An $O(mn)$ Time Algorithm for Optimal Buffer Insertion of Nets with m Sinks

Zhuo (Robert) Li and Weiping Shi
Department of Electrical and Computer Engineering
Texas A&M University, College Station, TX 77843

***Abstract*— Buffer insertion is an effective technique to reduce interconnect delay. In this paper, we give a simple $O(mn)$ time algorithm for optimal buffer insertion, where m is the number of sinks and n is the number of buffer positions. This is the first linear time buffer insertion algorithm for nets with constant number of sinks. When m is small, it is a significant improvement over our recent $O(n \log^2 n)$ time algorithm, and the $O(n^2)$ time algorithm of van Ginneken. For b buffer types, the new algorithm runs in $O(b^2n + bmn)$ time, an improvement of our recent $O(bn^2)$ algorithm. The improvement is made possible by a clever bookkeeping method and an innovative linked list data structure that can perform addition of a wire, and addition of a buffer in amortized $O(1)$ time. On industrial test cases, the new algorithm is faster than previous best algorithms by an order of magnitude.**

I. Introduction

Delay optimization techniques for interconnect are increasingly important for achieving timing closure of high performance designs. One popular technique for reducing interconnect delay is buffer insertion. A recent study by Saxena *et al* [1] projects that 35% of all cells will be intra-block repeaters for the 45 nm node. Consequently, algorithms that can efficiently insert buffers are essential for the design automation tools.

This paper studies buffer insertion in interconnect with a set of possible buffer positions and a discrete buffer library. In 1990, van Ginneken [2] proposed an $O(n^2)$ time dynamic programming 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 [3] extended van Ginneken's algorithm to allow b buffer types in time $O(b^2n^2)$. In 2003, Shi and Li [4] used a number of techniques to improve the time complexity to $O(b^2n \log n)$ for 2-pin nets, and $O(b^2n \log^2 n)$ for multi-pin nets. To reduce the quadratic effect of b, Li and Shi [5] recently proposed an algorithm with time complexity $O(bn^2)$. However, all these algorithms do not utilize the fact that in real applications most nets have small numbers of pins and large number of buffer positions. As a result, the running time is still long for large nets, especially when other constraints such as slew and cost are considered.

In this paper, we first propose a new algorithm that performs optimal buffer insertion for 2-pin nets in time $O(b^2n)$. The speedup is achieved by an observation that the best candidate to be associated with any buffer must lie on the convex hull of the (Q, C) plane, a clever bookkeeping method and an innovative linked list that allow $O(1)$ time update for adding a wire or a candidate. The new data structure, which is a linked list, is much simpler than the candidate tree used in [4] and the skip list used in [6]. We then extend the algorithm to m-pin nets in time $O(b^2n+bmn)$. Experimental results show that our algorithm is faster than previous best algorithms by an order of magnitude. Note that all previous research assumed m and n are of the same order. But in fact, m is often much less than n. For example, in one group of nets reported in [13], which is extracted from industrial ASIC chips with 300k+ gates and consists the 5000 most run time consuming nets for buffer insertion, over 90% of nets has less than 50 sinks. For another group with 1000 nets, all of nets has less than 20 sinks. On the other hand, each net has several hundreds to thousands of buffer positions. Note that even if $m > n$, we can merge sinks in a branch that contains no buffer position, without changing the problem. Therefore in this paper we assume $m \leq n$.

Many extensions have been made based on van Ginneken style algorithms, include wire sizing [3], simultaneous tree construction [7], [8], noise constraints [9] and resource minimization [3], [10]. Our new algorithms are fundamental improvements, and are therefore applicable to some of these extensions.

Finally, we note that for 2-pin nets, when there is no restriction on where a buffer is allowed, Dhar and Franklin [11] gave a closed form solution, assuming buffers can be continuously sized. Chu and Wong [12] proposed a convex quadratic programming method to find the optimal buffer insertion location and buffer sizing with discrete set of buffers. However in real applications, buffer blockage is always a serious restriction. Such information should be considered as early as possible to reduce the design cycle. Therefore these algorithms are often used in the very early stage of design planning when buffer blockage information is not available, not in actual physical synthesis. Also for multi-pin nets, no simple closed form solution is available.

The paper is organized as follows. Section II formulates the problem. Section III describes the new algorithm for 2-pin nets. Section IV extends the algorithm to multi-pin nets. Simulation results are given in Section V, and conclusions are drawn in Section VI.

II. Preliminary

A net is a tree $\boldsymbol{T} = (V, E)$, where $V = \{v_0\} \cup V_s \cup V_n$, and $E \subset V \times V$. Vertex v_0 is the *source* vertex and also

This research was supported by the NSF grants CCR-0098329, CCR-0113668, EIA-0223785, ATP grant 512-0266-2001.

0-7803-9451-8/06/$20.00

the root of $\boldsymbol{T}$, 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 $RAT(s)$. A buffer library $\boldsymbol{B}$ contains b types of buffers. For each buffer type $B \in \boldsymbol{B}$, the intrinsic delay is $K(B)$, driving resistance is $R(B)$, and input capacitance is $C(B)$. A function $f : V_n \rightarrow 2^{\mathbf{B}}$ specifies the types of buffers allowed at each buffer position. Each edge $e \in E$ is associated with lumped resistance $R(e)$ and capacitance $C(e)$.

Following previous researchers [2], [3], [4], [7], we use Elmore delay for the interconnect and the linear delay for buffers due to their high fidelity in the synthesis stage. 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. If v_i is inserted a buffer of type B_k, then the buffer delay is

$$D(v_i) = R(B_k) \cdot C(v_i) + K(B_k),$$

where $C(v_i)$ is the downstream capacitance at v_i, and the capacitance viewed from the upper stream is $C(B_k)$.

For any vertex 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 $s_i \in T(v)$ under α is

$$D(v, s_i, \alpha) = \sum_{e=(v_j,v_k)} (D(v_j) + D(e)),$$

where the sum is over all edges e in the path from v to s_i. If a buffer is inserted at v_j in α, then $D(v_j)$ is the buffer delay. Otherwise, $D(v_j) = 0$. The slack of v under α is

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

Max-Slack Buffer Insertion Problem: Given a net $\boldsymbol{T} = (V, E)$, capacitance and RAT for all sinks, capacitance and resistance for all edges, possible buffer position function f, and buffer library $\boldsymbol{B}$, find a candidate α for $\boldsymbol{T}$ that maximizes $Q(v_0, \alpha)$.

The effect of a candidate α for tree $T(v)$ at v to the upstream is traditionally described by slack $Q(v, \alpha)$ and downstream capacitance $C(v, \alpha)$ [2]. 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 $N(v_0)$, the candidate that gives the maximum $Q(v_0, \alpha)$ can be found easily.

III. Two-Pin Nets

In this section, we show how to compute optimal buffer insertion for 2-pin nets in $O(b^2 n)$ time. We use van Ginneken style dynamic programming paradigm, enhanced with two techniques 1) convex pruning to find the best candidate and delete redundancy, and 2) a simple implicit data structure to store and update (Q, C) values. Our data structure is inspired by the candidate tree of Shi and Li [4], but much simpler.

A. Convex Pruning

The concept of convex pruning was first proposed by Li and Shi [5]:

Definition 1: Let α_1, α_2 and α_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

$$\frac{Q(\alpha_2) - Q(\alpha_1)}{C(\alpha_2) - C(\alpha_1)} < \frac{Q(\alpha_3) - Q(\alpha_2)}{C(\alpha_3) - C(\alpha_2)}, \tag{1}$$

then we call α_2 *non-convex*, and prune it.

Convex pruning can be explained by Figure 1. 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 nonredundant candidates $N(v)$ is a monotonically increasing sequence. Candidate $\alpha_2 = (Q_2, C_2)$ in the above definition is shown in Figure 1(a), and is pruned in Figure 1(b). The set of nonredundant candidates after convex pruning $M(v)$ is a convex hull.

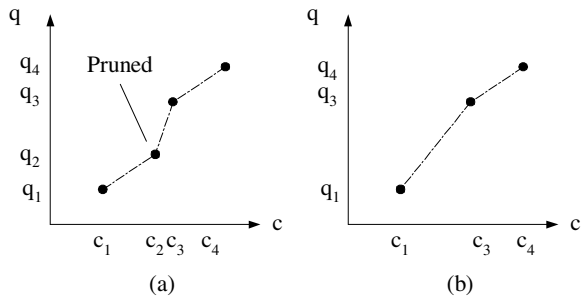

Fig. 1. (a) Nonredundant candidates $N(v)$. (b) Nonredundant candidates $M(v)$ after convex pruning.

Lemma 1: For 2-pin nets, convex pruning preserves optimality.

Proof: Let α_1, α_2 and α_3 be candidates of $T(v)$ that satisfy the condition in Definition 1. In a 2-pin net, every candidate will be connected to some wires, which could be empty, before reaches an upstream buffer or the source driver. Let v' be the upstream buffer or driver, D be the total sum of the delay of wires from v' to v and the delay of the buffer or driver at v' driving wires from v' to v, and R be the sum of the resistance of wires from v' to v and the resistance of the buffer or driver at v'. Then

$$Q(v', \alpha_i) = Q(v, \alpha_i) - R \cdot C(v, \alpha_i) - D,$$

where $i = 1, 2$ or 3. Therefore when

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

we have

$$R(C(v, \alpha_3) - C(v, \alpha_2)) < Q(v, \alpha_3) - Q(v, \alpha_2),$$

and

$$Q(v, \alpha_2) - R \cdot C(v, \alpha_2)) < Q(v, \alpha_3) - R \cdot C(v, \alpha_3).$$

Therefore

$$Q(v', \alpha_2) \quad < \quad Q(v', \alpha_3).$$

On the other hand when

$$R \geq \frac{Q(v,\alpha_3) - Q(v,\alpha_2)}{C(v,\alpha_3) - C(v,\alpha_2)},$$

condition (1) implies

$$R > \frac{Q(v,\alpha_2) - Q(v,\alpha_1)}{C(v,\alpha_2) - C(v,\alpha_1)}.$$

Therefore

$$R(C(v,\alpha_2) - C(v,\alpha_1)) > Q(v,\alpha_2) - Q(v,\alpha_1),$$

which implies

$$Q(v',\alpha_1) \quad > \quad Q(v',\alpha_2).$$

This shows α_2 gives a slack that is worse than either α_1 or α_3 when the source or an upstream buffer is reached. When a buffer is attached, the input capacitance of that buffer will reset $C(\alpha_i)$. Therefore α_2 is redundant. ■

We note that this lemma 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 [5]. 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.

B. 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 add 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 α:

$$P_i(v,\alpha) = Q(v,\alpha) - R(B_i) \cdot C(v,\alpha) - K(B_i). \qquad (2)$$

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 β_i:

$$\begin{aligned} Q(v,\beta_i) &= \max_{\alpha \in N'(v)} \{P_i(v,\alpha)\}, \\ C(v,\beta_i) &= C(B_i). \end{aligned}$$

Define the *best candidate* for B_i as the candidate $\alpha \in N'(v)$ such that α maximizes $P_i(v,\alpha)$ among all candidates in $N'(v)$. If there are multiple α's that maximize $P_i(v,\alpha)$, choose the one with minimum C. From Lemma 1, 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 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. Now add wire e to each candidate α_j and denote it as $\alpha_j + e$. For any buffer type B_i, if α_j gives the maximum $P_i(\alpha_j)$ and α_k gives the maximum $P_i(\alpha_k + e)$, then $k \leq j$.

Proof: From the definition,

$$\begin{aligned} P_i(\alpha_j + e) &= Q(v,\alpha_j + e) - R(B_i)C(v,\alpha_j) \\ &\quad -R(B_i)C(e) - K(B_i) \\ &= P_i(\alpha_j) - R(e)C(\alpha_j) \\ &\quad -R(e)C(e)/2 - R(B_i)C(e). \end{aligned}$$

Since $P_i(\alpha_j + e) \leq P_i(\alpha_k + e)$, we have

$$P_i(\alpha_j) - R(e)C(\alpha_j) \leq P_i(\alpha_k) - R(e)C(\alpha_k),$$

which is equivalent to

$$P_i(\alpha_j) - P_i(\alpha_k) \leq R(e)(C(\alpha_j) - C(\alpha_k)).$$

On the other hand, $P_i(\alpha_j) \geq P_i(\alpha_k)$ and $R(e) > 0$, therefore

$$C(\alpha_j) - C(\alpha_k) \geq 0.$$

This implies $k \leq j$. ■

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

Lemma 3: 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 α_j is the best candidate for buffer type B_i and

$$\begin{aligned} &P_i(\alpha_1) \leq \cdots \leq P_i(\alpha_{j-1}) \leq P_i(\alpha_j), \\ &P_i(\alpha_j) \geq P_i(\alpha_{j+1}) \geq \cdots \geq P_i(\alpha_k). \end{aligned}$$

Proof: From $P_i(\alpha_{j-1}) \leq P_i(\alpha_j)$, we have

$$Q(\alpha_{j-1}) - R(B_i)C(\alpha_{j-1}) \leq Q(\alpha_j) - R(B_i)C(\alpha_j).$$

Therefore,

$$R(B_i) \leq \frac{Q(\alpha_j) - Q(\alpha_{j-1})}{C(\alpha_j) - C(\alpha_{j-1})}.$$

Since all candidates are convex, (1) is false. Hence

$$R(B_i) \leq \frac{Q(\alpha_{j-1}) - Q(\alpha_{j-2})}{C(\alpha_{j-1}) - C(\alpha_{j-2})},$$

which implies $P_i(\alpha_{j-2}) \leq P_i(\alpha_{j-1})$. Then, we can easily get

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

The other direction is similar. From $P_i(\alpha_j) \geq P_i(\alpha_{j+1})$, we have

$$Q(\alpha_j) - R(B_i)C(\alpha_j) \geq Q(\alpha_{j+1}) - R(B_i)C(\alpha_{j+1}).$$

Therefore,

$$R(B_i) \geq \frac{Q(\alpha_{j+1}) - Q(\alpha_j)}{C(\alpha_{j+1}) - C(\alpha_j)}.$$

Since all candidates are convex, (1) is false. Hence

$$R(B_i) \geq \frac{Q(\alpha_{j+2}) - Q(\alpha_{j+1})}{C(\alpha_{j+2}) - C(\alpha_{j+1})},$$

which implies $P_i(\alpha_{j+1}) \geq P_i(\alpha_{j+2})$. We can also easily get

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

Since $P_i(\alpha_j)$ is the maximum $P_i(\alpha)$ among all candidates, α_j is the best candidates for buffer type B_i. ■

C. Data Structure

We store all nonredundant candidates of $T(v)$ in a linked list $L(v)$ of the following data structure:

```
typedef struct Candidate {
    double q, c;
    Candidate *next, *prev;
} Candidate;
```

We also have three global variables:

```
double Qa, Ca, Ra;
```

$L(v)$ is organized in increasing C and Q order, and pruned by convex pruning. The value of Q and C of each candidate α, pointed by `a`, are given by fields `a->q` and `a->c`, as well as global variables `Qa`, `Ca` and `Ra`:

$$\begin{aligned} Q(\alpha) &= (\texttt{a->q}) - \texttt{Qa} - \texttt{Ra} \cdot (\texttt{a->c}), \\ C(\alpha) &= (\texttt{a->c}) + \texttt{Ca}. \end{aligned} \tag{3}$$

To facilitate the search for best candidates and the insertion of new candidates, we have two arrays of pointers:

```
Candidate *best[b], *new[b];
```

where `best[i]` points to the most recent best candidate for B_i, and `new[i]` points to the most recent new candidate for B_i.

D. Algorithm

When we reach an edge `e` with resistance `e->R` and capacitance `e->C`, we update `Qa`, `Ca` and `Qa` to reflect the new values of Q and C of all candidate in L in $O(1)$ time, without actually touching any candidate:

```
void AddWire (e)
{
   Qa = Qa + e->R*e->C/2 + e->R*Ca;
   Ca = Ca + e->C;
   Ra = Ra + e->R;
 }
```

This is similar to Shi and Li's algorithm [4], but much simpler.

When we reach a buffer position, we may generate a new candidate for each buffer type B_i. But first, we have to find the best candidate for B_i. This is done by pointer `best[i]`:

```
void AddBuffer (i)
{
    Candidate *a;
    while (P(i, best[i]->prev) >
      P(i, best[i]))
        best[i] = best[i]->prev;
    ...
```

Function `P(i, ...)` computes P_i of a candidate defined in (2). From Lemma 2, the best candidate is always to the left of where we found the best candidate last time. From Lemma 3, we can find the best candidate by local search. Therefore the `while` loop can find the best candidate that gives the maximum P_i. Now form the new candidate:

```
    ...
    a = new Candidate;
    a->c = B[i]->C - Ca;
    a->q = P(i, best[i]) + Qa + Ra*a->c;
    ...
```

With Eqn. (3), it is easy to verify that the above transformation of `q` and `c` fields will make the new candidate consistent with every other candidate in L. Now insert the new candidate into L:

```
    while (a->c < new[i]->c)
        new[i] = new[i]->prev;
    a->next = new[i]->next;
    new[i]->next->prev = a;
    a->prev = new[i];
    new[i]->next = a;
    ...
```

The location to insert new candidates also moves to the left in L, because the capacitances of all candidates increase when wires are added. Finally, we perform convex pruning around the new candidate:

```
    if (! Convex(a->prev, a, a->next)) {
        a->prev->next = a->next;
        a->next->prev = a->prev;
        Delete(a);
        return;
    }
    while (! Convex(a, a->next,
      a->next->next)) {
        a->next = a->next->next;
        a->next->next->prev = a;
        Delete(a->next);
    }
    while (! Convex(a->prev->prev,
      a->prev, a)) {
        a->prev = a->prev->prev;
        a->prev->prev->next = a;
        Delete(a->prev);
    }
}
```

Function `Convex(...)` checks if the middle candidate is convex. Function `Delete(...)` deletes a candidate, and moves `best` and `new` pointers to the right by one if the pointer points to the candidate to be deleted. Now we describe the entire algorithm:

Algorithm 2-Pin

Input Routing tree $T(v_1)$ consists of path $v_1, \ldots, v_{n+1}$, where v_{n+1} is the sink.

Output Nonredundant candidates of $T(v_1)$ stored in linked list L.

Begin

1: Let `Qa=0, Ca=0, Ra=0;`
2: Let L contain one candidate (Q, C), where $Q = RAT(v_{n+1})$ and $C = C(v_{n+1})$;
3: Let all `best` and `new` pointers point to the only candidate in L;
4: **For** $i = n$ **to** 1 **do**
5: `AddWire(e)`, where $e = (v_i, v_{i+1})$;
6: **For** each buffer type B_j allowed at v_i **do**
7: `AddBuffer(j);`
8: Return L;

End.

Theorem 1: Algorithm 2-Pin finds the optimal buffer insertion of any 2-pin nets in worst-case time $O(b^2n)$.

Proof: The only difference between our algorithm and previous algorithms, other than speedup, is convex pruning. Lemma 1 guarantees convex pruning does not lose the optimality. Therefore our algorithm is correct.

Now consider the time complexity. The outer loop between lines 4 and 7 is executed n times. The inner loop between lines 6 and 7 is executed b times. This requires $O(bn)$ time. In addition, the number of times that any pointers `best[i]` and `new[i]` move equals the total number of candidates, which is bn. Since there are b `best` pointers and b `new` pointers, the total time to move these pointers is $O(b^2n)$. The total deletion time is the same as the number of candidates, which is $O(bn)$. Therefore, the overall time complexity of our algorithm is $O(b^2n)$. ■

Some properties can be used to speed up the implementation, but it does not change the asymptotic time complexity. If buffers are sorted in decreasing driving resistance $R(B_1) \geq R(B_2) \geq \cdots \geq R(B_b)$, and let α_i be the best candidate for B_i. Then it is easy to see that $C(\alpha_1) \geq C(\alpha_2) \geq \cdots \geq C(\alpha_b)$. This helps to reduce the search time for `best` pointers. A similar order can be explored to reduce the search time for `new` pointers.

IV. Multi-Pin Nets

We now extend the 2-pin algorithm to multi-pin nets. In a multi-pin net, a candidate for a 2-pin segment may be merged with a candidate of a different branch, before associated with a buffer. In this case, optimal solution could come from a non-convex candidate. Therefore we need all nonredundant candidates of every 2-pin segment, not only the convex ones.

This is done by a subroutine `2PinSubroutine(...)` for 2-pin segments. The subroutine is similar to Algorithm 2-Pin, but in addition to list $L(v)$, maintains a second list $A(v)$. $A(v)$ contains ALL nonredundant candidates of $T(v)$, including non-convex ones. So $A(v)$ is a superset of $L(v)$. Best candidates are still found through L, yet new candidates are inserted to both L and A. Note that to facilitate the insertion of new candidates into A, another array of pointers `newA[b]` is used and the operation is similar to `new[b]`. For any 2-pin segment $u_1, u_2, \ldots, u_k$, the subroutine takes as input $A(u_k)$, prunes non-convex ones to get $L(u_k)$, and computes each $L(u_i)$ and $A(u_i)$ as it moves to u_1.

Algorithm M-Pin
Input Routing tree $T(v)$ with root v.
Output List $A(v)$ that contains all nonredundant candidates of $T(v)$.
Begin
1: **If** $T(v)$ consists of path v to v_1 where v_1 is a branch vertex **then**
2: Recursively compute $A(v_1)$ for $T(v_1)$;
3: $A(v) =$ `2PinSubroutine`$(A(v_1))$;
4: **Else** $T(v)$ consists of subtrees $T(v_1)$ and $T(v_2)$
5: Recursively compute $A(v_1)$ and $A(v_2)$;
6: Merge $A(v_1)$ and $A(v_2)$ to form $A(v)$;
7: Return $A(v)$;
End.

Theorem 2: Algorithm M-Pin computes the optimal buffer insertion of an m-pin net in time $O(b^2n + bmn)$.

Proof: We compute the same set of all nonredundant candidates as previous algorithms. Therefore the algorithm is correct.

For all 2-pin segments, the total time is bounded by $O(b^2n)$. At each branch vertex, the time is $O(bn)$. Therefore the total time is $O(b^2n + bmn)$. ■

Our new algorithm can be easily integrated with predictive pruning [10], [4], and inverting buffer types [3].

V. Simulation

All algorithms are implemented in C and run on a Sun SPARC workstations with 400 MHz clock and 2 GB memory. The device and interconnect parameters are based on TSMC 180 nm technology and are same as those used in [5] and [4]. We have 4 different buffer libraries, of size 1, 4, 8, and 16 respectively. The value of $R(B_i)$ is from 180 Ω to 7000 Ω, $C(B_i)$ is from 0.7 fF to 23 fF, and $K(B_i)$ is from 29 ps to 36.4 ps. The sink capacitances range from 2 fF to 41 fF. The wire resistance is 0.076 $\Omega/\mu m$ and the wire capacitance is 0.118 fF$/\mu m$.

Table I shows for a 2mm long two-pin net with different possible buffer insertion locations, the new algorithm is up to 20 times faster than previous best algorithms. Table II shows for large industrial multi-pin nets where m is as high as 337, the new algorithm is still faster than previous best algorithms. All algorithms generate same slacks.

VI. Conclusion

We presented a new $O(mn)$ algorithm for optimal buffer insertion on nets with m sinks. When m is small, the new algorithm is a significant improvement over the recent $O(n \log^2 n)$ time algorithm [4], and the $O(n^2)$ time algorithm of van Ginneken. Also, the new algorithm is much simpler than the $O(n \log^2 n)$ algorithm. Simulation results show the new algorithm is faster than these algorithms by an order of magnitude. In addition, for large buffer libraries, the new algorithm is faster than recent $O(bn^2)$ algorithm [5]. In the journal version of this paper, we will apply our algorithm to resource minimization to show significant speedup. Since the new algorithm could run for large number of buffer positions and large buffer libraries in just few seconds, synthesis tool can use very refined buffer positions to select the best quality solutions with small amount of run time overhead.

References

[1] P. Saxena, N. Menezes, P. Cocchini, and D. A. Kirkpatrick, "Repeater scaling and its impact on CAD," *IEEE Trans. Computer-Aided Design*, vol. 23, no. 4, pp. 451–463, 2004.

[2] 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.

[3] 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, 1996.

TABLE I

SIMULATION RESULTS FOR A 2MM TWO-PIN NET.

Buffer pos. n	**Library size** b	**CPU Time (sec)** Lillis-Cheng-Lin [3] $O(b^2n^2)$	Shi-Li [4] $O(b^2n\log n)$	Li-Shi [5] $O(bn^2)$	New $O(b^2n)$
404	1	0.02	0.01	0.03	0.001
	4	0.04	0.11	0.04	0.01
	8	0.08	0.41	0.04	0.02
	16	0.14	1.64	0.06	0.04
2044	1	0.51	0.10	0.80	0.01
	4	1.08	0.70	0.84	0.04
	8	1.78	2.50	0.92	0.10
	16	3.28	9.09	1.01	0.21
10404	1	13.70	0.56	21.85	0.05
	4	28.11	4.33	23.01	0.23
	8	46.71	16.18	23.26	0.49
	16	83.97	59.64	23.75	1.10

TABLE II

SIMULATION RESULTS FOR INDUSTRIAL MULTI-PIN TEST CASES.

Sinks m	**Buffer pos.** n	**Library size** b	**CPU Time (sec)** Lillis-Cheng-Lin [3] $O(b^2n^2)$	Shi-Li [4] $O(b^2n\log^2 n)$	Li-Shi [5] $O(bn^2)$	New $O(b^2n+bmn)$
25	107	1	0.01	0.002	0.002	0.002
		4	0.01	0.03	0.01	0.01
		8	0.02	0.16	0.01	0.01
		16	0.05	0.67	0.02	0.03
	1337	1	0.24	0.04	0.14	0.02
		4	1.06	0.48	0.44	0.11
		8	1.95	2.06	0.60	0.20
		16	3.32	8.62	0.78	0.33
	2567	1	0.75	0.08	0.50	0.05
		4	4.08	1.04	1.47	0.19
		8	7.07	4.30	2.07	0.36
		16	12.12	17.94	2.58	0.64
337	337	1	0.02	0.02	0.02	0.03
		4	0.05	0.04	0.04	0.06
		8	0.09	0.75	0.08	0.12
		16	0.19	3.23	0.14	0.20
	5647	1	0.89	0.17	0.41	0.22
		4	2.51	2.03	0.98	0.59
		8	4.46	8.34	1.51	0.98
		16	7.34	31.55	2.03	1.73
	10957	1	3.40	0.34	1.24	0.42
		4	9.29	4.10	2.95	1.16
		8	16.03	16.88	4.44	1.93
		16	26.96	64.59	5.85	3.26

[4] W. Shi and Z. Li, "A fast algorithm for opitmal buffer insertion," *IEEE Trans. Computer-Aided Design*, vol. 24, no. 6, pp. 879–891, 2005.

[5] Z. Li and W. Shi, "An $O(bn^2)$ time algorithm for buffer insertion with b buffer types," in *Proc. Design, Automation and Test in Europe* 2005, pp. 1324–1329.

[6] R. Chen and H. Zhou, "A flexible data structure for efficient buffer insertion, " in *Proc. IEEE Int. Conf. Computer Design* 2004, pp. 216–221.

[7] 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.

[8] M. Hrkic and J. Lillis, "S-tree: a technique for buffered routing tree synthesis," in *Proc. ACM/IEEE Design Automation Conf.* 2002, pp. 578–583.

[9] C. J. Alpert, A. Devgan, and S. T. Quay, "Buffer insertion for noise and delay optimization," in *Proc. ACM/IEEE Design Automation Conf.* 1998, pp. 362–367.

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

[11] 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, 1991.

[12] 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*, vol. 18, no. 6, pp. 787-798, 1999.

[13] Z. Li, C. N. Sze, C. J. Alpert, J. Hu and W. Shi, "Making fast buffer insertion even faster via approximation techniques," in *Proc. Asia South Pacific Design Automation Conf.*, 2005, pp. 13–18.