 1;
```

Similarly, let the *right boundary* be the set of candidates γ such that $u(\gamma)$ is on the path from $u(\beta_l)$ to $nca(\beta_j, \beta_l)$ and $Q(v_1, \gamma) \leq Q(v_1, \beta_l)$. In Fig. 18, nodes with "R" are the right boundary. For every right boundary node u, not including $nca(\beta_j, \beta_l)$, we make the following changes.

```
// c values
u->c = u->c + x->c;
u->left->ca = u->left->ca + x->c;
u->left->dirty = TRUE;
// buffer type and location
TypeLoc *p;
p = malloc(sizeof(TypeLoc));
p->left = u->b;
p->right = x->b;
p->btype = MERGE;
u->b->used++;
x->b->used++;
u->b = p;
p->used = 1;
p = malloc(sizeof(TypeLoc));
p->left = u->left->ba;
p->right = x->b;
p->btype = MERGE;
u->left->ba->used++;
x->b->used++;
u->left->ba = p;
p->used = 1;
```

Finally, for $\mathrm{nca}(\beta_j, \beta_l)$. Let it be pointed by u. We make the following changes.

```
// c value
u->c = u->c + x->c;
// buffer type and location
TypeLoc *p;
p = malloc(sizeof(TypeLoc));
p->left = u->b;
p->right = x->b;
p->btype = MERGE;
u->b->used++;
x->b->used++;
u->b = p;
p->used = 1;
```

Among the newly generated candidates, no one dominates another. The total search time for β_js and β_ls is $O(n_2 \log n_1)$. It is easy to see all the *ncas* can be found in the same time. The total number of nodes in the left and right boundaries, for all intervals, is at most the number of nodes visited. Therefore, the total time to update fields c and ca for all intervals is $O(n_2 \log n_1)$. The expiration list $L(v_1)$ does not change.

Step 3: Insert list Z of size $O(n_2)$ generated in Step 1 into the candidate tree $A(v_1)$ of size $O(n_1)$ obtained in Step 2. For each α_i in Z, we search α_{j-1}, α_j in $A(v_1)$, such that $C(\alpha_{j-1}) < C(\alpha_i) < C(\alpha_j)$. Then check if α_i is b-dominated by α_{j-1}, and if α_i b-dominates α_j. Delete redundancy if any, then insert α_i into $A(v_1)$. When we finish, candidate tree $A(v_1)$ is $A(v)$.

Since there are $O(n_2)$ searches and $O(n_2)$ insertions, the total time for search and insertion is $O(n_2 \log n_1)$.

The insertion of α_i between α_{j-1} and α_j will cause the following updates to $L(v_1)$: delete old l_j and insert two new ls corresponding to α_{j-1}, α_i and α_i, α_j, respectively. This can be done in $O(n_2 \log n_1)$ time.

V. Analysis

We first prove a fact we need later in the estimation of the time complexity.

Lemma 4: For any node v, if $T(v)$ contains n possible buffer positions, then there are at most $n+1$ nonredundant candidates for $T(v)$.

Proof: By induction on n. When $n = 0$, the lemma is clearly true.

If v is a buffer position and v is connected to subtree $T(v_1)$ by an edge (v, v_1), and $T(v_1)$ contains $n-1$ buffer positions. From the induction hypothesis, $T(v_1)$ has at most n nonredundant candidates. Adding a buffer at v, we can get at most one more nonredundant candidate.

If v is connected to subtrees $T(v_1)$ and $T(v_2)$, where $T(v_1)$ and $T(v_2)$ contain n_1 and n_2 buffer positions respectively, where $n = n_1 + n_2$. From the induction hypothesis, $T(v_1)$ and $T(v_2)$ have at most $n_1 + 1$ and $n_2 + 1$ nonredundant candidates respectively. The Q value of each candidate of $T(v)$ is decided by $T(v_1)$ or by $T(v_2)$ or by both. If the Q value of a candidate of $T(v)$ is decided by an candidate of $T(v_1)$, then there is at most one choice for the candidate of $T(v_2)$, and vice versa.

The value of maximum Q among all candidates in $T(v_1)$ and $T(v_2)$ can not appear in $T(v)$. Therefore, there are at most $(n_1+1)+(n_2+1)-1=n+1$ nonredundant candidates for $T(v)$. ■

Theorem 1: Algorithm FBI correctly finds all nonredundant candidates in worst case time $O(n\log n)$ for 2-pin nets, and $O(n\log^2 n)$ for multipin nets, where n is the number of buffer positions. The worst-case space complexity is $O(n\log n)$.

Proof: The correctness proof is similar to that of van Ginneken's algorithm. From Lemma 1, using b-dominate to prune candidates will produce the same final result as van Ginneken's algorithm. Now consider the time complexity. Assume without loss of generality, the number of edges is the same as the number of sinks m and the number of buffer positions n. Otherwise, we can preprocess the routing tree in time $O(n+m)$ by shrinking any two edges (v_i,v_j) and (v_j,v_k), where v_j is degree 2 and $f(v_j)=\{\emptyset\}$, into one edge (v_i,v_k). Since each wire can be added in $O(1)$ time, we will only consider the time for inserting buffer and merging.

For 2-pin nets, our algorithm has $O(n\log n)$ time complexity since adding a buffer and wire only take $O(\log n)$ time. The space complexity is only $O(n)$ since both the candidate tree and the expiration list have only $O(n)$ element, and the buffer assignment storage is also size $O(n)$ since we use the pointer structure shown earlier to store the assignment and there is no merging operation.

Now consider the multipin nets, which need merging operation. Let $\mathcal{T}(n)$ be the worst case time complexity of the algorithm on search and insertion operations only, where n is the number of buffer positions. From Lemma 4, there are at most $n+1$ nonredundant candidates. Therefore, we have the following recurrence relation:

$$\mathcal{T}(n)\le\begin{cases}c, & \text{if } v \text{ is a sink}\\ \mathcal{T}(n-1)+c\log n, & \text{if } v \text{ is a wire or a buffer position}\\ \max\{\mathcal{T}(n_1)+\mathcal{T}(n_2)+cn_2\log n_1\}, & \text{i } v \text{ is a branch}\end{cases}$$

where c is a constant, n_1 and n_2 are the number of buffer positions of $T(v_1)$ and $T(v_2)$, respectively, and the maximum is taken over all n_1,n_2 such that $n_1+n_2=n$ and $n>n_1\ge n_2>0$. We prove by induction that

$$\mathcal{T}(n)\le cn\log^2 n. \tag{3}$$

Obviously, $\mathcal{T}(0)=0\le c\cdot 1\log^2 1$. Assume (3) is true for all $k<n$, then

$$\begin{aligned}\mathcal{T}(n)&\le\max\{\mathcal{T}(n_1)+\mathcal{T}(n_2)+cn_2\log n_1\}\\&\le\max\{cn_1\log^2 n_1+cn_2\log^2 n_2+cn_2\log n_1\}\\&<\max\{c\log n(n_1\log n+n_2\log n_2+n_2)\}\\&\le\max\{c\log n(n_1\log n+n_2\log(2n_2))\}\le cn\log^2 n.\end{aligned}$$

To show the total time for deletion is $O(n\log n)$, we use an argument known as the amortization. Each deletion uses at most $O(\log n)$ time. From Lemma 4, there are at most n insertions, so there are at most n deletions.

TABLE I
SIMULATION RESULTS FOR A 20-mm 2-PIN NET WITH ONE BUFFER TYPE

Buffer pos. n	Time (sec) VG	Time (sec) New	Speed-up	Mem (MB) VG	Mem (MB) New	Reduction
325	0.09	0.01	9	0.22	0.01	22
1297	1.6	0.06	26.7	3.29	0.05	65.8
5185	27.95	0.32	87.3	51.88	0.20	259.4

TABLE II
SIMULATION RESULTS FOR A 20-mm 2-PIN NET WITH FIVE BUFFER TYPES

Buffer pos. n	Time (sec) VG	Time (sec) New	Speed-up	Mem (MB) VG	Mem (MB) New	Reduction
325	0.11	0.13	0.85	0.22	0.02	11
1297	1.97	0.64	3.08	3.29	0.07	47
5185	33.42	3.08	10.85	51.90	0.26	199.6

TABLE III
SIMULATION RESULTS FOR INDUSTRIAL TEST CASES WITH ONE BUFFER TYPE

Sinks m	Buffer pos. n	Time (sec) VG	Time (sec) New	Speed-up	Mem (MB) VG	Mem (MB) New	Reduction
337	336	0.02	0.01	2.0	0.08	0.02	4.0
	2999	0.44	0.09	4.9	0.86	0.05	17.2
	8364	3.17	0.26	12.2	5.02	0.10	50.2
	13753	8.64	0.44	19.6	13.10	0.16	81.9
1944	1943	0.30	0.09	3.3	0.78	0.06	13.0
	17538	7.07	0.55	12.9	12.18	0.12	101.5
	48729	50.11	1.50	33.4	74.39	0.24	310.0
	79925	140.42	2.55	55.1	189.50	0.35	541.4
2676	2675	0.49	0.15	3.3	0.69	0.09	6.9
	23882	11.44	0.82	14.0	11.22	0.17	66.0
	66327	81.31	2.30	35.4	68.79	0.33	208.5
	108793	224.48	3.93	57.1	174.91	0.48	364.4
12052	12051	2.45	0.61	4.0	1.54	0.34	4.5
	104128	58.07	3.21	18.1	18.33	0.47	39.0
	288337	412.21	8.78	46.9	113.36	0.72	157.4
	472591	1230.61	14.88	82.7	288.37	0.97	297.3

The space complexity $\mathcal{S}(n)$ is bounded by $O(n\log n)$, due to the fact that the number of nodes in the boundary in Fig. 18 is $O(n_2\log(n_1/n_2+1))$ as shown in Brown and Tarjan [3].

$$\mathcal{S}(n)\le\begin{cases}c, & \text{if } v \text{ is a sink}\\ \mathcal{S}(n-1)+c\log n, & \text{if } v \text{ is a wire or a buffer position}\\ \max\{\mathcal{S}(n_1)+\mathcal{S}(n_2)+cn_2\log(n_1/n_2+1)\}, & \text{if } v \text{ is a branch.}\end{cases}$$

Using a similar induction, it can be shown the space complexity $\mathcal{S}(n)=O(n\log n)$. However, if we just compute the (C,Q) pairs instead of the buffer locations, then the space complexity can be reduced to $O(n)$ by omitting fields related to the buffer locations. ■

VI. MULTIPLE BUFFER TYPES

For multiple buffer types, the (P,C) pruning is defined for each type of buffer b_i: $P_i(v,\alpha)=Q(v,\alpha)-R(b_i)\cdot C(v,\alpha)-K(b_i)$. In other words, $P_i(v,\alpha)$ is the slack before an imaginary buffer of type b_i at v. For any two candidates α_1 and α_2 of $T(v)$, we say α_1 b_i-dominates α_2 if $P_i(v,\alpha_1)\ge P_i(v,\alpha_2)$ and $C(v,\alpha_1)\le C(v,\alpha_2)$. For each buffer type b_i, there will be one candidates tree $A_i(v)$ to store candidates of $T(v)$ that

TABLE IV
SIMULATION RESULTS FOR INDUSTRIAL TEST CASES WITH FIVE BUFFER TYPES

Sinks	Buffer	Time (sec)		Speed-	Mem (MB)		Reduc-
m	pos. n	$O(\|B\|^2n^2)$	New	up	$O(\|B\|^2n^2)$	New	tion
337	336	0.08	0.29	0.3	0.12	0.09	1.3
	6178	5.11	3.68	1.4	4.58	0.36	12.7
	12514	20.76	7.78	2.7	16.64	0.67	24.8
	24727	84.51	16.22	5.2	60.95	1.24	49.2
1944	1943	0.85	1.66	0.5	1.00	0.21	4.8
	36252	79.15	22.12	3.6	58.34	0.76	76.8
	73679	336.98	48.02	7.0	218.57	1.39	157.2
	145416	1701.97	101.49	16.8	811.10	2.62	309.6
2676	2675	1.21	2.31	0.5	0.96	0.22	4.4
	49315	112.59	31.31	3.6	57.18	0.83	68.9
	100172	490.21	67.50	7.3	214.51	1.54	139.3
	197691	2346.00	141.71	16.6	794.51	2.88	275.9
12052	12051	5.8	9.07	0.64	2.01	0.51	3.9
	214548	539.2	116.69	4.62	89.2	1.16	76.9
	435402	2343.50	258.00	9.08	333.7	2.05	162.8
	858805	13688.36	542.63	25.23	1240.29	3.7	335.2

are nonredundant under (P_i, C) pruning. For each buffer type b_i, there is also one expiration list $L_i(v)$ to tell if a candidate in $A_i(v)$ will be redundant when a wire is attached to v. The algorithm is similar to the algorithm for one buffer type. The differences are explained as follows.

In the sink case, if T is sink s_k, then for every buffer type b_i, we create a candidate tree $A_i(s_k)$ that contains only one node and all $A_i(s_k)$ are same. All expiration lists $L_i(s_k)$ are empty.

In the wire case, let $A_i(v)$ contain nonredundant candidates $\alpha_1, \ldots, \alpha_{n_i}$ in increasing C and Q order. The expiration list $L_i(v)$ contains $l_{i,2}, \ldots, l_{i,n_i}$, where

$$l_{i,j} = \frac{Q(v, \alpha_{i,j}) - Q(v, \alpha_{i,j-1})}{C(v, \alpha_{i,j}) - C(v, \alpha_{i,j-1})} - R(b_i).$$

We need compare $R(e)$ with the minimum $l_{i,j}$ in $L_i(v)$.

In the buffer case, when v is a possible buffer position, then for each buffer type b_i, we need to form a new candidate β_i from $A_i(v)$. For every β_i, it should be inserted to all nonredundant candidates trees and check the redundancy and update the expiration list $L_i(v)$.

In the merge case, for each buffer type b_i, candidate trees $A_i(v_1)$ and $A_i(v_2)$ are merged to form the new candidates tree $A_i(v)$.

Now consider the time complexity. Each step we have to perform $|B|$ times as much work as the single buffer case, and the number of nonredundant candidates is $O(|B|n)$. Therefore, for 2-pin nets, the time complexity is $O(|B| \cdot |B|n \log(|B|n)) = O(|B|^2 n \log n)$. For multipin nets, the time complexity is $O(|B| \cdot |B|n \log^2(|B|n)) = O(|B|^2 n \log^2 n)$.

VII. SIMULATION

Both van Ginneken's algorithm and the new algorithm are implemented in C an run on a Sun SPARC workstations with 400 MHz and 2 GB of memory. The device and interconnect parameters are based on TSMC 180 nm technology. Five different buffer types are used from 1X to 16X. For 1X buffer, $R(b) = 2880\ \Omega, C(b) = 1.5$ fF,$K(b) = 36.4$ ps. For other buffer types, $R(b)$ and $C(b)$ scale accordingly, and intrinsic delay is identical for all buffers. The sink capacitances range from 2 to 41 fF. The wire resistance is $0.076\ \Omega/\mu$m and the wire capacitance is 0.118 fF$/\mu$m. The implemented algorithms include buffer assignments. Table I shows for 2-pin nets with 20 mm long and one buffer type (16X), the new algorithm is 9–87 times faster than van Ginneken's algorithm and uses $1/22$–$1/250$ of memory. Table II shows for 2-pin nets with five buffer types, the new algorithm is 10 times faster than van Ginneken's algorithm and uses $1/200$ of memory. Table III shows for large industrial circuits with one buffer type (16X), the new algorithm is 2–80 times faster than van Ginneken's algorithm and uses $1/4$–$1/500$ of memory. Table IV shows for large circuits with five buffer types, the new algorithm can be 16 times faster than van Ginneken's algorithm and uses $1/300$ of memory.

In both cases, for multiple buffer type, when n is small, the new algorithm is slower than van Ginneken's algorithm due to multiple candidate trees overhead.

VIII. CONCLUSION

We presented a new algorithm for optimal buffer insertion of worst case time $O(n \log n)$ and space $O(n)$ for 2-pin nets, and worst case time $O(n \log^2 n)$ and space $O(n \log n)$ for multipin nets. This is an improvement, both in time and in space, of the classic van Ginneken's $O(n^2)$ time and space algorithm [14]. For multiple buffer types, we also presented an $O(|B|^2 n \log n)$ algorithm for 2-pin nets, and $O(|B|^2 n \log^2 n)$ algorithm for multipin nets. This is an improvement of the previous best $O(|B|^2 n^2)$ algorithm [8]. Simulation results show our new algorithms are significantly faster than van Ginneken's and $O(|B|^2 n^2)$ algorithms for large industrial circuits.

Since van Ginneken's algorithm and its variations are used by most existing algorithms on buffer insertion and buffer sizing, our new algorithm significantly improves the performance of all these algorithms.

ACKNOWLEDGMENT

The authors thank P. K. Agarwal and H. Edelsbrunner for reference and discussion, C. Alpert for providing test examples, and one reviewer for pointing out an error in the early version of the paper.

REFERENCES

[1] C. J. Alpert and A. Devgan, "Wire segmenting for improved buffer insertion," in *Proc. ACM/IEEE Design Automation Conf.*, 1997, pp. 588–593.

[2] C. J. Alpert, C. Chu, G. Gandham, M. Hrkic, J. Hu, C. Kashyap, and S. Quay, "Simultaneous driver sizing and buffer insertion using a delay penalty estimation technique," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 23, no. 1, pp. 136–141, Jan. 2004.

[3] M. R. Brown and R. E. Tarjan, "A fast merging algorithm," *J. ACM*, vol. 26, no. 2, pp. 211–226, 1979.

[4] C. C. N. Chu and D. F. Wong, "A quadratic programming approach to simultaneous buffer insertion/sizing and wire sizing," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 18, no. 6, pp. 787–798, Jun. 1999.

[5] T. Corman, C. E. Leiserson, and R. Rivest, *Introduction to Algorithms*. New York: McGraw-Hill, 1990.

[6] S. Dhar and M. A. Franklin, "Optimum buffer circuits for driving long uniform lines," *IEEE J. Solid-State Circuits*, vol. 26, no. 1, pp. 32–40, Jan. 1991.

[7] M. Kang, W. W.-M. Dai, T. Dillinger, and D. LaPotin, "Delay bounded buffered tree construction for timing driven floorplanning," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 1997, pp. 707–712.

[8] J. Lillis, C. K. Cheng, and T.-T. Y. Lin, "Optimal wire sizing and buffer insertion for low power and a generalized delay model," *IEEE J. Solid-State Circuits*, vol. 31, no. 3, pp. 437–447, Mar. 1996.

[9] T. Okamoto and J. Cong, "Buffered Steiner tree construction with wire sizing for interconnect layout optimization," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 1996, pp. 44–49.

[10] R. H. J. M. Otten, "Efficient floorplan optimization," in *Proc. IEEE Int. Conf. Comput. Design*, 1983, pp. 499–502.

[11] W. Shi, "A fast algorithm for area minimization of slicing floorplans," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 15, no. 12, pp. 1525–1532, Dec. 1996.

[12] W. Shi, Z. Li, and C. Alpert, "Complexity analysis and speedup techniques for optimal buffer insertion with minimum cost," in *Proc. Conf. Asia South Pacific Design Automation*, 2004, pp. 609–614.

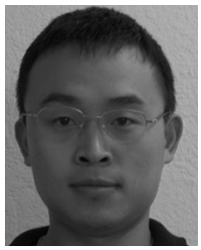
[13] L. Stockmeyer, "Optimal orientations of cells in slicing floorplan designs," *Inform. Control*, vol. 57, pp. 91–101, 1983.

[14] L. P. P. P. van Ginneken, "Buffer placement in distributed RC-tree network for minimal Elmore delay," in *Proc. IEEE Int. Symp. Circuits Syst.*, 1990, pp. 865–868.

[15] H. Zhou, D. F. Wong, I. M. Liu, and A. Aziz, "Simultaneous routing and buffer insertion with restrictions on buffer locations," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 19, no. 7, pp. 819–824, Jul. 2000.

Weiping Shi (S'91–M'92–SM'01) received the B.S. and M.S. degrees in computer science from Xi'an Jiaotong University, Xi'an, China, in 1982 and 1984, respectively, and the Ph.D. degree in computer science from the University of Illinois, Urbana–Champaign, in 1992.

He is an Associate Professor in the Department of Electrical Engineering. Texas A&M University, College Station. His research interests include layout synthesis, parasitic extraction, testing, and the design and analysis of algorithms.



Zhuo Li (S'01) received the B.S. and M.S. degrees in electrical engineering from Xi'an Jiaotong University, Xi'an, China, in 1998 and 2001, respectively. He is currently pursuing the Ph.D. degree in electrical engineering at Texas A&M University, College Station.

He was an Intern with the IBM Austin Research Laboratory, Austin, TX, in 2004. His research interests include interconnect optimization, clock network synthesis, delay testing, and the design and analysis of very large scale integration design automation algorithms.