FAST INTERCONNECT OPTIMIZATION

A Dissertation

by

ZHUO LI

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

December 2005

Major Subject: Computer Engineering

FAST INTERCONNECT OPTIMIZATION

A Dissertation

by

ZHUO LI

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Weiping Shi
Committee Members,	M. Ray Mercer
	Donald K. Friesen
	Duncan M. (Hank) Walker
Head of Department,	Chanan Singh

December 2005

Major Subject: Computer Engineering

ABSTRACT

Fast Interconnect Optimization. (December 2005)

Zhuo Li, B.E., Xi'an JiaoTong University;

M.S., Xi'an JiaoTong University

Chair of Advisory Committee: Dr. Weiping Shi

As the continuous trend of Very Large Scale Integration (VLSI) circuits technology scaling and frequency increases, delay optimization techniques for interconnect are increasingly important for achieving timing closure of high performance designs. For the gigahertz microprocessor and multi-million gate ASIC designs it is crucial to have fast algorithms in the design automation tools for many classical problems in the field to shorten time to market of the VLSI chip. This research presents algorithmic techniques and constructive models for two such problems: (1) Fast buffer insertion for delay optimization, (2) Wire sizing for delay optimization and variation minimization on non-tree networks.

For the buffer insertion problem, this dissertation proposes several innovative speedup techniques for different problem formulations and the realistic requirement. For the basic buffer insertion problem, an $O(n \log^2 n)$ optimal algorithm that runs much faster than the previous classical van Ginneken's $O(n^2)$ algorithm is proposed, where n is the number of buffer positions. For modern design libraries that contain hundreds of buffers, this research also proposes an optimal algorithm in $O(bn^2)$ time for b buffer types, a significant improvement over the previous $O(b^2n^2)$ algorithm by Lillis, Cheng and Lin. For nets with small numbers of sinks and large numbers of buffer positions, a simple $O(mn)$ optimal algorithm is proposed, where m is the number of sinks. For the buffer insertion with minimum cost problem, the problem is first proved to be NP-complete. Then several optimal and approximation techniques are proposed to further speed up the buffer insertion algorithm with resource control for big industrial designs.

For the wire sizing problem, we propose a systematic method to size the wires of general non-tree RC networks. The new method can be used for delay optimization and variation reduction.

To my grandparents, my parents and my wife

ACKNOWLEDGMENTS

I wish to express my great thanks to my advisor Professor Weiping Shi. Professor Shi introduced the world of VLSI design automation to me. I truly appreciate all his moral and financial support. Professor Shi is a wonderful teacher and person to work with. Professor Shi has shared his profound knowledge and professional manner of conducting research. I am very thankful to him for all the time he devoted to scientific discussions with me, as well as for his constant encouragement and friendship.

Many thanks to my dissertation committee members, Professor Donald Friesen, Professor Ray Mercer, Professor Weiping Shi, and Professor Hank Walker. I appreciate very much their invaluable assistance.

Many thanks to Professor Donald Friesen for introducing me to the algorithm world, which provided a firm base for my research work. Many thanks to Professor Ray Mercer for an inspiring course on switch theory, which opened my eyes to VLSI circuit designs and prepared me well to be confident in my research work, and for valuable advice for my career choices. Many thanks to Professor Hank Walker for invaluable guidance and comments on my research work on testing and for the wonderful time I had while attending his courses on testing and CAD tool development. Many thanks to Professor Jiang Hu for the help and comments on my work on physical design and for the great courses on physical design, which introduced Van Ginneken's algorithm to me.

Many thanks to Dr. Charles Alpert of the IBM Austin Research Lab for being my mentor during summers that I spend at the IBM Austin Research Lab. I appreciate very much Dr. Alpert's scientific support. Many thanks to Dr. Sani Nassif of the IBM Austin Research Lab for being a great manager and for sharing his scientific and industrial experiences with me. Also, I would like to thank all of the PLATO team for the wonderful time I had while staying at the IBM Austin Research Lab.

I would like to express my gratefulness to the Applied Materials Corporation for awarding the Applied Materials Fellowship to me.

Special thanks to my friends and fellow graduate students. Thanks to Xiang Lu for the great work on the fault models, simulation and timing analysis contributing to our collaborative research, and for the experience on SPICE and CAD tools. Thanks to Wangqi Qiu for the wonderful work on the testing contributing to our collaborative research. Thanks to Chin Ngai Sze (Cliff) for his work on the approximation techniques for buffer insertion contributing to part of this research. Thanks for Shu Yan for many interesting discussions and wonderful cakes. Thanks to Zhili Zhang and Yong Liu for their rich Unix and Linux experiences. Thanks to Cheng-Ta Chiang for introducing me to Latex and many useful hints on that. Thanks to Zhijun Cai, Frank Qiu, Haiyun You, Chuang He, Le Zou, Wentao Zhao for the making computer engineering group a big family and a wonderful place to work. Thanks to Linli He for making me always welcome at her home.

I wish to thank my grandparents for teaching me the great things that I will enjoy throughout my life. I am very grateful to my parents for their encouragement, support and truly believing in me.

Many thanks to my wife Ying Zhou. Without her support and belief in my work I could not imagine this dissertation being accomplished.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

A. Technology Trends and Background

As the continuous trend of Very Large Scale Integration (VLSI) circuits technology scaling and frequency increasing, interconnect delay becomes a significant bottleneck in system performances [6, 7]. This trend is a result of increased resistance of the interconnect when feature sizes enter the nano-meter era. From International Technology Roadmap for Semiconductors (ITRS) projection, interconnect delay can contribute to more than 50% of the delay when the feature size is beyond 180 nm. As a result, delay optimization techniques for interconnect are increasingly important for achieving timing closure of high performance designs. It is quite popular to apply such optimization techniques several times during the whole design cycle. Therefore, it is crucial to have fast algorithms for many classical problems in the field to shorten the time to market for the gigahertz microprocessors and multi-million gate *Application Specific Integrated Circuit* (ASIC) chips. This requirement becomes more urgent as design size gets larger and the technology scales further. Also, such fast algorithms can even be used in the early design planning stages, such as floorplan evaluation and physical prototyping. The propose of design planning is to provide accurate estimates of design properties (i.e. area, delay, power) for the designers to quickly determine if budgets have been exceeded or feed the results to the downstream tools as constraints. Previous works on design planning such as [8] always use approximation techniques to get closed-form solutions or low polynomial algorithms due to inefficiency of classical algorithms or insufficient information. As the increasing need of accurate estimation

The journal model is *IEEE Transactions on Automatic Control.*

at the planning stages for designers, fast yet optimal algorithms are important.

Buffer insertion (also called *repeater insertion*, is a popular technique to reduce the interconnect delay. 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. One example of buffer insertion is shown in Fig. 1, where the timing is dramatically improved by buffer insertion.

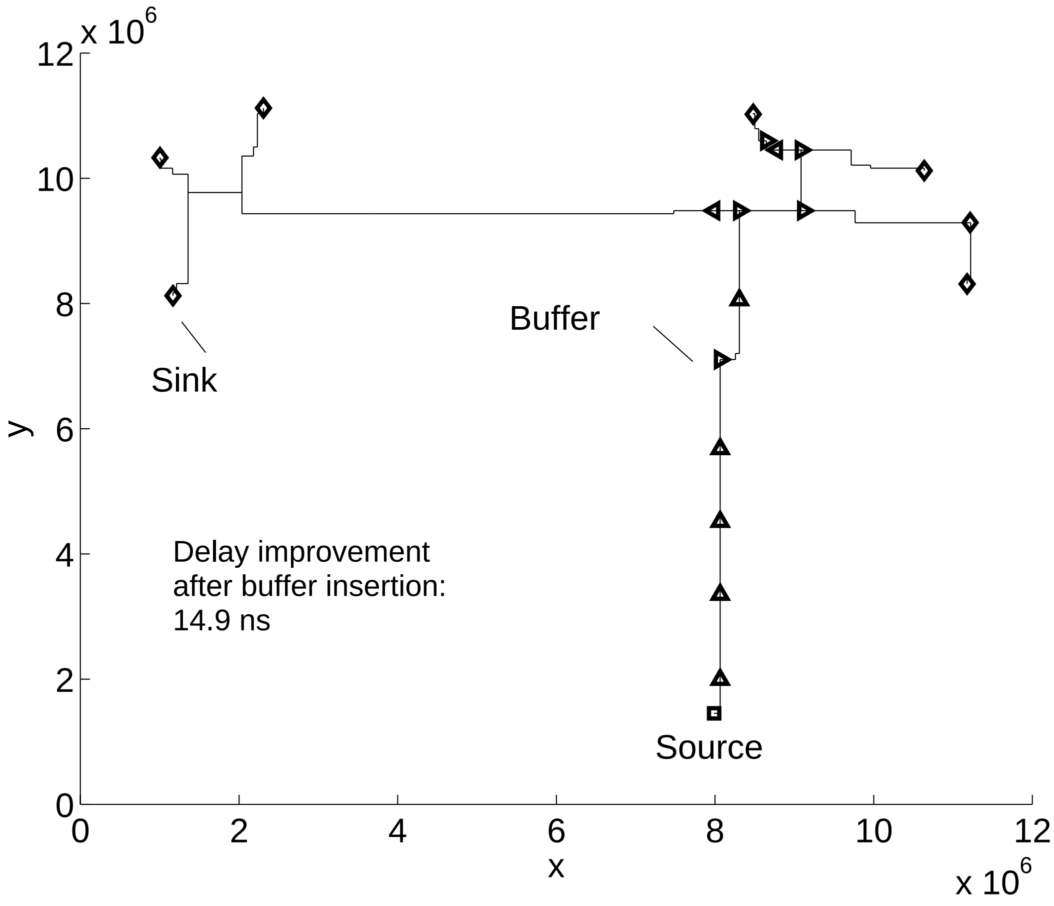


Fig. 1. One 7-sink net extracted from an ASIC chip. Before buffer insertion, the slack of the net is -13.9 *ns*. After 12 buffers inserted, the slack becomes 1.0 *ns*. The delay improvement is 14.9 *ns*.

Owing to the tremendous drop in VLSI feature size, a huge number of buffers are needed for achieving timing objectives for interconnects. It is stated in a recent study [5] that the number of block-level nets that need buffer insertion and the number of buffers will rise dramatically. For example, 12% of block-level nets require buffer insertion and the number of buffers (including clocked buffers) reaches about 15% of the total cell count for intrablock communications for 65nm technology. At 32nm technology nodes, these numbers become 29% and 70% respectively. The trend is shown in Figs. 2 and 3. Although we are not sure whether the number of 70% will finally be reached, hundreds of thousands of buffers can be found in today's ASICs. For example, Osler [9] presents an existing chip with 426 thousand buffers which occupy 15% of the available area. From Figs. 2 and 3, the rate at which the percentage of impacted nets is increasing and the rate at which the percentage of buffers is increasing both start accelerating. Therefore, both the complexity and importance of buffer insertion is increasing in an even faster pace.

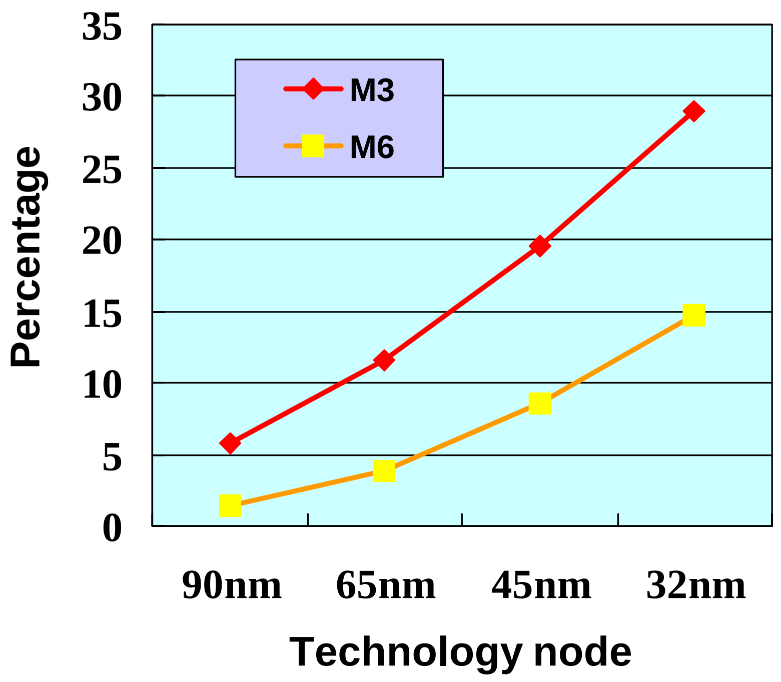


Fig. 2. Percentage of block-level nets requiring buffers [5]. $M3$ and $M6$ represent nets on third and sixth metal layer in a six metal layer technology.

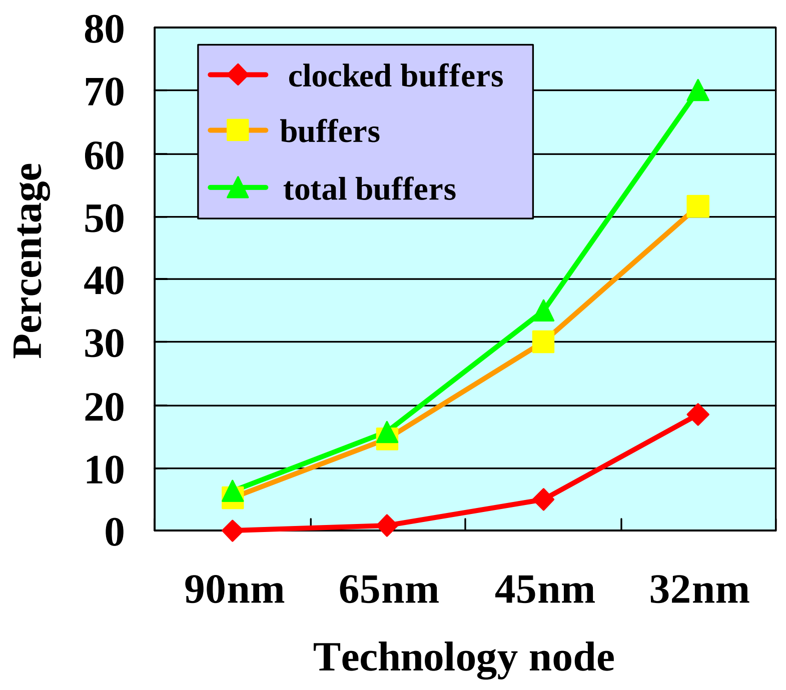


Fig. 3. Intrablock communication repeaters as a percentage of the total cell count for the block.

The increasing number of buffers cause various design problems such as congestion, space and power management. Despite those design problems, it is also a challenge to insert them efficiently and automatically. Even a buffer insertion tool that can process five nets a second requires around 7 hours to process one-hundred thousand nets. An order of magnitude speedup in buffer insertion technologies could enable this task to be accomplished in few minutes while it also enables more design iterations and faster timing closure.

In addition to buffer insertion, wire sizing plays another important role in achieving desirable circuit performance when interconnect delay becomes dominant. Wide wires are now widely used to reduce resistance on critical nets. It can also be tuned to meet clock skew and electromigration targets. Most existing methods for interconnect wire sizing are designed for RC trees. With the increasing popularity of the non-tree topology in clock networks and multiple link networks, wire sizing for non-tree networks becomes an important problem.

B. Contribution

In this thesis, several innovative fast algorithms are proposed for the interconnect optimization. For the basic buffer insertion problem that maximize the required arrival time, a new optimal algorithm that runs much faster than previous classical van Ginneken's $O(n^2)$ algorithm is first proposed, where n is the number of buffer positions. For 2-pin nets, the new algorithm time complexity is $O(n \log n)$ and space complexity is $O(n)$. For multi-pin nets, the 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. Then, we propose an $O(bn^2)$ optimal algorithm for b buffer types is proposed, which is an significant improvement of previous $O(b^2n^2)$ algorithm by Lillis, Cheng and Lin for modern design libraries that generally contain hundreds of buffers. The reduction is achieved by the observation that the (Q, C) pairs of the candidates that generate the new candidates must form a convex hull, where Q and C represents the slack and capacitance of each candidate respectively. Finally, considering in real applications the number of sinks is quite small compared to the number of buffer positions, a simple $O(mn)$ optimal algorithm is proposed, where m is the number of sinks. All the algorithms are quite flexible to be extended to buffer cost minimization and inverting buffer types. Since van Ginneken's algorithm with multiple buffer types are used by most existing algorithms on buffer insertion and buffer sizing, our new algorithms improve the performance of all these algorithms.

Due to the buffer explosion crisis, the basic buffer insertion problem has been modified to minimize buffer cost to become more practical. We first prove that this problem is NP-complete. To improve the practical usage, however, several optimal and approximation techniques are proposed to further speed up the buffer insertion algorithms with cost minimization for big industrial designs. They are motivated from our innovative algorithms for the basic buffer insertion problem. All these techniques make super fast buffer insertion in real industry designs become possible and they can be easily integrated with the current buffer insertion engine which considers slew, noise and capacitance constraints. Consequently, we believe these techniques are essential to embed in a physical synthesis buffer insertion system.

For the wire sizing problem, we propose a new systematic method to size the wires of general non-tree RC networks. Our method consists of three steps: decompose a non-tree RC network into a tree RC network such that the Elmore delay at every sink remains unchanged; size wires of the tree; and merge the wires back to the original non-tree network. All three steps can be implemented in low order polynomial time. Using this method, we can optimized different objectives for non-tree topologies, such as delay optimization w/o area or power constraints and skew variation reduction under process variations, with previous well-developed tree based wire sizing techniques. For certain types of networks, such as the tree+link network [4], our method gives the optimal solution, provided the tree wire sizing is optimal.

The remainder of this thesis is organized as follows. In Chapter II, we first introduce the previous work on buffer insertion. Some preliminary definitions and problem formulations are followed. For the maximizing required arrival time buffer insertion problem, we present three new algorithms to speedup basic van Ginneken's algorithm, big buffer libraries, and small number of sinks respectively. Then the NP-complete proof of the buffer insertion with minimum cost problem is presented and several optimal and approximate speedup techniques are shown. Simulation results are shown for each algorithm. In Chapter III, a new systematic wire sizing approach for non-tree networks is described. Experiments are presented to verify the effectiveness of the new method on delay optimization and variation reduction. Finally, conclusions and some directions for future work are presented in Chapter IV.

CHAPTER II

FAST BUFFER INSERTION FOR DELAY OPTIMIZATION

This chapter presents efficient algorithms for fast buffer insertion for a given routing tree. By utilizing efficient data structures and innovative ideas, we invented several techniques to speed up the buffer insertion problem w/o cost minimization in orders of magnitude.

A. Previous Work

In 1990, van Ginneken [10] proposed a buffer insertion algorithm that is now considered a classic. Given a fixed routing tree, the algorithm inserts buffers in a bottom-up manner to optimize the worst slack to any sink under Elmore delay model [11]. The algorithm has time and space complexity $O(n^2)$, where n is the number of potential insertion points.

Several works have built upon van Ginneken's algorithm. Lillis, Cheng and Lin [1] extended van Ginneken's algorithm to include multiple buffer types and wire sizing. Alpert and Devgan [12] performed wire segmenting to find better buffer positions for van Ginneken's algorithm. Since van Ginneken's algorithm is quadratic in the size of buffer library, Alpert *et al* [13] studied how to reduce the size of the buffer library to make the algorithm practical. A van Ginneken's style algorithm for noise optimization is shown in [14] and higher-order delay models are combined with van Ginnken's algorithm in [15]. Cocchini [16] extends van Ginnken's algorithm to flip-flop (clocked buffers) insertion.

Some researchers consider simultaneous routing tree construction and buffer insertion, which is an NP-hard problem [17]. Early heuristics include Singh and Sangiovanni-Vincentelli [18], and Lin and Marek-Sadowska [19]. Okamoto and Cong [20] combined A-tree construction with van Ginneken's algorithm. Kang *et al* [21] constructed a bounded delay tree, and then used van Ginneken's algorithm to optimize buffers. Zhou *et al* [22] combined the shortest path algorithm with buffer insertion to find the routing path for two-pin net. Recently, Hassoun *et al* [23] extended the algorithm in [22] to the clock domain routing, and Hrkic and Lillis [24, 25, 26] proposed S-tree and SP-tree, which are buffer routed trees considering more constraints.

For buffer insertion on a two-pin net allowing continuous buffer positions, Dhar and Franklin [27] proposed a closed form solution assuming continuous buffer sizes, and Chu and Wong [28] proposed a convex quadratic programming approach with given buffer sizes. However in real applications, buffer blockage is always a serious restriction, which restricts the buffer location. Given the advent of System on Chip (SoC) design and the trends towards large memory arrays, IP cores, and hierarchical design, an ever increasing percentage of the layout is covered by blocks in which buffers cannot be inserted (though routes may cross over). 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. In addition, 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. Moreover, the continuous methods can not be applied to trees.

The performance of most of the above algorithms are limited by the quadratic time complexity of van Ginneken's algorithm, as pointed out by the researchers [12, 1]. For large nets, large number of segments or large buffer libraries, van Ginneken's algorithm becomes the bottleneck.

Furthermore, van Ginneken's algorithm does not control buffering resources and will insert as many buffers as needed to obtain the optimal slack. In practice, this results in a significant over buffering whereby a few picoseconds of performance may be squeezed out for several additional buffers. Also, one frequently wants to find the cheapest solution that meets the timing target, not necessarily the optimal solution in terms of minimal delay. In fact, van Ginneken [10] recognized this, writing, "In addition to the optimization of the timing, the number of buffers used can be optimized. This is done by using triples of numbers rather than pairs for the options." "Unfortunately, this makes the algorithm no longer polynomial." The pairs he referred to are slack Q and capacitance C. Lillis, Cheng and Lin [1] presented an implementation that adds a third element W to control resource utilization, but were unable to claim a polynomial algorithm.

B. Delay Models and Problem Formulations

A net is given as a routing tree $\boldsymbol{T} = (V, E)$, 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 $\boldsymbol{T}$, 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 target required arrival time for source $RAT(s_0)$ is also given. A buffer library $\boldsymbol{B}$ contains different types of buffers and its size is represented by b. For each buffer type $B_i \in \boldsymbol{B}$, the intrinsic delay is $K(B_i)$, driving resistance is $R(B_i)$, input capacitance is $C(B_i)$, and buffer cost . Without loss of generality, we assume the driver at source s_0 is also in B. A function $f : V_n \rightarrow \mathbf{2}^{\boldsymbol{B}}$ specifies the types of buffers allowed at each internal vertex. Each buffer type b_i also has a buffer cost weight $W : \boldsymbol{B} \rightarrow [0, \infty)$. Each edge $e \in E$ is associated with lumped resistance $R(e)$ and capacitance $C(e)$.

Following previous researchers [10, 1, 20, 22, 12], 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 type B_i at vertex v_j, the buffer delay is

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

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

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

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

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

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

The cost of α is the total cost of buffers used by α in $T(v)$:

$$W(v, \alpha) = \sum_{B_i \in \alpha} W(B_i).$$

Note that this cost function definition is quite flexible to represent buffer areas, dynamic and leakage powers or their combinations. The area or power of interconnect edges can also be added.

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

Minimum Cost Buffer Insertion Problem: Given routing tree $\boldsymbol{T} = (V, E)$, sink capacitance $C(s)$ and $RAT(s)$ for each sink s, capacitance $C(e)$ and resistance $R(e)$ for each edge e, possible buffer position f, buffer library $\boldsymbol{B}$, and buffer cost function W, find a candidate α for $\boldsymbol{T}$ that satisfies $Q(s_0, \alpha) \geq RAT(s_0)$ and the cost $W(s_0, \alpha)$ is minimum.

An example of maximum slack buffer insertion problem is shown in Fig. 4 and one of its candidate solutions is shown in Fig. 5.

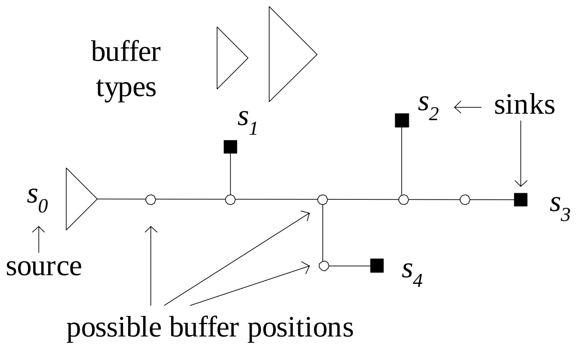


Fig. 4. An example of buffer insertion problem.

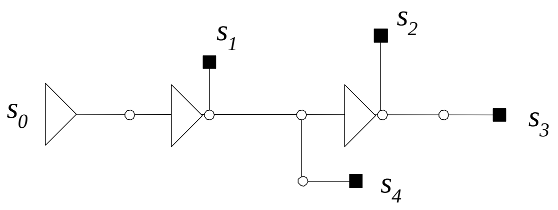


Fig. 5. One candidate solution for Fig. 4.

The effect of a candidate to the upstream is traditionally described by slack Q

and downstream capacitance C [10]. 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.

Assume we have computed nonredundant candidates for $T(v_1)$, and now reach a buffer position v, see Fig. 6. 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:

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

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 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.

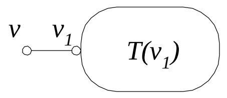


Fig. 6. $T(v)$ consists of buffer position v and $T(v_1)$.

Example II.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. Assume further 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:

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

Therefore, the best candidate to insert buffer is α_2, and the (Q, C) value of the new candidate β is $(135, 3)$. Insert 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.

When a wire $e = (v, v_1)$ is added as shown in Fig. 7, every candidate α for $T(v_1)$ becomes a candidate for $T(v)$, where

$$\begin{aligned} Q(v, \alpha) &= Q(v_1, \alpha) - R(e)C(e)/2 - R(e)C(v_1, \alpha), \\ C(v, \alpha) &= C(v_1, \alpha) + C(e). \end{aligned}$$

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

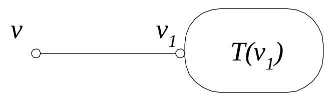


Fig. 7. $T(v)$ consists of wire (v, v_1) and $T(v_1)$.

Example II.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 sub-trees are merged as shown in Fig. 8, 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 $T(v)$:

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

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 $T(v_2)$.

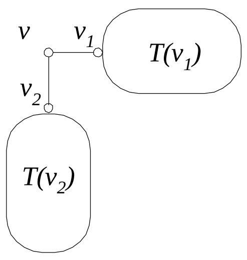


Fig. 8. $T(v)$ consists of $T(v_1)$ and $T(v_2)$.

Example II.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 $T(v_2)$:

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

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

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

C. An $O(n \log^2 n)$ Algorithm for Optimal Buffer Insertion [*]

1. Speedup Techniques

To illustrate the main ideas, we assume for now there is only one non-inverting buffer type B, and s_0 is also driven by a buffer of type B. Extensions to multiple buffer types are in Section 4.

[*]©2005 IEEE. Part of this section is reprinted, with permission, from "A Fast Algorithm for Optimal Buffer Insertion", by W. Shi and Z. Li, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 6, pp. 879-891, June 2005.

a. Predictive Pruning

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

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

among all candidates. However, such a candidate is not necessarily the candidate with maximum Q as shown in Example II.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 II.4. *If α_1 B-dominates α_2, then α_2 is redundant.*

Proof. The general situation is shown in Fig. 9, 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 β's. 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 $Q(v,\beta_1) \geq Q(v,\beta_2)$.

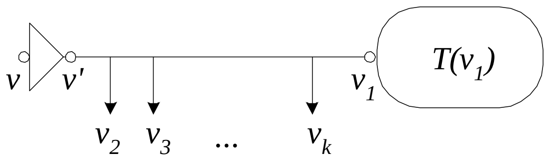


Fig. 9. If α_1 B-dominates α_2 at v_1, β_1 dominates β_2 at v.

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.

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

□

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 that:

Lemma II.5. *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 II.1.

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

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 II.4 and II.5 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 [29]. 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 re-balance 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. 7, 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 5 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. 10 is an example candidate tree where qa, ca and ra fields are all 0.

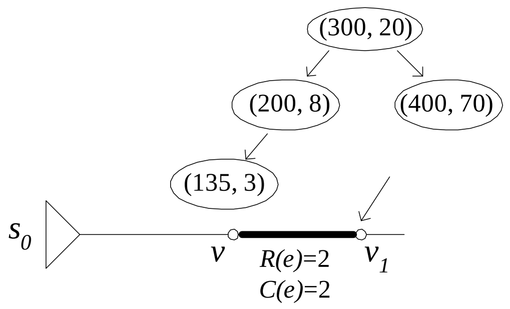


Fig. 10. Candidate tree $A(v_1)$ of four candidates. Fields qa, ca and ra are 0 for all candidates.

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:

- $ca = C(e)$, meaning that C of every candidate in the tree will be increased by $C(e)$,
- $qa = -R(e)C(e)/2$, meaning that Q of every candidate in the tree will be

decreased by $R(e)C(e)/2$, and

- $ra = R(e)$, 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

$$\begin{aligned} C(v,\alpha) &= c(u_k) + \sum_{i=1}^{k} ca(u_i), \\ Q(v,\alpha) &= q(u_k) - \sum_{i=1}^{k} ra(u_i)\left(c(u_k) + \sum_{j=i+1}^{k} ca(u_j)\right) + \sum_{i=1}^{k} qa(u_i). \end{aligned} \tag{2.1}$$

Fig. 11 shows the candidate tree after adding a wire (v, v_1) in Fig. 10.

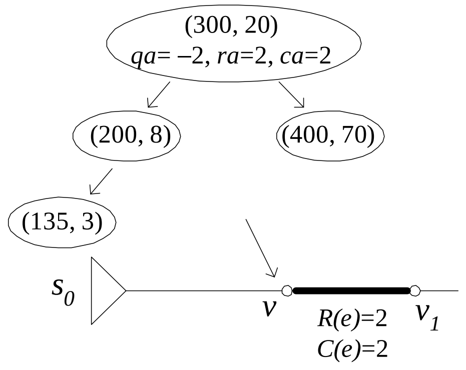


Fig. 11. Candidate tree $A(v)$ of four candidates after the wire is added.

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;//buffer type and 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 Eqn. (2.1) is evaluated.

Fig. 12 is an example showing how the candidate tree is updated when node $(200, 8)$ is visited.

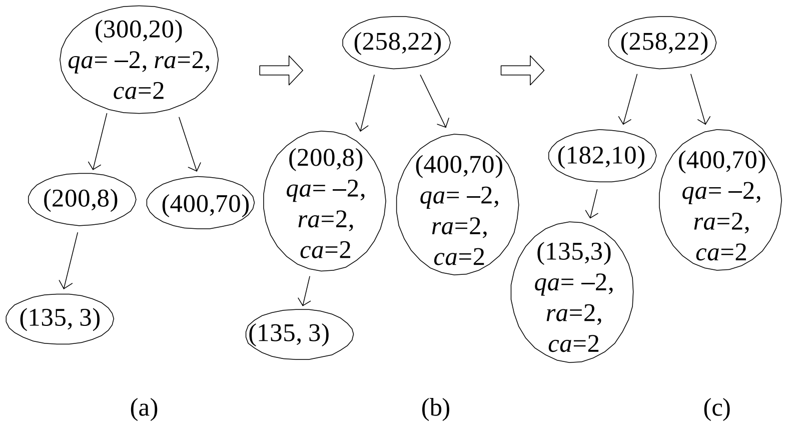

(300,20)
qa= −2, ra=2, ca=2
(200,8)
(400,70)
(135, 3)
(258,22)
(200,8) qa= −2, ra=2, ca=2
(400,70) qa= −2, ra=2, ca=2
(135, 3)
(258,22)
(182,10)
(400,70) qa= −2, ra=2, ca=2
(135,3) qa= −2, ra=2, ca=2
(a)
(b)
(c)

Fig. 12. Update of candidate tree $A(v)$ when some nodes are 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`.

```
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;
}
```

For simplicity, we illustrate a recursive version, though the implemented algorithm is non-recursive [29].

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 [10], 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

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

information in memory $O(n)$ for 2-pin nets, and $O(n \log n)$ for multi-pin 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 {//buffer type and 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 $O(n)$ 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 non-empty. Then every candidate in the sub-tree 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;
}
```

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

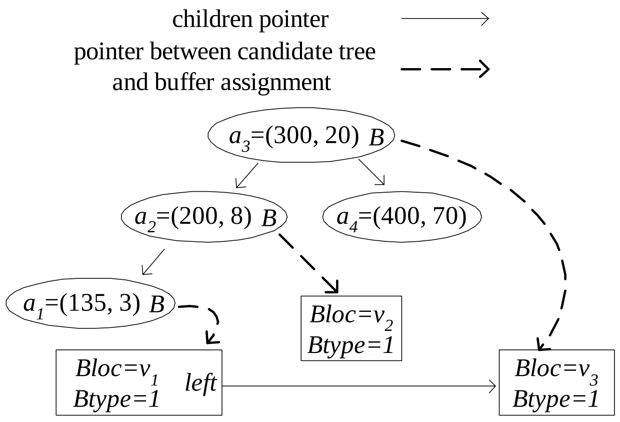


Fig. 13. Four candidates with their buffer types and locations: α_4 has no buffer, α_3 has one buffer at v_3, α_2 has one buffer at v_2, and α_1 consists of α_3 and a buffer at v_1 as shown in Fig. 10.

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

$$l_i = \frac{Q(v,\alpha_i) - Q(v,\alpha_{i-1})}{C(v,\alpha_i) - C(v,\alpha_{i-1})} - R(B). \tag{2.2}$$

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

Lemma II.6. *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 Eqn. (2.2). 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,

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

Therefore,

$$\begin{gathered} P(v,\alpha_1) - P(v,\alpha_2) = Q(v_1,\alpha_1) - Q(v_1,\alpha_2) \\ + (R(e) + R(B)) \cdot (C(v_1,\alpha_2) - C(v_1,\alpha_1)). \end{gathered}$$

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 $R(e) \geq l_2$. $\square$

$L(v)$ 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. 14 to 16 are examples showing how the candidate tree and expiration list change when a wire is added.

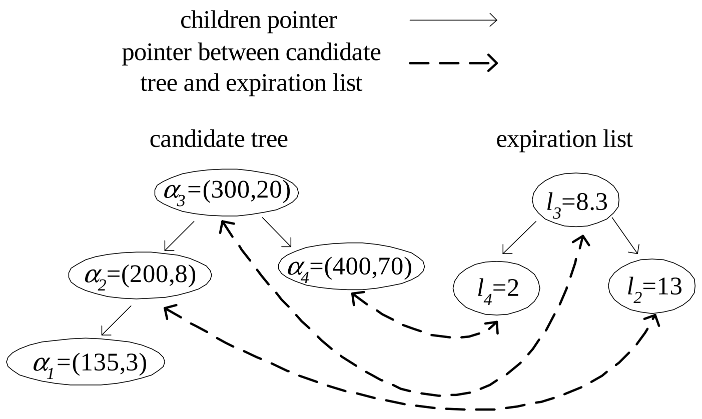


Fig. 14. The candidate tree and expiration list before adding a wire.

e. Fast Merge

The case for merge in Fig. 8 is more involved. Assume we have computed all nonredundant candidates for $T(v_1)$ and $T(v_2)$, and stored the results in $A(v_1)$, $L(v_1)$, $A(v_2)$,

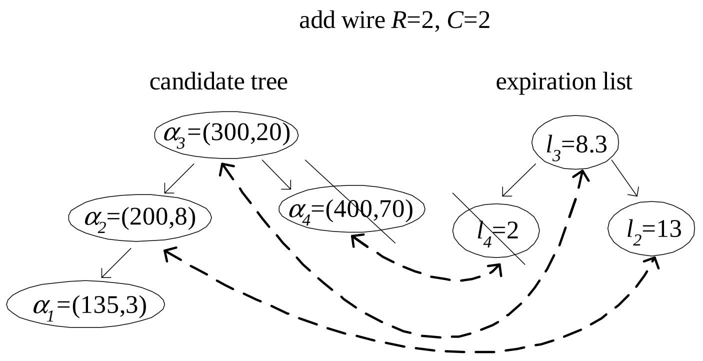


Fig. 15. After adding a wire with $R = 2$, $C = 2$, (400, 70) is redundant.

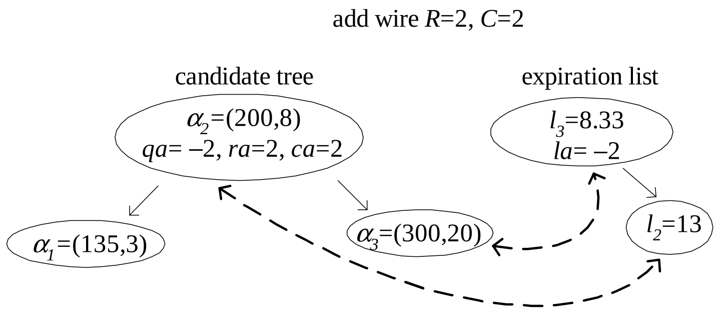


Fig. 16. Final candidate tree and expiration list.

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 $n_1 \geq n_2$.

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_j's. 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. 17 to 20 are an example of the fast merge process. For simplicity, fields qa, ra, ca of all candidates are initially 0.

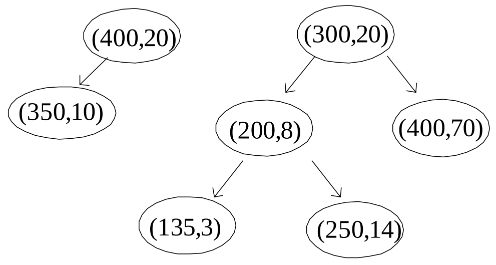


Fig. 17. Two candidate trees $A(v_1)$ (right) and $A(v_2)$.

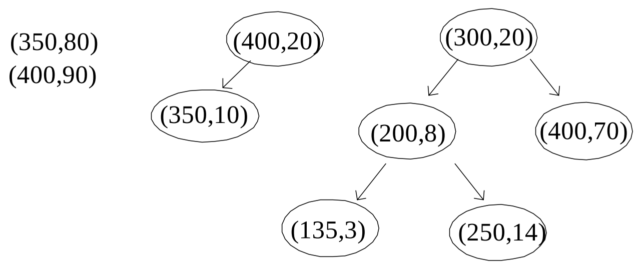


Fig. 18. List Z of candidates of $T(v)$ whose Q is decided by $T(v_2)$.

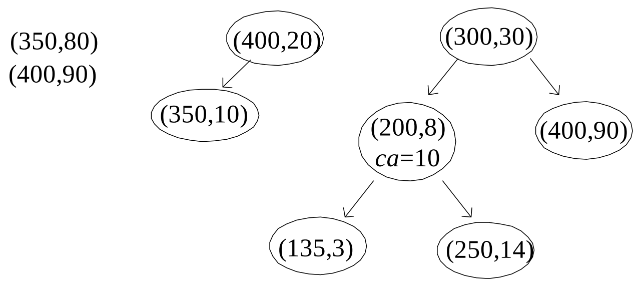


Fig. 19. Candidate tree $A(v_1)$ now stores candidates of $T(v)$ whose Qs are decided by $T(v_1)$.

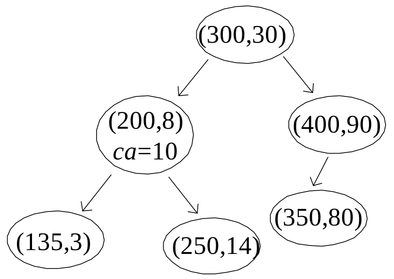


Fig. 20. Insert candidates in Z to the updated candidate tree and delete redundancy. Final candidate tree.

2. Algorithm

We will compute all nonredundant candidates $N(s_0)$ for the given tree $\boldsymbol{T}$. Our algorithm FBI (Fast Buffer Insertion) 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 $T(v)$.
if $v$ is a sink then
    Create a candidate tree $A(v)$ to store the only candidate of $T(v)$;
    return $A(v)$;
else if $T(v)$ consists of edge $(v, v_1)$ and $T(v_1)$ then
    $A(v_1) \leftarrow FBI(v_1)$;
    Modify $A(v_1)$ to include delay due to wire $(v, v_1)$;
    Delete redundancy;
    return the modified $A(v_1)$;
else if $T(v)$ consists of buffer position $v$ and $T(v_1)$ then
    $A(v_1) \leftarrow FBI(v_1)$;
    Find candidate $\alpha$ 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 $A(v_1)$;
else
    $T(v) = T(v_1) \cup T(v_2)$;
    $A(v_1) \leftarrow FBI(v_1)$; $A(v_2) \leftarrow FBI(v_2)$;
    Assume without loss of generality $|A(v_1)| \geq |A(v_2)|$;
    $Z \leftarrow$ nonredundant candidates of $T(v)$ whose $Q$ are determined by $T(v_2)$;
    Compute nonredundant candidates of $T(v)$ whose $Q$ are determined by $T(v_1)$;
    Change $A(v_1)$ to store the resulting candidates;
    Insert $Z$ into $A(v_1)$ and delete redundancy;
    return the modified $A(v_1)$;
end
```

We now explain the details.

a. Sink

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

```
x->c = C(s);
x->q = RAT(s);
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. 6, 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

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

and $C(v,\beta) = C(B)$. From Lemma II.5, 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 $C(v,\alpha_i) \le C(v,\beta) \le C(v,\alpha_{i+1})$. 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 II.8.

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 l's 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. 7, 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 β's redundant. We compare $R(e)$ with the minimum l_i in $L(v_1)$. If $R(e) \geq l_i$, according to Lemma II.6, 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 II.8.

From Eqn. (2.2), it can be seen that the addition of e decreases the value of all l_i's 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_i's 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_j's. In other words, we want to find index j:

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

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 Q's 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

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

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 $T(v)$:

$$\begin{aligned} \alpha_i \cup \beta_j &: \quad Q = Q(v_1,\beta_j), C = C(v_1,\beta_j) + C(v_2,\alpha_i), \\ & \quad \ldots \\ \alpha_i \cup \beta_l &: \quad Q = Q(v_1,\beta_l), C = C(v_1,\beta_l) + C(v_2,\alpha_i). \end{aligned}$$

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. 21 illustrates the general situation of nodes $u(\beta_j)$, $u(\beta_{j+1})$, …, $u(\beta_l)$. 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 $Q(v_1, \gamma) \ge Q(v_1, \beta_j)$. In Fig. 21, 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
```

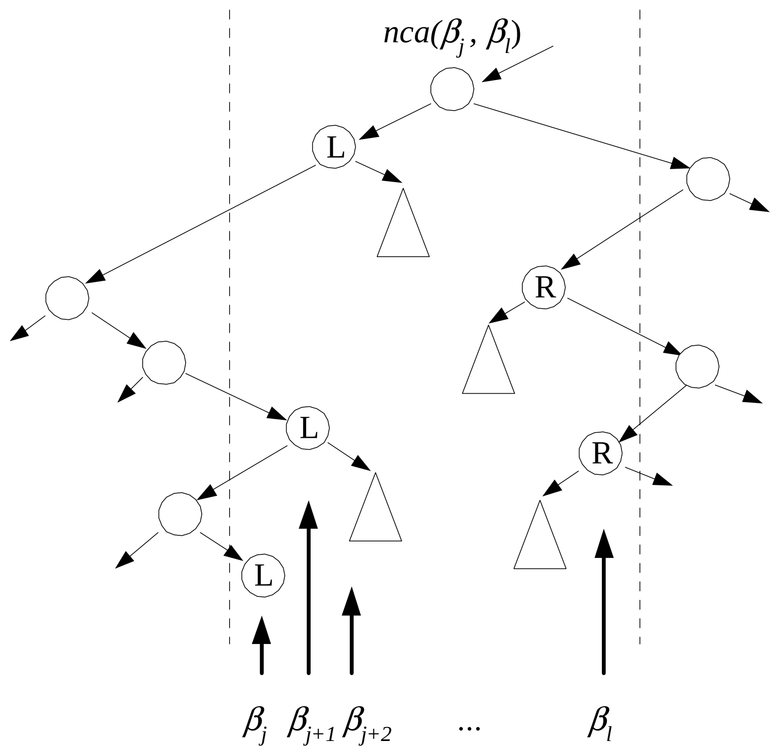


Fig. 21. Nodes $u(\beta_j), u(\beta_{j+1}), \ldots, u(\beta_l)$ in candidate tree $A(v_1)$ form an interval.

```
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. 21, 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 $nca(\beta_j, \beta_l)$. 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_j's and β_l's is $O(n_2 \log n_1)$. It is easy to see all the nca's 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 $O(n_2 \log n_1)$. 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 l's 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.

3. Analysis

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

Lemma II.7. *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 sub-tree $T(v_1)$ 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 sub-trees $T(v_1)$ 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 II.8. *Algorithm FBI correctly finds all nonredundant candidates in worst case time $O(n \log n)$ for two-pin nets, and $O(n \log^2 n)$ for multi-pin 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 II.4, 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 pre-process 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, 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 two-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 multi-pin 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 II.7, there are at most $n+1$ nonredundant candidates. Therefore, we have the following recurrence relation:

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

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 $n > n_1 \geq n_2 > 0$. We prove by induction that

$$\mathcal{T}(n) \leq cn \log^2 n. \tag{2.3}$$

Obviously $\mathcal{T}(0) = 0 \leq c \cdot 1 \log^2 1$. Assume Eqn. (2.3) is true for all $k < n$, then

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

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 II.7 there are at most n insertions, so there are at most n deletions.

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. 21 is $O(n_2 \log(n_1/n_2+1))$ as shown in Brown and Tarjan [30].

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

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. $\square$

4. 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 $P_i(v, \alpha_1) \geq P_i(v, \alpha_2)$ and $C(v, \alpha_1) \leq C(v, \alpha_2)$. For each buffer type B_i, there will be one candidates tree $A_i(v)$ to store candidates of $T(v)$ that 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 of adding buffer we have to perform

b^2 times as much work as the single buffer case since we have b candidate trees, and the number of nonredundant candidates is $O(bn)$. Therefore for 2-pin nets, the time complexity is $O(b^2 n \log(bn)) = O(b^2 n \log n)$. For multi-pin nets, the time complexity is $O(b^2 n \log^2(bn)) = O(b^2 n \log^2 n)$.

5. 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 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 Ω, $C(B)$=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 fF 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 two-pin nets with 20 mm long and one buffer type (16X), the new algorithm is 9 to 87 times faster than van Ginneken's algorithm and uses 1/22 to 1/250 of memory. Table II shows for two-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 to 80 times faster than van Ginneken's algorithm and uses 1/4 to 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.

Table I. Simulation results for a 20 mm two-pin net with one buffer type, where n is the number of buffer positions.

n	**Time (sec)**		**Speed-**	**Mem (MB)**		**Reduc-**
	VG [10]	**New**	**up**	**VG [10]**	**New**	**tion**
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 two-pin net with five buffer types, where n is the number of buffer positions.

n	**Time (sec)**		**Speed-**	**Mem (MB)**		**Reduc-**
	$\boldsymbol{O(b^2n^2)}$ [1]	**New**	**up**	$\boldsymbol{O(b^2n^2)}$ [1]	**New**	**tion**
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, where m is the number of sinks and n is the number of buffer positions.

m	n	**Time (sec)**		**Speed-**	**Mem (MB)**		**Reduc-**
		VG [10]	**New**	**up**	**VG [10]**	**New**	**tion**
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

Table IV. Simulation results for industrial test cases with five buffer types, where m is the number of sinks and n is the number of buffer positions.

m	n	**Time (sec)**		**Speed-**	**Mem (MB)**		**Reduc-**
		$\boldsymbol{O(b^2n^2)}$ [1]	**New**	**up**	$\boldsymbol{O(b^2n^2)}$ [1]	**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

D. An $O(bn^2)$ Algorithm for Optimal Buffer Insertion with b Buffer Types

Modern design libraries may contain hundreds of different buffers with different input capacitances, driving resistances, intrinsic delays, power levels, etc. If every buffer available for the given technology is allowed, it is stated in [13] that the current buffer insertion algorithms could possibly take days or even weeks for large designs since all these algorithms are quadratic in terms of b. Alpert *et al* [13] studied how to reduce the size of the buffer library with a clustering algorithm. Though the buffer library size is reduced, the solution quality is degraded accordingly.

In Section C, we have proposed an $O(b^2nlog^2n)$ algorithm, though it is efficient for nets with large number of buffer positions, it is still not fast enough from large buffer libraries since the algorithm is still quadratic in terms of b as observed from simulation results.

In this Section, we propose a new algorithm that performs optimal buffer insertion with b buffer types in $O(bn^2)$ time. Our speedup is achieved by the observation that the candidates that generate new buffered candidates must lie on the convex hull of (Q, C).

1. New Algorithm

The previous best algorithm for multiple buffer types by Lillis, Cheng and Lin consists of three major operations: 1) adding buffers at a buffer position in $O(b^2n)$ time, 2) adding a wire in $O(bn)$ time, and 3) merging two branches in $O(bn_1+bn_2)$ time, where n_1 and n_2 are the numbers of buffer positions in the two branches. As a result, their algorithm has time complexity $O(b^2n^2)$. Note that the bottleneck of their algorithm is adding buffers. Their algorithm takes $O(b^2n)$ time to generate all new candidates and $O(b^2n)$ time to insert nonredundant ones into the original list of nonredundant candidates.

In this part, we show that the time complexity of the first operation, adding buffers at a buffer position, can be reduced to $O(bn)$, and thus our algorithm can achieve total time complexity $O(bn^2)$.

Assume we have computed the set of nonredundant candidates $N(v_1)$ for $T(v_1)$, and now reach a buffer position v, see Fig. 6. Wire (v, v_1) has 0 resistance and capacitance. Define $P_i(\alpha)$ as the slack if we add a buffer type B_i at v for any candidate α in $N(v_1)$:

$$P_i(\alpha) = Q(v_1, \alpha) - R(B_i) \cdot C(v_1, \alpha) - K(B_i). \tag{2.4}$$

If we do not insert any buffer at v, then every candidate for $T(v_1)$ is a candidate for $T(v)$. If we insert a buffer at v, then for every buffer type B_i, $i = 1, 2, \dots, b$, there will be a new candidate β_i:

$$\begin{aligned} Q(v, \beta_i) &= \max_{\alpha \in N(v_1)} \{P_i(\alpha)\}, \\ C(v, \beta_i) &= C(B_i). \end{aligned}$$

Note that some of the new candidates β_is could be redundant. Define the *best candidate* for B_i as the candidate $\alpha_i \in N(v_1)$ such that α_i maximizes $P_i(\alpha)$ among all candidates of $N(v_1)$. If there are multiple α's that maximize $P_i(\alpha)$, the one with minimum $C(\alpha)$ is chosen.

We show how to generate all β_is in $O(bn)$ time. Since all candidates discussed in this Section are in $N(v_1)$, we will write $Q(\alpha)$ for $Q(v_1, \alpha)$, and $C(\alpha)$ for $C(v_1, \alpha)$. Suppose buffers in the buffer library are sorted according to its driving resistance $R(B_i)$ in non-increasing order, $R(B_1) \geq R(B_2) \geq \cdots \geq R(B_b)$. If some buffer types are not allowed at v, we simply omit them without affecting the rest of the algorithm.

Lemma II.9. *For any two buffer types B_i and B_j, where $i > j$, let their best candidates be α_i and α_j, respectively. Then we must have $C(\alpha_i) \geq C(\alpha_j)$.*

Proof. From the definition of α_i, we have $P_i(\alpha_i) \geq P_i(\alpha_j)$ and $P_j(\alpha_j) \geq P_j(\alpha_i)$. Consequently,

$$\begin{aligned} Q(\alpha_i) - Q(\alpha_j) &\geq R(B_i) \cdot (C(\alpha_i) - C(\alpha_j)), \\ Q(\alpha_j) - Q(\alpha_i) &\geq R(B_j) \cdot (C(\alpha_j) - C(\alpha_i)). \end{aligned}$$

Therefore, $(R(B_i) - R(B_j))(C(\alpha_i) - C(\alpha_j)) \leq 0$.

Since $i > j$, $R(B_j) \geq R(B_i)$. If $R(B_j) > R(B_i)$, $C(\alpha_i) \geq C(\alpha_j)$. If $R(B_j) = R(B_i)$, then it is easy to get $P_i(\alpha_i) = P_i(\alpha_j)$ and $P_j(\alpha_j) = P_j(\alpha_i)$. From the definition, when there are multiple α's that maximize $P_i(\alpha)$, the one with minimum $C(\alpha)$ is chosen. Thus α_i and α_j should be the same candidate, which means $C(\alpha_i) = C(\alpha_j)$. □

Lemma II.9 implies that the best candidates $\alpha_1, \ldots, \alpha_b$ for buffer types $B_1, \ldots, B_b$ are in increasing order of C. However, this is not enough for an $O(bn^2)$ time algorithm. In the following, we define the concept of *convex pruning*, which can be used to prune useless candidates that are not pruned by the traditional van Ginneken's algorithm.

Convex pruning: Let α_1, α_2 and α_3 be three nonredundant candidates of $T(v_1)$ such that $C(\alpha_1) < C(\alpha_2) < C(\alpha_3)$. If

$$\frac{Q(\alpha_2) - Q(\alpha_1)}{C(\alpha_2) - C(\alpha_1)} < \frac{Q(\alpha_3) - Q(\alpha_2)}{C(\alpha_3) - C(\alpha_2)}, \tag{2.5}$$

then we prune candidate α_2.

Convex pruning can be explained by Fig. 22. Consider Q as the Y-axis and C as the X-axis. Then the set of nonredundant candidate $N(v_1)$ are a set of points in the two-dimensional plane. Candidate α_2 in the above definition is shown in Fig. 22(a), and is pruned in Fig. 22(b). Call the candidates after convex pruning $M(v_1)$. It can be seen that $N(v_1)$ is a monotonically increasing sequence, while $M(v_1)$ is a convex hull.

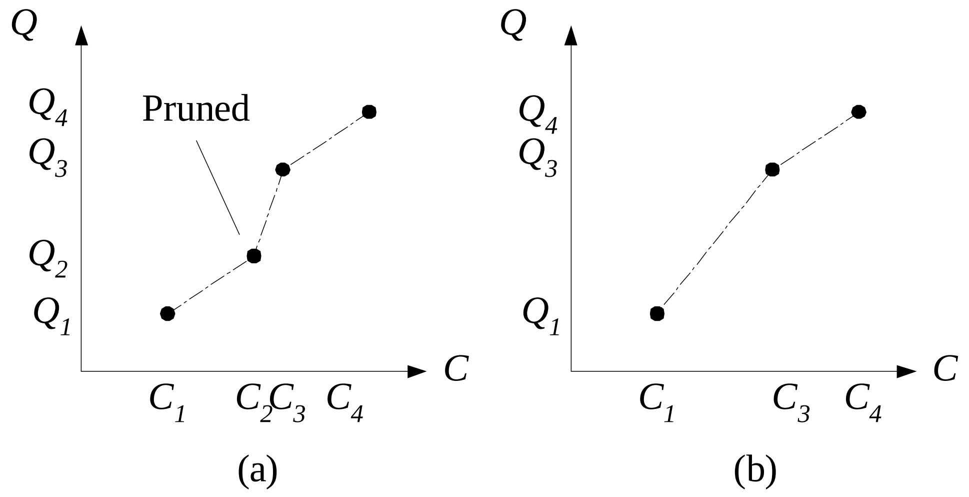


Fig. 22. (a) Nonredundant candidates $N(v_1)$ on (Q, C) plane. (b) Nonredundant candidates $M(v_1)$ after convex pruning.

Function `ConvexPruning` performs convex pruning for any list of nonredundant candidates sorted in increasing Q and C order. The following C code defines the linked list data structure for the candidates:

```
typedef struct Candidate {
    double Q, C;
    struct Candidate *next, *prev;  // double link list
} Candidate;
```

Let the candidate with minimum C be α_1. We add a dummy candidate $(-\infty, C(\alpha_1))$ at the beginning of the list to simplify the algorithm. The list is pointed by `header`. Function `LeftTurn` checks if `a1`, `a2` and `a3` form a left turn on the plane. It is the same as the condition in Eqn. (2.5).

Lemma II.10. *Given any set of k nonredundant candidates sorted in increasing Q*

```
void ConvexPruning(Candidate *header) {
    Candidate *a1, *a2, *a3;

    a1 = header;
    a2 = a1->next;
    a3 = a2->next;

    while (a3 != NULL) {
        if (LeftTurn(a1, a2, a3)) {
            // prune a2 and move backward
            free(a2);
            a1->next = a3;
            a3->prev = a1;
            a2 = a1;
            a1 = a1->prev;
        } else {
            // move forward
            a3 = a3->next;
            a2 = a2->next;
            a1 = a1->next;
        }
    }
}
```

and C order, function `ConvexPruning` *performs convex pruning in $O(k)$ time.*

Proof. This procedure is known as Graham's scan in computational geometry [31]. It finds the convex hull of a set of points in sorted order in linear time.

A simple proof is shown here. It is well known that a set of points form a convex hull if and only if there are no consecutive α_1, α_2 and α_3 that satisfy Eqn. (2.5). Therefore, `ConvexPruning` is correct since it checks all consecutive candidates.

To analyze the time complexity, consider the number of forward and backward moves. Each time `ConvexPruning` moves backward, it deletes a candidate. Therefore, there can be at most k backward moves. The number of forward moves is the size of the list plus the number of backward moves. Therefore the number of forward moves is at most $2k$. Hence the time complexity is $O(k)$. □

Lemma II.11. *For any buffer type $B_i \in \boldsymbol{B}$, its best candidate α_i that maximizes $P_i(\alpha)$ is not pruned by* `ConvexPruning`*.*

Proof. Consider any candidate $\gamma \in N(v_1)$ with $C(\gamma) > C(\alpha_i)$. According to the definition of α_i, we have $P_i(\alpha_i) \geq P_i(\gamma)$. Therefore,

$$\begin{aligned} Q(\gamma) - Q(\alpha_i) &\leq R(B_i) \cdot (C(\gamma) - C(\alpha_i)), \\ \frac{Q(\gamma) - Q(\alpha_i)}{C(\gamma) - C(\alpha_i)} &\leq R(B_i). \end{aligned}$$

Similarly for any candidate $\eta \in N(v_1)$ with $C(\eta) < C(\alpha_i)$, we have

$$\begin{aligned} Q(\alpha_i) - Q(\eta) &\geq R(B_i) \cdot (C(\alpha_i) - C(\eta)), \\ \frac{Q(\alpha_i) - Q(\eta)}{C(\alpha_i) - C(\eta)} &\geq R(B_i). \end{aligned}$$

Therefore,

$$\frac{Q(\alpha_i) - Q(\eta)}{C(\alpha_i) - C(\eta)} \geq \frac{Q(\gamma) - Q(\alpha_i)}{C(\gamma) - C(\alpha_i)},$$

where η is any candidates with $C(\eta) < C(\alpha_i)$, and γ is any candidates with $C(\gamma) > C(\alpha_i)$. According to the definition of convex pruning, α_i is not pruned. □

Lemma II.12. *Let the set of nonredundant candidates after* `ConvexPruning` *be $M(v_1)$ and assume $M(v_1)$ are sorted in increasing Q and C order. Consider any three candidates η, α, γ in $M(v_1)$, such that $C(\eta) < C(\alpha) < C(\gamma)$. For any buffer type $B_i \in \boldsymbol{B}$, if $P_i(\eta) \geq P_i(\alpha)$, then $P_i(\eta) \geq P_i(\gamma)$; if $P_i(\gamma) \geq P_i(\alpha)$, then $P_i(\gamma) \geq P_i(\eta)$.*

Proof. From the definition of convex pruning, we have

$$\frac{Q(\gamma) - Q(\alpha)}{C(\gamma) - C(\alpha)} \leq \frac{Q(\alpha) - Q(\eta)}{C(\alpha) - C(\eta)}.$$

If $P_i(\eta) \geq P_i(\alpha)$, then

$$\begin{aligned}
\frac{Q(\alpha)-Q(\eta)}{C(\alpha)-C(\eta)} &\leq R(B_i),\\
\frac{Q(\gamma)-Q(\alpha)}{C(\gamma)-C(\alpha)} &\leq R(B_i),\\
Q(\alpha)-R(B_i)\cdot C(\alpha) &\geq Q(\gamma)-R(B_i)\cdot C(\gamma),\\
P_i(\alpha) &\geq P_i(\gamma).
\end{aligned}$$

Therefore, $P_i(\eta) \geq P_i(\gamma)$. Similarly, if $P_i(\gamma) \geq P_i(\alpha)$, then

$$\begin{aligned}
\frac{Q(\gamma)-Q(\alpha)}{C(\gamma)-C(\alpha)} &\geq R(B_i),\\
\frac{Q(\alpha)-Q(\eta)}{C(\alpha)-C(\eta)} &\geq R(B_i),\\
Q(\alpha)-R(B_i)\cdot C(\alpha) &\geq Q(\eta)-R(B_i)\cdot C(\eta)\\
P_i(\alpha) &\geq P_i(\eta).
\end{aligned}$$

Therefore, $P_i(\gamma) \geq P_i(\eta)$. $\square$

Lemma II.12 implies that for any buffer type B_i, if candidate α maximizes $P_i(\alpha)$ among its previous and next consecutive candidates in $M(v_1)$, then α maximizes $P_i(\alpha)$ among all candidates in $M(v_1)$.

Function `NewCandidate` identifies the best candidates α_i from $N(v_1)$ and generates new candidates β_i, for $i = 1, \ldots, b$. Nonredundant candidates in $N(v_1)$ are stored in increasing C order using a double link list pointed by `header`. Buffer types are sorted in non-increasing driver resistance order and stored in array `B`. Function `P(i, a)` computes $P_i(\alpha)$ as defined in Eqn. (2.4). Function `Sort(beta)` sorts β's in nondecreasing C order.

Theorem II.13. *If v is a buffer position, wire (v, v_1) is a wire with zero resistance and capacitance, nonredundant candidates of $N(v_1)$ are stored in increasing Q and C*

```
void Candidate *NewCandidate(Candidate *header,
   Candidate *beta) {
     Candidate *a1, *a2;
     int i;

     ConvexPruning(header);

     a1 = header;
     a2 = a1->next;
     for (i = 1; i <= b; i ++) {
          while (a2 != NULL) {
               if (P(i, a1) < P(i, a2)) {
                    a1 = a1->next;
                    a2 = a1->next;
               } else
                    break;
          }
          // generate new candidate
          beta[i]->Q = P(i, a1);
          beta[i]->C = B[i]->C;
     }
     Sort(beta);
}
```

order, then function `NewCandidate` *generates all new candidates for $N(v)$ in $O(bn)$ time.*

Proof. Let the set of nonredundant candidates after `ConvexPruning` be $M(v_1)$. From Lemma II.11, we know that all best candidates α_i's are in $M(v_1)$. From Lemma II.9 and Lemma II.12, starting from the first candidates in $M(v_1)$, function `NewCandidate` can find all β_i's in the increasing order of i.

Now consider the time complexity. According to Lemma II.10, function `ConvexPruning` takes $O(bn)$ time. The `for` loop takes $O(bn + b) = O(bn)$ time. To reduce the time complexity for function `Sort`, we sort the entire buffer library according to input capacitance $C(B_i)$ in $O(b \log b)$ time during pre-processing, and establish an order from buffer index i to the order in $C(B_i)$. Then each time function `Sort` is called, the new candidates β_i's can be sorted in nondecreasing C order by using the index in $O(b)$ time. □

Once we have all new candidates generated and sorted in increasing Q and C order, it is easy to merge with nonredundant candidates in $N(v_1)$ to produce $N(v)$. The time it takes is linear in terms of the two lists: $O(bn) + O(b) = O(bn)$. Since the other two operations, adding a wire and merging, can both be done in time $O(bn)$, we have:

Theorem II.14. *The optimal buffer insertion problem for b buffer types and n possible buffer positions can be computed in time $O(bn^2)$.*

2. Simulation

Both the algorithm of Lillis, Cheng and Lin [1] and the new algorithm are implemented in C and run on a Sun SPARC workstations with 400 MHz and 2 GB memory. The device and interconnect parameters are based on TSMC 180 nm technology. We have 4 different buffer libraries, of size 8, 16, 32 and 64 respectively. The value of $R(B_i)$ is from 180 Ω to 11520 Ω, $C(B_i)$ is from 0.36 fF to 23.4 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 V shows for large industrial circuits, the new algorithm is up to 11 times faster than Lillis' algorithm. The memory usage is only 2% more due to the double linked list used by the new algorithm.

Fig. 23 compares the time complexity of two algorithms for the net with 1944 sinks and 33133 buffer positions with respect to the size of buffer library b. In the

Table V. Simulation results for industrial test cases, where m is the number of sinks (pins), n is the number of buffer positions, and b is the library size.

m	n	b	**CPU Time (sec)**		**Speedup**
			New $O(bn^2)$	Lillis-Cheng-Li [1] $O(b^2n^2)$	
337	336	8	0.08	0.09	1.11
		16	0.14	0.16	1.14
		32	0.23	0.36	1.57
		64	0.42	0.91	2.17
	5647	8	1.54	2.15	1.40
		16	2.11	4.55	2.16
		32	2.81	9.99	3.56
		64	4.05	22.52	5.56
	10957	8	4.56	7.15	1.57
		16	6.02	15.74	2.61
		32	7.62	34.02	4.46
		64	9.98	74.55	7.47
1944	1943	8	0.93	0.90	0.97
		16	1.62	1.86	1.15
		32	2.78	4.38	1.58
		64	4.54	10.71	2.36
	33133	8	22.96	38.19	1.66
		16	31.97	90.08	2.82
		32	40.83	209.82	5.14
		64	50.42	457.22	9.07
	64323	8	70.23	141.07	2.01
		16	95.78	337.97	3.53
		32	117.38	755.46	6.44
		64	136.85	1596.61	11.67
2676	2675	8	1.16	1.13	0.97
		16	2.07	2.38	1.15
		32	3.83	5.78	1.51
		64	6.18	14.15	2.30
	45075	8	27.31	44.29	1.62
		16	36.75	98.31	2.68
		32	47.8	226.25	4.73
		64	64.02	543.45	8.49
	87475	8	82.67	163.87	1.98
		16	108.16	372.22	3.44
		32	134.83	835.04	6.19
		64	164.08	1864.08	11.36

figure, the y axis is normalized to the running time of the case when the buffer library size is 8. Though the worst case time complexity of Lillis' algorithm is quadratic in terms of b, it behaves more like a linear function of b, as observed in [13]. The time complexity of our algorithm is also linear, but has a much smaller slope.

Fig. 24 compares the time complexity of the two algorithms for the net with 1944 sinks, with respect to the number of buffer positions n. The buffer library size is 32. In the figure, the y axis is normalized to the running time of the case with 1943 buffer positions. We can see that while Lillis' and our algorithms both behave quadratically, our algorithm shows much slower growing trend since the operation of adding buffers becomes more dominant among three major operations when n increases.

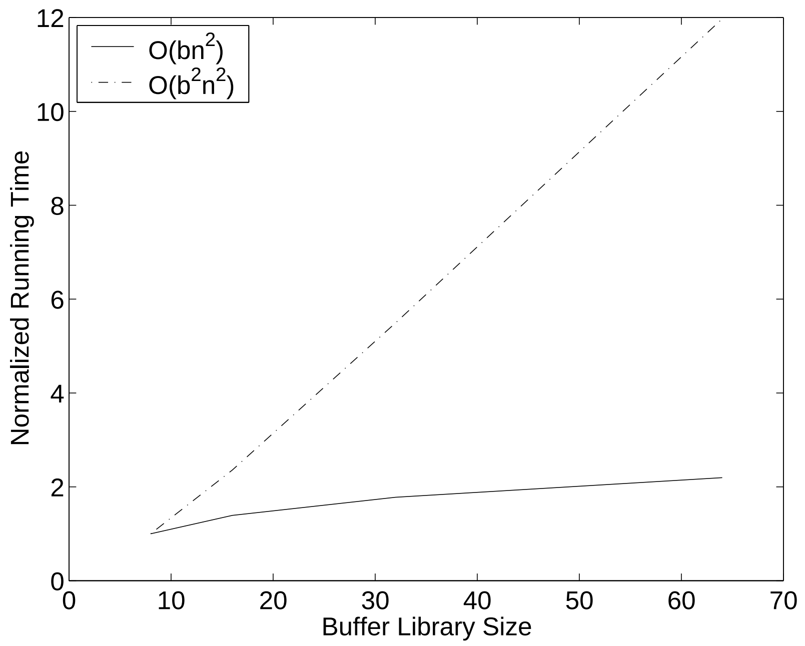


Fig. 23. Comparison of normalized running time of our new $O(bn^2)$ time algorithm and the $O(b^2n^2)$ time algorithm [1]. Number of sinks is 1944 and number of buffer positions is 33133.

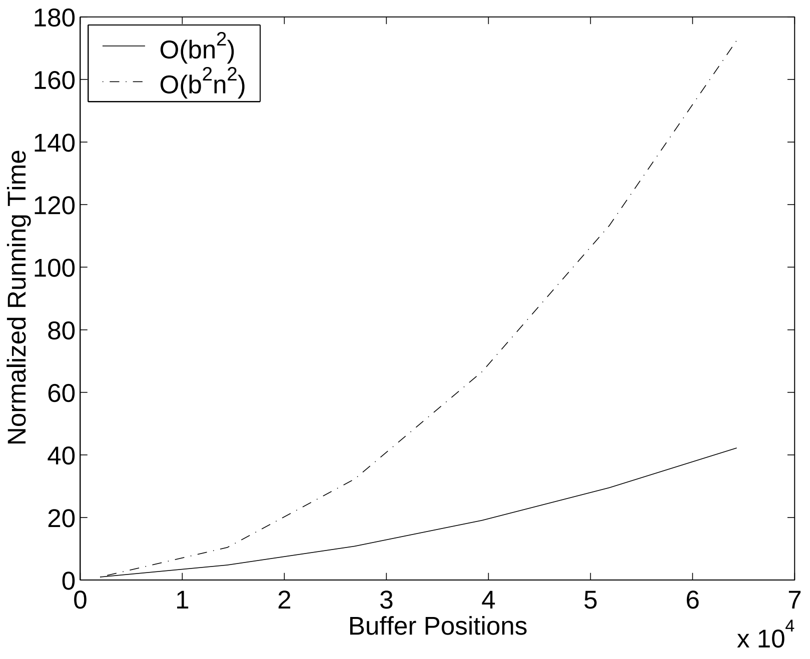


Fig. 24. Comparison of normalized running time of our new $O(bn^2)$ time algorithm and the $O(b^2n^2)$ time algorithm [1]. Number of sinks is 1944 and number of buffer types is 32.

3. Extension

The algorithm described in Section 1 can be extended to improve the buffer cost minimization algorithm by Lillis, Cheng and Lin [1]. They represent each candidate as a tuple (Q, C, W), where W is the total buffer cost, and perform three operations during dynamic programming: 1) adding buffers at a buffer position, 2) adding a wire, and 3) merging two branches. In their algorithm, candidates are first grouped according to W, and then for each value of W, stored in increasing order of (Q, C). According the analysis in [1], the operation of adding buffers takes $O(bN)$ time to generate new candidates, where N is the number of nonredundant candidates.

We extend our algorithm to (Q, C, W) framework as follows. For each W, we apply function `NewCandidate` on its list of (Q, C) candidates. With a similar analysis as in Section 1, it is clear that the time to generate new candidates is reduced to $O(N)$. The time for other two operations is the same.

Our new algorithm can also be easily integrated with predictive pruning [32, 2], and inverting buffer types [1].

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

All previous buffer insertion algorithms do not utilize the fact that in real applications most nets have small numbers of sinks and large number of buffer positions. As a result, the running time of these algorithms is still not very fast, especially when other constraints such as slew and cost are considered.

In this Section, 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 simple implicit linked list, is much simpler than the candidate tree used in [33] and the skip list used in [34]. 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. 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 $m \leq n$.

1. Two-Pin Nets

In this part, we show how to compute optimal buffer insertion for 2-pin nets in $O(b^2n)$ 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 [33], but much simpler.

a. Convex Pruning

The concept of convex pruning was first proposed by Li and Shi [35] and has been explained in Section D:

Definition II.15. *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*

$$\frac{Q(\alpha_2) - Q(\alpha_1)}{C(\alpha_2) - C(\alpha_1)} < \frac{Q(\alpha_3) - Q(\alpha_2)}{C(\alpha_3) - C(\alpha_2)}, \tag{2.6}$$

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

Convex pruning can be explained by Fig. 22. 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 Fig. 22(a), and is pruned in Fig. 22(b). The set of nonredundant candidates after convex pruning $M(v)$ is a convex hull.

Lemma II.16. *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 the definition. In a 2-pin net, every candidate will be connected to some wires, could be empty, before reaches an upstream buffer or source. Let v' be the upstream buffer or source, D be the sum of the delay of wires from v' to v and the delay of buffer or source 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 buffer or source 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

$$\begin{aligned} Q(v', \alpha_2) &< Q(v, \alpha_3) - R \cdot C(v, \alpha_3) - D \\ &= Q(v', \alpha_3). \end{aligned}$$

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 (2.6) implies

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

Therefore

$$\begin{aligned} Q(v', \alpha_2) &< Q(v, \alpha_1) - R \cdot C(v, \alpha_1) - D \\ &= Q(v', \alpha_1). \end{aligned}$$

This shows α_2 always gives worse slack than 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 be reset $C(\alpha_i)$. Therefore α_2 is redundant. $\square$

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 [35]. 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 we have computed the set of nonredundant candidates $N(v)$ for $T(v)$, and now reach a buffer position v', see Fig. 6. Wire (v', v) has 0 resistance and capacitance. Define $P_i(\alpha)$ as the slack if we add a buffer of type B_i at v' for any candidate α:

$$P_i(\alpha) = Q(v, \alpha) - R(B_i) \cdot C(v, \alpha) - K(B_i). \tag{2.7}$$

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

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

Define the *best candidate* for B_i as the candidate $\alpha \in N(v)$ such that α maximizes $P_i(\alpha)$ among all candidates in $N(v)$. If there are multiple α's that maximize $P_i(\alpha)$, choose the one with minimum C.

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 II.17. *For any $T(v)$, let nonredundant candidates after convex pruning be $\alpha_1, \alpha_2, \dots, \alpha_k$, 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 $k \le j$.*

Proof. From the definition,

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

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

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

which is equivalent to

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

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

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

This implies $k \le j$. □

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

Lemma II.18. *For any $T(v)$, let nonredundant candidates after convex pruning be $\alpha_1, \alpha_2, \dots, \alpha_k$, in increasing Q and C order. If $P_i(\alpha_{j-1}) \le P_i(\alpha_j)$, $P_i(\alpha_j) \ge P_i(\alpha_{j+1})$,* *then α_j is the best candidate for buffer type B_i and*

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

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, (2.6) 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, (2.6) 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. $\square$

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`:

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

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 [33], 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 (2.7). From Lemma II.17, the best candidate is always to the left of where we found the best candidate last time. From Lemma II.18, we can confirm 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;
    ...
```

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 $L(v)$. 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 $v_1, \dots, v_{n+1}$ where $v_{n+1}$ is the
         sink.
  Output: Nonredundant candidates of $T(v_1)$ stored in linked list $L$.
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     foreach buffer type $B_j$ allowed at $v_i$ do
7         AddBuffer(j);
8     end
9 end
```

Theorem II.19. *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 II.16 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 $O(b^2n)$. $\square$

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.

2. 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. For any 2-pin segment $v_1, v_2, \ldots, v_k$, the subroutine takes as input $A(v_k)$, prunes non-convex ones to get $L(v_k)$, and computes each $L(v_i)$ and $A(v_i)$ as it moves to v_1.

Theorem II.20. *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

```
Algorithm M-Pin
    **Input**: Routing tree $T(v)$ with root $v$.
    **Output**: List $A(v)$ that contains all nonredundate candidates of $T(v)$.
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
5     $T(v)$ consists of subtrees $T(v_1)$ and $T(v_2)$ ;
6     Recursively compute $A(v_1)$ and $A(v_2)$;
7     Merge $A(v_1)$ and $A(v_2)$ to form $A(v)$;
8 end
9 return $A(v)$;
```

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 [2, 33], and inverting buffer types [1].

3. Buffer Cost Minimization for 2-Pin Net

Now we consider the min-cost buffer insertion problem. Again we start with 2-pin nets. Let integer ω be the maximum possible cost of any candidate, while the minimum non-zero cost is scaled to 1. Lillis, Cheng and Lin's algorithm performs the following operations for 2-pin nets: At each buffer position, insert $b \cdot \omega$ new candidates. Since there are n buffer positions, the total number of nonredundant candidates is $O(bn\omega)$. Therefore, the time complexity of their algorithm is $O(b^2n^2\omega)$. In this section, we reduce the time complexity to $O(b^2n\omega)$.

We use the same (Q, C, W) paradigm, where W is the total buffer cost. For each $T(v)$, candidates are stored in ω lists $L_1, L_2, \ldots, L_\omega$. List L_i contains candidates with cost i. In each list, candidates are stored as (Q, C) pairs using implicit representation described above, and pruned through convex pruning. The same global variables are used: `Qa`, `Ca` and `Ra`.

When we reach each wire, we perform the same operation as before in $O(1)$ time. When we reach a buffer position, we perform the same operation for each list L_i: Form b new candidates with each buffer B_j and insert the new candidates into list $L_{i+W(B_j)}$. We do not perform pruning across different lists. This gives the total time as claimed.

4. Simulation

All algorithms, $O(b^2n^2)$ [1], $O(b^2nlog^2n)$ in Section C, $O(bn^2)$ in Section D and the $O(mn)$ algorithm introduced in this Section, 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. 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 2880 Ω, $C(B_i)$ is from 1.46 fF to 23.4 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 VI 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 VII 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.

The second set of experiments are for Min-Cost Buffer Insertion Problem. We test our new algorithm on nets extracted from an industrial ASIC chip with 300k+ gates [36]. The gates have been placed and buffers are required to optimize timing. This group consists 429 two pin nets among 1000 most time consuming nets from one ASIC chip. Each net has tens to few hundreds of buffer positions with different metal layers and vias. The buffer library consists of 24 buffers, in which 8 are non-inverting buffers and 16 are inverting buffers. The range of driving resistance is from 120 Ω to 945 Ω, and the input capacitance is from 6.27 fF to 121.56 fF. In this case, our new algorithm is 10% faster than previous best optimal algorithm [2].

Table VI. Simulation results for a 2mm two-pin net, where n is the number of buffer positions, and b is the library size.

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

Table VII. Simulation results for industrial test cases, where m is the number of sinks, n is the number of buffer positions, and b is the library size.

m	n	b	CPU Time (sec)			
			New $O(b^2n + bmn)$	Section D $O(bn^2)$	Section C $O(b^2n\log^2 n)$	Lillis-Cheng-Lin [1] $O(b^2n^2)$
25	107	1	0.002	0.002	0.002	0.01
		4	0.01	0.01	0.03	0.01
		8	0.01	0.01	0.16	0.02
		16	0.03	0.02	0.67	0.05
	1337	1	0.02	0.14	0.04	0.24
		4	0.11	0.44	0.48	1.06
		8	0.20	0.60	2.06	1.95
		16	0.33	0.78	8.62	3.32
	2567	1	0.05	0.50	0.08	0.75
		4	0.19	1.47	1.04	4.08
		8	0.36	2.07	4.30	7.07
		16	0.64	2.58	17.94	12.12
	12407	1	0.26	11.14	0.50	21.43
		4	1.00	32.73	6.22	100.31
		8	1.76	46.05	25.54	200.61
		16	3.26	56.33	104.87	334.92
337	336	1	0.03	0.02	0.02	0.02
		4	0.06	0.04	0.04	0.05
		8	0.12	0.08	0.75	0.09
		16	0.20	0.14	3.23	0.19
	5647	1	0.22	0.41	0.17	0.89
		4	0.59	0.98	2.03	2.51
		8	0.98	1.51	8.34	4.46
		16	1.73	2.03	31.55	7.34
	10957	1	0.42	1.24	0.34	3.40
		4	1.16	2.95	4.10	9.29
		8	1.93	4.44	16.88	16.03
		16	3.26	5.85	64.59	26.96
	53437	1	2.13	25.67	1.96	83.03
		4	6.05	58.08	23.85	250.7
		8	10.13	83.2	94.18	435.84
		16	17.23	100.38	337.30	757.62

5. Summary for Fast Algorithms for Max Slack Buffer Insertion Algorithm

In Section C to E we have described three fast algorithms for optimal buffer insertion for different cases. For nets with large number of sinks and buffer positions (over 2000 sinks and 5000 buffer positions) with small buffer library (in our experience, less than 4 types of buffers due to its multiple candidate tree overhead), $O(n \log^2 n)$ algorithm gives the good performance. For medium nets with large buffer library that general consists of 40 to 100 buffers, $O(bn^2)$ algorithm gives the best performance. For most of nets with relative small number of sinks with medium buffer library, $O(mn)$ algorithm gives the best performance.

F. Complexity Analysis and Speedup Techniques for Optimal Buffer Insertion with Minimum Cost

Van Ginneken's algorithm does not control buffering resources while it only focuses on obtaining the optimal slack. In practice, inserting 30 buffers to fix a slew constraint or to meet a delay target when 3 buffers may suffice is not acceptable as it will accelerate the buffer explosion crisis. Also, people frequently want to find the cheapest solution that meets the timing target, not necessarily the optimal solution in terms of maximum slack. Lillis, Cheng and Lin [1] presented an implementation based on (Q, C, W) framework to control resource utilization.

In this section, we show that even if there is only one buffer type, the number of non-redundant candidate solutions (Q, C, W) could be exponential in the number of potential insertion positions. Therefore, any algorithm that explicitly computes non-redundant candidates will have worst-case time complexity exponential. Furthermore, we prove for arbitrary buffer cost functions, the problem of minimizing buffering resources subject to timing constraints is NP-complete. On the other hand, we show how to apply the predictive pruning technique to minimize the buffer cost. Experiment results show that this technique can significantly speed up the running time and reduce the memory usage.

At the last of this section, we address the problem of merging branches when the number of child nodes is more than two. For these cases, previous work suggests converting the tree into a binary tree by using zero length wires. However, the mechanism for conversion can potentially yield different results. We show that one can explore the entire solution space and still maintain a polynomial algorithm as long as the maximum degree of a node is bounded by a constant

Before we start, let us brief review the (Q, C, W) framework. In van Ginneken's original algorithm, the effect of a candidate α to the upstream is described by the (Q, C) pair, where $Q = Q(v, \alpha)$ is the slack at the current tree node v and $C = C(v, \alpha)$ is the downstream capacitance. To constrain total resource usage, van Ginneken [10] suggested to add cost $W = W(v, \alpha)$ to form tuple (Q, C, W), which is implemented by Lillis, Cheng and Lin [1].

For any two candidates α_1 and α_2 of $T(v)$, we say α_1 dominates α_2, if $Q(v, \alpha_1) \geq Q(v, \alpha_2)$, $C(v, \alpha_1) \leq C(v, \alpha_2)$ and $W(v, \alpha_1) \leq W(v, \alpha_2)$. 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 any candidate of $T(v)$ is dominated by some candidates in $N(v)$.

1. Complexity Analysis

Lemma II.21. *Consider a sub-tree $T(v)$ consists of two branches $T(v_1)$ and $T(v_2)$. Assume $T(v_1)$ has 2 non-redundant candidates*

$$\alpha_1 = (\infty, L, 0)$$
$$\alpha_2 = (\infty, 0, 1).$$

Assume $T(v_2)$ has k non-redundant candidates

$$\beta_i = (Q_i, C_i, W_i), i = 1, 2, \ldots, k,$$

where $Q_i > Q_{i+1}$, $W_i \geq W_{i+1}$ for $i = 1, 2, \ldots, k-1$, and $C_i < L$ for $i = 1, 2, \ldots, k$. Then $T(v)$ has $2n$ non-redundant candidates γ_{ij}, $i = 1, 2$, $j = 1, 2, \ldots, k$, where

$$\gamma_{1i} = (Q_i, L + C_i, W_i), \gamma_{2i} = (Q_i, C_i, 1 + W_i),$$

for $i = 1, 2, \ldots, k$. Furthermore, the $2n$ new candidates satisfy above three conditions.

Proof. We first check redundancy. Clearly, γ_{i1} can not dominate γ_{j1}, since otherwise β_i would dominate β_j. Similarly, γ_{i2} can not dominate γ_{j2}. Furthermore, for any i, γ_{i1} can not dominate γ_{j2} because $L + C_i \geq L > C_j$. Finally, γ_{i2} can not dominate γ_{j1} since $Q_i > Q_j$ implies $W_i \geq W_j$ and hence $1 + W_i > W_j$. □

Theorem II.22. *If the cost of each buffer is an arbitrary integer, then the minimum cost buffer insertion problem is NP-complete.*

Proof. The problem is clearly in NP. We now show a reduction from 2-1 partition, a known NP-complete problem [37]:

Instance: Positive integers $x_1, x_2, \ldots, x_{2n}$. Let $\sum_{i=1}^{2n} x_i = 2N$

Question: Is there an index set I that contains exactly one of $2i-1$ and $2i$ for $1 \le i \le n$, such that $\sum_{i \in I} x_i = N$?

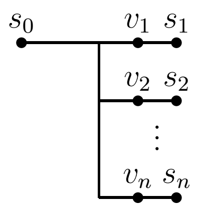


Fig. 25. Construction used for reduction, where $v_1, \ldots, v_n$ are buffer positions and $s_1, \ldots, s_n$ are sinks.

Table VIII. Construction of sinks.

Sink s_i	$C(s_i)$	$Q(s_i)$
s_1	N^{n+2}	$N^{n+1} + N^{n+2}$
s_2	N^{n+1}	$N^{n+1} + N^{n+2}$
$\vdots$	$\vdots$	$\vdots$
s_n	N^3	$N^{n+1} + N^{n+2}$

Given an instance of the 2-1 partition problem, we construct an instance of the buffer insertion problem as shown in Fig. 25. There are n sinks and $2n$ buffer types as shown in Tables VIII and IX. For source s_0, the buffer driver resistance $R(s_0) = N^n$. All wires have zero resistance and capacitance. Clearly, every number in the construction can be expressed in $O(n \log N)$ bits.

Now we claim there is a solution for the buffer insertion instance with $Q(s_0) \ge 0$ and total buffer cost at most

$$M = N + \sum_{i=1}^{n} N^i$$

if and only if there is a solution for the 2-1 partition instance.

Table IX. Construction of buffers.

Buffer b_i	$R(b_i)$	$C(b_i)$	$W(b_i)$
b_1	1	x_1	$x_2 + N^n$
b_2	1	x_2	$x_1 + N^n$
b_3	N	x_3	$x_4 + N^{n-1}$
b_4	N	x_4	$x_3 + N^{n-1}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
b_{2n-1}	N^{n-1}	x_{2n-1}	$x_{2n} + N$
b_{2n}	N^{n-1}	x_{2n}	$x_{2n-1} + N$

First assume there is a solution for the buffer insertion problem. It is easy to see that there must be a buffer at every v_i, since otherwise $R(s_0) \cdot C(s_i) \geq N^{n+3} > Q(s_i)$ for any i. Furthermore, v_1 must use either buffer type b_1 or b_2, since otherwise $R(b_i) \cdot C(s_1) \geq N^{n+3} > Q(s_0)$. Since v_1 must use either b_1 or b_2, v_2 can not use b_1 or b_2 anymore, since otherwise the total buffer cost will be at least $2N^n > M$. Repeat the argument for every i, we know the buffer types for v_i can only be b_{2i-1} or b_{2i}. This way, the delay caused by buffers at v_i is N^{n+2}, for all $i = 1, 2, \ldots, n$.

Let I be the set of buffer indices that are inserted at $v_1, \ldots, v_n$. Then

$$\begin{aligned} Q(s_0) &= \min_{1 \leq i \leq n} \{Q(s_i) - N^{n+2}\} - R(b_0) \cdot \sum_{i \in I} C(b_i) \\ &= N^{n+1} - N^n \sum_{i \in I} x_i, \\ W(s_0) &= \sum_{i \notin I} x_i + \sum_{i=1}^{n} N^i. \end{aligned}$$

Since we have both $Q(s_0) \geq 0$ and $W(s_0) \leq M$, we must have $Q(s_0) = 0$ and

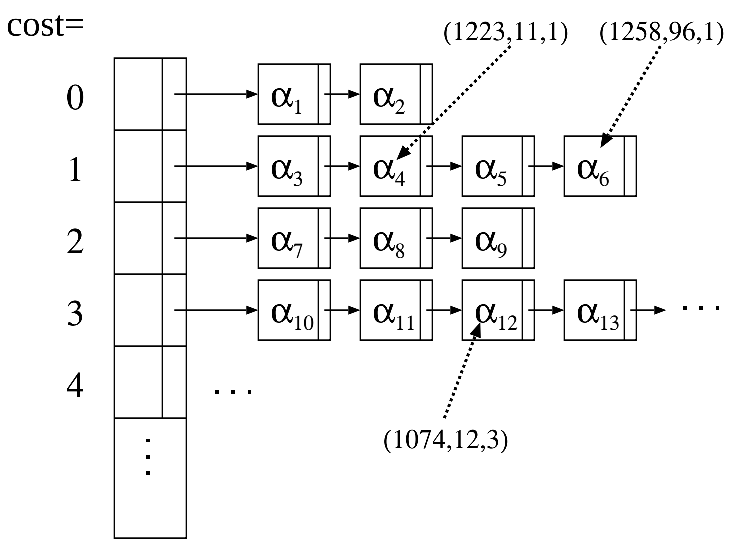
$W(s_0) = M$, which is a solution to the 2-1 partition instance.

On the other hand, any solution to the 2-1 partition instance, we can assign buffers according to the partition, and prove the solution satisfy the requirements. $\square$

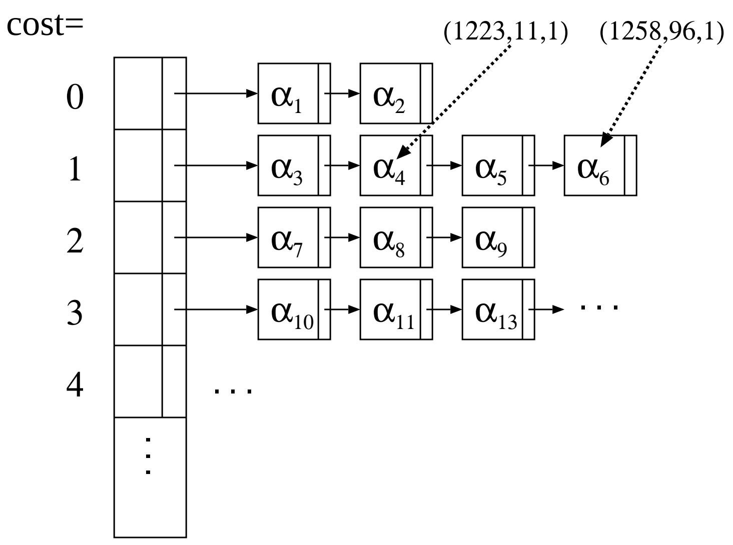
Finally, we say a few words about the difference between NP-complete and NP-hard [37]. Some literatures use NP-complete to describe a decision problem, and NP-hard to describe an optimization problem. This difference is rather technical. However, there is a fundamental difference that should be emphasized: The NP-hard class includes problems that are NP-complete, PSPACE-complete, EXTIME-complete, etc, all the way to undecidable [37]. Therefore, by saying a problem is NP-complete, it also puts an upper bound on the complexity.

2. Algorithm

The algorithm of Lillis, Cheng and Lin [1] generates candidates in a bottom up manner starting from the sinks. The candidate solutions at each node are organized as an array of linked lists as shown in Fig. 26. The solutions in each list of the array have the same buffer cost value $W = 0, 1, 2, \ldots.$ The polarity is handled by maintaining two arrays of candidate solutions. In buffer insertion algorithm, a solution can be pruned only if it is redundant, i.e., there exists another solution that is better in slack, capacitance and buffer cost. More specifically, for two candidate solutions $\alpha_1 = (Q_1, C_1, W_1)$ and $\alpha_2 = (Q_2, C_2, W_2)$, α_2 dominates α_1 if $Q_2 \geq Q_1$, $C_2 \leq C_1$ and $W_2 \leq W_1$. In such case, we say α_1 is redundant and has to be pruned. For example, in Fig. 26(a), assume $\alpha_4 = (1223ps, 11fF, 1)$ and $\alpha_{12} = (1074ps, 12fF, 3)$, α_{12} is dominated by α_4 and is then pruned. The data structure after pruning is shown in Fig. 26(b). After pruning, every list with the same cost is a sorted in terms of Q and C.


cost=
0
1
2
3
4
(1223,11,1)
(1258,96,1)
(1074,12,3)
α_1 α_2
α_3 α_4 α_5 α_6
α_7 α_8 α_9
α_{10} α_{11} α_{12} α_{13}

(a) The basic data structure storing candidate solutions


cost=
0
1
2
3
4
(1223,11,1)
(1258,96,1)
α_1 α_2
α_3 α_4 α_5 α_6
α_7 α_8 α_9
α_{10} α_{11} α_{13}

(b) After pruning

Fig. 26. Examples of data structure and pruning.

Here we assume that the cost function is the simple (yet practical) number of buffers, i.e., $W(b_i) = 1$, for all $b_i \in B$. While total buffer area can be used, to the first order, the number of buffers provides a reasonably good approximation for the buffer resource utilization. This type of function is useful if one is using just one buffer type or if one is at a stage in the design where area is not as significant as the designer effort to make an ECO (Engineering Change Order) change. Also, it is certainly a reasonable choice if all the buffers in the given library are fairly close in size. Note that, the techniques presented in [1] and this thesis can be applied on any buffer resources model, such as total buffer area or power.

Initially, each sink s has a single candidate α, whereby $Q(s, \alpha) = RAT(s_i)$, $C(s, \alpha) = C(s_i)$, and $W(s, \alpha) = 0$. There are three basic operations during the bottom-up traversal as shown from Figs. 6 to 8.

1. **Add a wire**. As one propagates candidates from node v_1 up to its parent node v, one must incorporate the delay of wire (v, v_1) into each candidate of v_1. The number of candidates does not increase (and may even decrease due to pruning).

2. **Add a buffer**. At a node v, one may potentially consider adding buffers to some subset of candidates at v_1. The number of candidates will increase, but is bounded by the number of different values of W that can possibly be generated.

3. **Merge two sub-trees**. For now, assume that every Steiner tree can transformed into an equivalent binary tree by adding zero length wires. The merging of sub-trees $T(v_1)$ and $T(v_2)$ when controlling resources is the most expensive of the three operations. We discuss this process in further detail below.

After performing these operations as required, eventually the set of candidates

$N(s_0)$ at the source is identified. One can then explore the candidates in this set to find the solution corresponding to the desired cost/slack tradeoff.

The third step of merging two branches can be problematic because the number of candidate solutions can potentially explode. Consider the example in Fig. 27.

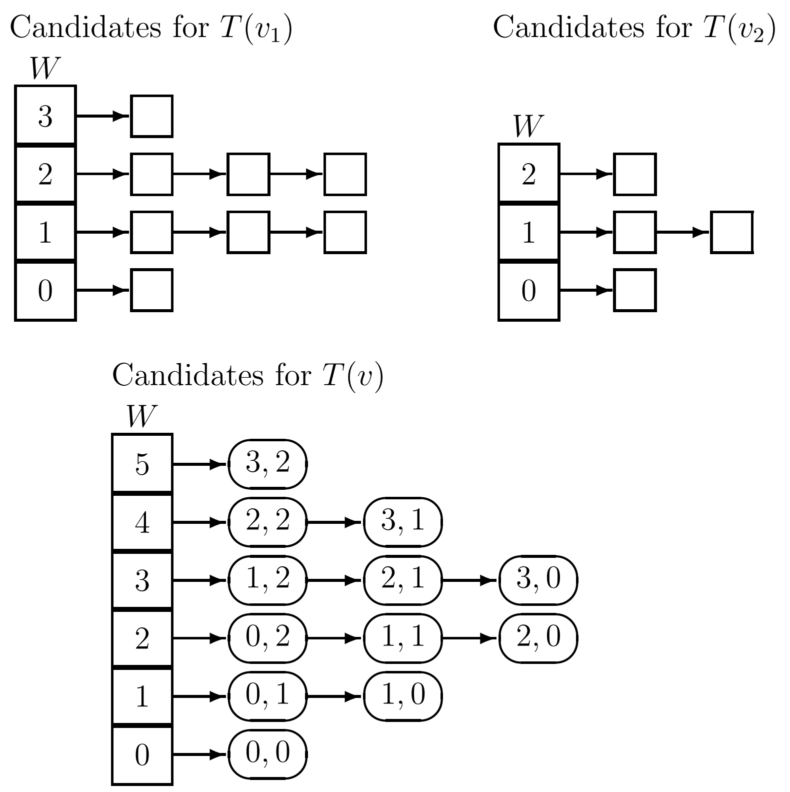


Fig. 27. Example of the algorithm for merging left and right candidates to obtain a single set of candidates for the branching point. Here, the cost function is the number of buffers inserted.

The set of candidates is stored as an array, indexed by the cost W. New candidates are generated by exploring potential merges so that the new candidates are generated in nondecreasing order of cost. For example, first the zero-buffer left candidates are merged with the zero-buffer right candidates. Then the zero-buffer left candidates are merged with the one-buffer right candidates followed by the one-buffer left candidates and the zero-buffer right candidates.

When each new candidate is generated, its capacitance and slack can be inserted into a range-query tree [1] to allow for pruning based on just Q and C. The trick is that by visiting all new candidates in nondecreasing order of cost, it is guaranteed that each new candidate added to the range-query tree will be dominated in terms of cost by the other candidates already in the tree. Then one only needs to determine additional dominance in Q and C to see whether the candidate should be rejected. This test can be done in time logarithmic in the size of the tree.

As one can see from Fig. 8, the number of candidates can potentially explode. Let n_1 and n_2 be the number of candidates in the sub-trees $T(v_1)$ and $T(v_2)$, then there can be $n_1 \cdot n_2$ possible candidates, this leads to a possibly exponential algorithm.

3. Predictive Pruning

a. General Idea

The concept of predictive pruning has been proposed in Section C. For now assume there is only one buffer type B. When we compare candidates at v, it is insufficient to only compare Q and C values at v. Instead, we want a candidate α that maximizes slack

$$P(v,\alpha) \quad = \quad Q(v,\alpha) - K(B) - R(B) \cdot C(v,\alpha), \tag{2.8}$$

among all candidates. However, such a candidate is not necessarily the candidate that maximizes Q. It is because when a buffer is attached, some nonredundant candidates might become redundant.

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 $C(v,\alpha_1) \leq C(v,\alpha_2)$. There are two important lemmas are shown in

Section C, Lemmas II.4 and II.5.

We can expand the above concept and lemmas considering the buffer cost. For any candidates α_1 and α_2 of $T(v)$, we say α_1 *B-dominates* α_2 if $P(v, \alpha_1) \geq P(v, \alpha_2)$, $C(v, \alpha_1) \leq C(v, \alpha_2)$ and $W(v, \alpha_1) \leq W(v, \alpha_2)$.

Lemma II.23. *If α_1 B-dominates α_2, then α_2 is redundant.*

Proof. From Lemma II.4, we know that α_2 is redundant in terms of (Q, C). Since $W(v, \alpha_1) \leq W(v, \alpha_2)$, and for any cases of adding a buffer, adding wire delay or merging, same cost will be added to both α_1 and α_2, then α_2 is redundant. $\square$

It is easy to see if α_1 dominates α_2, then α_1 B-dominates α_2, and α_2 will be pruned. 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, W) pruning, except that some candidates that are nonredundant under (Q, C, W) are redundant under (P, C, W).

Most traditional buffer insertion algorithms based on (Q, C, W) can be improved using the new pruning technique based on (P, C, W). Also, this method does not change the data structure or the frame of previous algorithms. In the application, the value P does not need to be stored. It can be computed from Q and will only be used when the redundancy is checked.

For multiple buffer types, the (P, C, W) pruning is defined for each type of buffer B_i: $P_i(v, \alpha) = Q(v, \alpha) - K(B_i) - R(B_i) \cdot C(v, \alpha)$. 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 $P_i(v, \alpha_1) \geq P_i(v, \alpha_2)$, $C(v, \alpha_1) \leq C(v, \alpha_2)$, and $W(v, \alpha_1) \leq W(v, \alpha_2)$.

b. Application to Buffer Cost Minimization

For buffer insertion with minimum cost, we use the framework of Lillis, Cheng and Lin [1] and apply the predictive pruning technique to both solution set indexed by cost and the range-query tree (if we perform on only one of them, the optimal solution can still be achieved except that few redundant solutions will be pruned). Since all new candidates are visited in nondecreasing order of cost, the predictive pruning technique guarantees the optimality.

Multiple buffer types are also considered here. However, this will increase the space complexity by a factor of b. We can avoid the extra space increase by only pruning those candidates B_k-dominated by other candidates, where B_k is the buffer with the smallest $R(B_i)$ among all buffers. There will be less redundant solutions pruned compared with using b lists, but our experiments show that there are still many redundant solutions been pruned, compared with previously Q pruning technique.

c. Experimental Results

To show the advantage of new pruning technique, we tested our new algorithm for buffer insertion with cost constraints. Six different buffer types based on TSMC 180 nm technology are used. For the smallest buffer(1X), $R(B) = 1440\Omega$, $C(B) = 2.9$ fF, and $K(b) = 36.4$ ps. The largest buffer is 8X. 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 $fF/\mu m$. Intrinsic gate delay is identical for all buffers. All algorithms are implemented in C and run on a Sun SPARC workstations with 400 MHz and 2 GB memory. The implemented algorithms also output buffer positions.

The first experiment is to compare our new technique with Lillis, Cheng and Lin's original algorithm [1] for buffer cost minimization. The buffer cost is the number of buffers. In Table X, the time and memory results are shown for six buffer types and the number of buffer positions are the number of sinks. Our algorithm is 2 to 17 times faster than Lillis' algorithm and uses 1/1.5 to 1/30 of memory. The performance of our algorithm is better when the size of buffer library is large. Since generally the buffer library is large in industry designs, so our algorithm can achieve more significant improvement in practice.

We also compare our algorithm with the $O(b^2 n \log^2 n)$ time algorithm of Shi and Li [33], without considering the buffer cost. The $O(b^2 n \log^2 n)$ time algorithm is asymptotically the fastest algorithm reported. Simulation results are shown in Table XI for six buffer types on several industrial test cases. We can see that for multiple buffer types without considering buffer cost, our new algorithm is better than the $O(b^2 n \log^2 n)$ time algorithm when n is small. The reason is that $O(b^2 n \log n)$ algorithm needs to use b trees to store B_i-dominant solutions, which results in high overhead. Instead, our new algorithm only needs one list to store B_k-dominant solutions, for B_k that the smallest driving resistance.

One benefit of predictive pruning algorithm is that it is very flexible and can be combined with many buffer insertion algorithms, such as ones described in Section D and Section E, and the performances of those algorithms are all improved.

4. High Degree Vertices

During buffer insertion, one problem is to deal with routing tree vertices of out-degree greater than 2. Although any such vertex can be replaced by a number of out-degree 2 vertices, the order is not unique. For example, there are three ways to replace an out-degree 3 vertex in Fig. 28 with out-degree 2 vertices. Each new vertex, solid or hollow, is a possible buffer position. Solid vertices involves merging and buffer insertion. Hollow points involve buffer insertion only.

Table X. Simulation results for six buffer types with buffer cost constraints, where m is the number of sinks, n is the number of buffer positions, and W is the buffer cost.

m	n	W	**CPU Time (sec)**		**Speed-up**	**Memory (MB)**		**Reduc-tion**
			Lillis (Q, C, W)	New (P, C, W)		Lillis (Q, C, W)	New (P, C, W)	
337	336	30	2.51	0.79	3.2	0.91	0.22	4.1
		50	3.11	1.09	2.9	1.39	0.25	5.6
		100	4.58	2.48	1.8	1.52	0.26	6.1
1944	1943	30	142.56	9.80	14.5	13.14	0.70	18.8
		50	244.90	14.14	17.3	24.15	0.96	25.2
		100	470.87	26.84	17.5	53.06	1.76	30.1
2676	2675	30	196.23	22.26	8.8	12.50	1.60	7.8
		50	347.86	33.63	13.0	22.86	2.26	10.1
		100	559.54	57.81	9.7	42.82	3.81	11.2

Theorem II.24. *Let v be an out-degree d vertex, then there are at most $n + 2^{O(d)}$ nonredundant candidates for $T(v)$, where n is the total number of candidates in the branches.*

Proof. For a vertex with out-degree d, the number of ways to merge the d branches by a sequence of 2-merges is $1 \cdot 3 \cdot 5 \cdots (2d-3) = 2^{O(d)}$. This can be shown by solving the recurrence relation using generation function [38]. Fig. 28 shows the three ways for $d = 3$. We will first consider whether to insert a buffer at each hollow point. This way, the total number of candidates in the branches is at most $n + d$.

Now consider the merging points, shown as the solid points in Fig. 28. In each case, say Fig. 28(b), since there can be at most $d - 1$ new buffers, the number of candidates with new buffers is at most d. The candidates without new buffers will be the same as the candidates without new buffers in any other case, say Fig. 28(c). Therefore if we combine all the cases, there will be only $2^{O(d)}$ additional candidates. $\square$

Table XI. Simulation results for six buffer types without buffer cost constraints, where m is the number of sinks and n is the number of buffer positions.

m	n	**CPU Time (sec)**		**Memory (MB)**	
		$O(b^2 n \log n)$	New	$O(b^2 n \log n)$	New
337	336	0.42	0.04	0.12	0.05
	2992	2.45	0.45	0.24	0.25
	26887	25.84	27.96	1.56	5.91
1944	1943	2.42	0.28	0.23	0.21
	17538	16.09	4.23	0.54	1.34
	95513	96.57	101.32	2.15	21.70
2676	2675	3.32	0.41	0.24	0.25
	23875	21.92	6.26	0.58	1.88
	129875	132.65	153.1	2.23	34.84

G. Approximation Techniques for Buffer Insertion with Minimum Cost

With the buffer library and cost consideration, van Ginneken's algorithm becomes practical yet more runtime intensive. The work in [2], which is also shown in Section F, has proved that minimizing buffering resource in buffer insertion is NP-complete. In this section, we propose three approximation techniques to further speed up the min-cost buffer insertion algorithm in [1].

1. **Aggressive Predictive Pruning (APP)**. When considering candidate buffer solutions at a node, it can be concluded that there must exist a minimum resistance driving this node, which leads to additional delay. This delay can be factored in during pruning to prune more aggressively than van Ginneken's

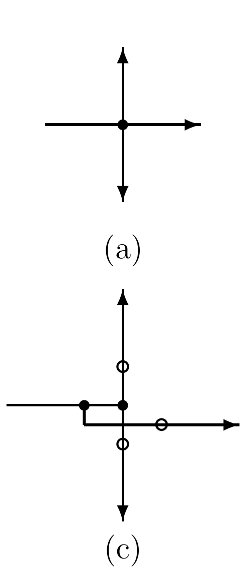


Fig. 28. Three ways to replace an out-degree 3 vertex (top left) by out-degree 2 vertices. For an out-degree d vertex, there are $1 \cdot 3 \cdot 5 \cdots (2d-3)$ ways.

algorithm allows. One can even increase this resistance value to prune more aggressively but potentially sacrifice optimality.

2. **Convex Pruning (CP)**. When examining three sorted candidates, one may be able to guess that the middle one will soon become dominated by the one just below or just above it in the candidate solution list. By determining this before the candidate actually becomes dominated, squeeze pruning allows it to be removed earlier.

3. **Library Lookup (LL)**. During van Ginneken style buffer insertion, one considers inserting each buffer from a library for a given candidate. Instead, Library Lookup considers just the inverting and non-inverting buffer from the library that yields the best delay. These are determined from a pre-computed lookup table.

Here, we use a simple cost function to find the solution with the minimum number of buffers that meets the delay target and fixes the slew constraint. Also, we consider a discrete set of non-inverting and inverting buffers of various power levels. The framework of Lillis, Cheng and Lin's algorithm has been shown in Section F. Note that at the end of the algorithm, a set of solutions with different cost-RAT tradeoff is obtained. Each solution gives the maximum RAT achieved under the corresponding cost bound. Practically, we choose neither the solution with maximum RAT at source nor the one with minimum total buffer cost. Usually, we would like to pick one solution in the middle such that the solution with one more buffer brings marginal timing gain. In our implementation, we use the following scheme namely the "10ps rule". For the final solutions sorted by the source's RAT value, we start from the solution with maximum RAT and compare it with the second solution (usually it has one buffer less). If the difference in RAT is more than 10ps, we pick the first solution. Otherwise, we drop it (since with less than 10ps timing improvement, it does not worth an extra buffer) and continue to compare the second and the third solution. Of course, instead of 10ps, any time threshold can be used when applying to different nets.

Different from the work of [32, 2] and Section F, which emphasizes more on large and huge nets, our techniques are more effective on small and medium nets which are the majority in most chip designs. Our experiments on thousands of nets from industry designs show that when we combine these three techniques, we are able to gain a factor of 9$\times$ to 25$\times$ speedup over traditional buffer insertion algorithm while the slack only degrades by 2-3% on average.

These techniques can be easily integrated with the current buffer insertion engine which considers slew, noise and capacitance constraints. Consequently, we believe these techniques are essential to embed in a physical synthesis buffer insertion system.

1. Aggressive Predictive Pruning (APP)

Aggressive predictive pruning is based on predictive pruning. It has been shown in Section F that the technique is very effective and produces optimal results. The detail proof and discussion can be found in Section C and F. Intuitively, predictive pruning is based on anticipated upstream resistance (such as buffer or driver resistance) of at least R_{min}.

Predictive Pruning (PP): For two non-redundant solutions (Q_1, C_1, W_1) and (Q_2, C_2, W_2), where $Q_1 < Q_2$, $C_1 < C_2$, and $W_1 \leq W_2$, if $(Q_2 - Q_1)/(C_2 - C_1) \geq R_{min}$, then (Q_2, C_2, W) is pruned.

For example, in Fig. 26(b), we assume $R_{min} = 1k\Omega$. For two non-dominating candidates $\alpha_4 = (1223ps, 11fF, 1)$ and $\alpha_6 = (1258ps, 96fF, 1)$, since $1258 - 1000 \times 0.096 = 1162 < 1223 - 1000 \times 0.011 = 1212$, α_6 is pruned since we certainly predict α_6 will be dominated by α_4 when the bottom-up process continues. In other words, the extra $85fF$ of capacitance will add at least $85ps$ of upstream delay eventually. Thus, the $25ps$ slack advantage of α_4 is overshadowed by its weakness of larger capacitance.

An interesting observation is that if we use a resistance R which is larger than R_{min} value, obviously more solutions are pruned and the algorithm becomes faster. However, it will sacrifice the solution quality. This technique is referred to as the aggressive predictive pruning - APP since it does more aggressive pruning than predictive pruning. In order to investigate the relationship of algorithm speed-up and solution sacrifice for APP, we have performed a set of experiments on 1000 industrial nets (with a buffer library consisting of 24 buffers). The comparison is shown in Fig. 29. The figure shows the degradation in slack and the decrease in CPU time as a function of resistance. When $R = 120\Omega$, this is the minimum resistance value which still yields the optimal solution. As R increases, the CPU time drops much more sharply than the slack. For example, one can get a 50% speedup for less than 5% slack degradation when $R = 600\Omega$. Also, note from the bottom chart that the number of buffers stays fairly stable until R gets quite large. The promising experiment results show that by using APP, a tiny sacrifice in solution quality can bring a huge speed-up in the van Ginneken's algorithm.

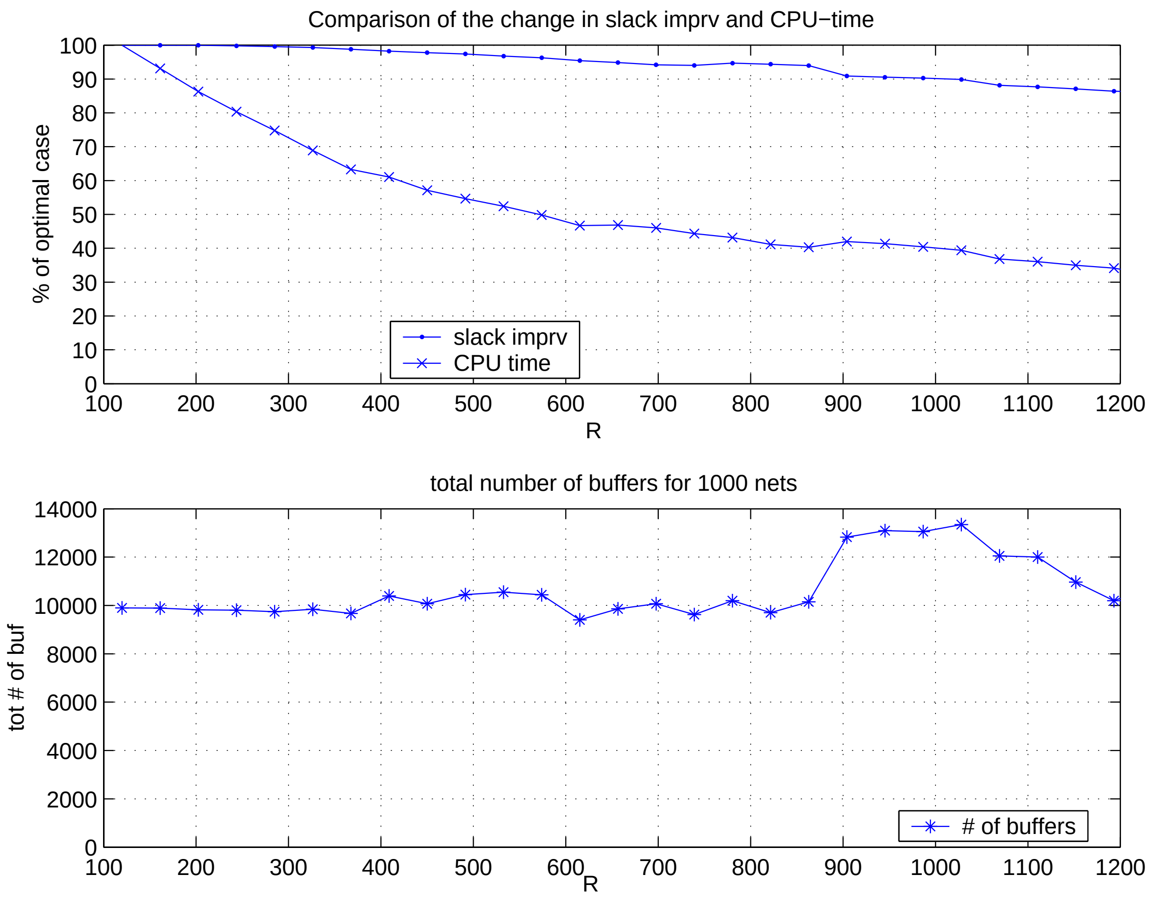


Fig. 29. The speed-up and solution sacrifice of APP in 1000 nets

2. Convex Pruning (CP)

The basic data structure of van Ginneken style algorithms is a sorted list of non-dominated candidate solutions. Both the pruning in van Ginneken style algorithm and the predictive slack pruning are performed by comparing two neighboring candidate solutions at a time. However, more potentially inferior solutions 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 the convex pruning. The concept of convex pruning was first proposed by Li and Shi [35] and has been explained in Section D and E. It has been shown that in Lemma II.16 that convex pruning preserves optimality for two-pin nets.

For a multi-sink net with Steiner nodes, convex pruning can not keep optimality since each candidate solution may merge with different candidate solutions from the other branch. For example, the middle candidate solution in Fig. 22(b) may offer smaller capacitance to other candidate solutions in the other branch. Convex pruning may prune out a post-merging candidate solution that is originally with less total capacitance. Therefore, the final solution may be sub-optimal. However, convex pruning only causes little degradation on the solution quality since it is performed for each set of solutions with the same cost and the capacitance under-estimation effect is alleviated.

One example of squeeze pruning that is not optimal at a Steiner node is shown as follows: Suppose the candidate solutions at the right branch are $\alpha_{r1} = (1186ps, 13fF, 12)$, $\alpha_{r2} = (1190ps, 150fF, 12)$ and $\alpha_{r3} = (1243ps, 201fF, 12)$, and the candidate solutions at the left branch are $\alpha_{l1} = (1187ps, 20fF, 10)$ and $\alpha_{l2} = (1200ps, 40fF, 10)$. If convex pruning are performed after merging point, the candidate solutions after merging are $\alpha_1 = (1186ps, 33fF, 22)$, $\alpha_2 = (1190ps, 190fF, 22)$. If convex pruning are performed before merging point, the candidate solutions after merging are $\alpha_1 = (1186ps, 33fF, 22)$, $\alpha_2 = (1187ps, 221fF, 22)$. It can be seen that convex pruning is sub-optimal at merging point except for the case when the convex pruning is only performed after last merging point.

Note that, the predictive and aggressive predictive slack pruning techniques prune the second candidate solution when the slope value between every two neighboring candidate solutions is smaller than a threshold value. We can treat these two techniques as special cases of convex pruning if we assume there is a dummy third candidate solution with the slope to the first candidate solution being that threshold value.

3. Library Lookup (LL)

In van Ginneken style algorithms, the size of buffer library is an important factor. Modern designs often have tens of power levels for buffers and inverters. Sometimes it climbs into the hundreds. From the algorithm analysis, the size of buffer library has the square effect on the running time. In practice, this effect appears to be linear, but it is still a bottleneck when we perform buffer insertion with large buffer libraries. On the other hand, it is essential to have a reasonably sized library to obtain sufficient timing performance. In [13], a buffer library selection algorithm is proposed to prune the big library to get small library and use small library to perform buffer insertion. In Section D, an $O(bn^2)$ algorithm is proposed for optimal buffer insertion with b buffer types. Is that possible to speed up further to reduce b effect to constant with a little degradation of solution quality?

During van Ginneken style buffer insertion, every buffer in the library is tried for each candidate solution. If there are n candidate solutions at an internal node before buffer insertion and the library consists of b buffers, then bn tentative solutions are evaluated. For example, in Fig. 30(a), all eight buffers are considered for all n candidate solutions.

However, many of these candidate solutions are clearly not worth considering (such as small buffers driving large capacitance). But van Ginneken style algorithms generate them anyway and let pruning step eliminates the redundant candidate solutions. Instead, we seek to avoid generating poor candidate solutions in the first place and not even consider adding m buffered candidate solutions for each unbuffered candidate solution. We propose to consider each candidate solution in turn. For each candidate solution with capacitance C_i, we look up the best non-inverting buffer and the best inverting buffer that yield the best delay from two pre-computed tables before optimization. In the example shown in Fig. 30(b), the capacitance C_i results in selecting buffer $B3$ and inverter $I2$ from the non-inverting and inverting buffer tables.

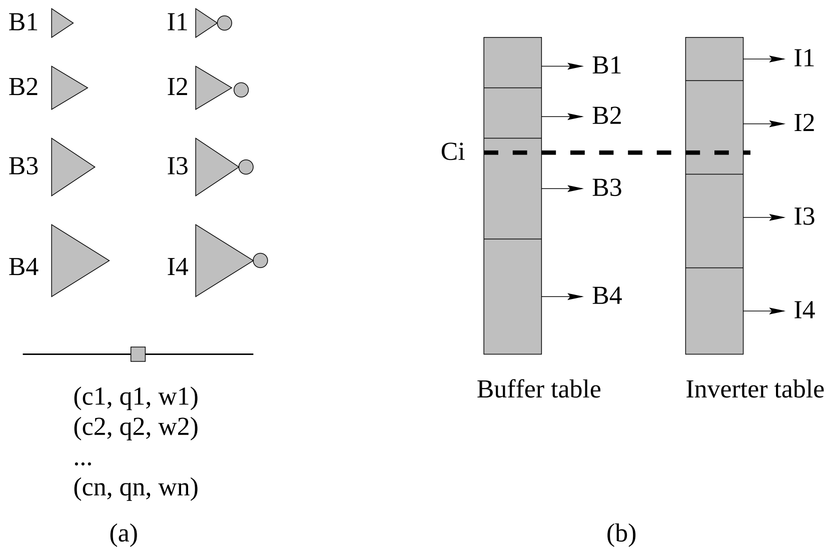


Fig. 30. Library Lookup example. $B1$ to $B4$ are non-inverting buffers. $I1$ and $I4$ are inverting buffers. (a) van Ginneken style buffer insertion. (b) Library Lookup.

Two pre-computed tables are built as follows: for each possible capacitance value, we gives the non-inverting (inverting) buffer with minimum delay when a driver with average size drives this buffer driving this capacitance. Using a table with 300 entries can be quickly computed before buffering and gives more than sufficient granularity.

All $2n$ tentative new buffered candidate solutions can be divided into two groups, where one group includes n candidate solutions with an inverting buffer just inserted and the other group includes n candidate solutions with a non-inverting buffer just inserted. We only choose one candidate solution that yields the maximum slack from each group and finally only two candidate solutions are inserted into the original candidate solution lists. Since the number of tentative new buffered solutions is reduced from bn to $2n$, the speedup is achieved. Also, since only two new candidate solutions instead of b new candidate solutions are inserted to the candidate solution lists (these new solutions could be pruned later), the number of total candidate solutions are reduced. It is equivalent to the case when the buffer library size is only two, but the buffer type can change depending on the downstream load capacitance and all b buffers in the original library can be used if they are all listed in the table. This is the major difference between this technique and library pruning in [13] since after library pruning, only those surviving buffers can be used.

4. Experimental Results

The proposed techniques are implemented together with the buffer insertion algorithm in C and are tested on a SUN SPARC workstations with 400 MHz and 2 GB memory. We test our speedup techniques on three groups of nets from industry ASICs with 300K+ gates. They have been placed and require physical synthesis to optimize timing and fix electrical violations. The first group are extracted from one ASIC chip and consists 1000 most time consuming nets for the algorithm of Lillis, Cheng and Lin [1]. We named it as ChipA-1K. The second group and third group are extracted from another ASIC chip and consist 1000 and 5000 most time consuming nets for the algorithm in [1], respectively. We named them as ChipB-1K and ChipB-5K. We choose the third group since this group includes more nets of small and middle size. The net information for these three groups is shown in Table XII.

Table XII. Net information.

# sinks m	$m \leq 5$	$5 < m \leq 20$	$20 < m \leq 50$	$50 < m \leq 100$	$m > 100$
# nets in ChipA-1k	944	56	0	0	0
# nets in ChipB-1k	0	29	581	345	45
# nets in ChipB-5k	2	1478	2956	513	51

The buffer library (denote by *Full*) consists of 24 buffers, in which 8 are non-inverting buffers and 16 are inverting buffers. The range of driving resistance is from 120 Ω to 945 Ω, and the input capacitance is from 6.27 fF to 121.56 fF. We use a scaled Elmore delay as interconnect delay, and apply the 10 ps rule to choose the most cost efficient solution from a set of solutions with different cost-slack tradeoff.

Table XIII shows the simulation results for these three test groups. In each experiment, we show the total slack improvement after buffer insertion, the number of inserted buffers for all nets and the total CPU time of the buffer insertion algorithm with our speedup techniques, APP, CP and LL. Finally we show the results when three techniques are combined. For comparison, we also show the results of the algorithm of Lillis, Cheng and Lin (baseline) [1] and predictive pruning technique (PP) in Section F and [2]. Note that for the CP and LL, we also combine them with predictive pruning technique. For APP, the resistance is chosen as 5% of the range from minimum buffer resistance to the maximum buffer resistance plus the minimum resistance. In [2], it is claimed that predictive slack pruning can achieve up to 17 times speedup. This huge speedup is achieved mainly because the size of test nets and the number of buffer locations in [2] are much larger than our test cases.

The results show that our speedup techniques can provide 9× to 25× speedup

Table XIII. Simulation results for ChipA-1K, ChipB-1K and ChipB-5K nets on *Full* library consisting of 24 buffers. Baseline are the results of the algorithm of Lillis, Cheng and Lin[1]. PP are results of predictive pruning technique [2]. W is the number of buffers.

Test Case	**Algorithm**	**Slack Imp. (ns)**	W	**CPU (s)**	**Speedup**
ChipA-1K	Baseline	5954.93	9895	315.14	1
	PP	5954.93	9895	280.67	1.12
	APP	5954.84 (-0.001%)	9893	267.33	1.18
	PP+CP	5954.91 (-0.000%)	9895	185.56	1.70
	PP+LL	5945.47 (-0.16%)	9723	43.87	7.18
	APP+CP+LL	5945.44 (-0.16%)	9724	33.50	9.41
ChipB-1K	Baseline	1310.62	9295	1861.26	1
	PP	1310.62	9295	860.32	2.16
	APP	1311.64 (+0.08%)	9475	737.38	2.52
	PP+CP	1311.04 (+0.03%)	9482	433.66	4.29
	PP+LL	1288.29 (-1.7%)	9370	144.89	12.85
	APP+CP+LL	1290.01 (-1.57%)	9746	75.00	24.82
ChipB-5K	Baseline	2868.86	30438	3577.38	1
	PP	2868.86	30438	1881.78	1.90
	APP	2870.32 (+0.05%)	30702	1692.77	2.11
	PP+CP	2868.95 (+0.03%)	30822	1036.20	3.45
	PP+LL	2782.25 (-3.02%)	29080	312.96	11.43
	APP+CP+LL	2785.36 (-3.00%)	29776	175.66	20.36

over baseline buffer insertion while the slack only degrades by 2-3%. From the table, for ChipB-1K and ChipB-5K nets, the slack improvement of our speedup techniques is slightly greater than the baseline algorithm in few cases. This is due to the usage of the 10 *ps* rule in which the most cost-efficient solution is selected instead of the maximum slack solution.

It is obvious that if the size of buffer library becomes smaller, the running time can also go down with the degradation of the solution quality. Since our three techniques are independent of the library size, it is interesting to compare our techniques with the baseline algorithm with different buffer libraries. We adopt the buffer library

Table XIV. Simulation results for ChipA-1K, ChipB-1K, ChipB-5k nets on different libraries. The number after each library is the library size. PP are results of predictive pruning technique [2]. W is the number of buffers.

Test Case	**Library**	**Algorithm**	**Slack Imp. (ns)**	***W***	**CPU (s)**
ChipA-1k	Tiny(4)	PP	5904.61	9864	59.09
	Small(6)	PP	5912.07	9843	77.48
	Medium(9)	PP	5949.30	9720	101.27
	Large(13)	PP	5951.46	9771	144.32
	Full(24)	APP+CP+LL	**5945.44**	**9724**	**33.50**
ChipB-1k	Tiny(4)	PP	1287.11	10235	220.02
	Small(6)	PP	1298.06	9511	237.47
	Medium(9)	PP	1305.26	9274	336.19
	Large(13)	PP	1306.93	9292	460.69
	Full(24)	APP+CP+LL	**1290.01**	**9746**	**75.00**
ChipB-5k	Tiny(4)	PP	2755.03	32361	460.43
	Small(6)	PP	2802.99	29758	509.72
	Medium(9)	PP	2844.57	29895	737.19
	Large(13)	PP	2853.04	30225	1009.91
	Full(24)	APP+CP+LL	**2785.36**	**29776**	**175.66**

selection algorithm [13] and generate four different buffer libraries from our original library. The libraries are named *Tiny*, *Small*, *Medium* and *Large*, and they have 4, 6, 9 and 13 non-inverting and inverting buffers respectively. Then we run predictive pruning technique on each library, and the results for three test groups are shown in the Table XIV. For the ease of comparison, the results for APP+CP+LL on original Full library are also shown in the last row for each test group. From the results we can see that with APP+CP+LL, the running time is even faster (up to 3$\times$ faster) than the results on *Tiny* buffer library, while we still achieve better slack improvement (up to 1%) and use less buffers. This shows that applying our techniques on a large buffer library could achieve both better slack improvement and faster running time than applying traditional van Ginneken style algorithm on a small buffer library.

CHAPTER III

WIRE SIZING FOR NON-TREE NETWORKS

Most existing methods for interconnect wire sizing are designed for RC trees. With the increasing popularity of the non-tree topology in clock networks and multiple link networks, wire sizing for non-tree networks becomes an important problem.

In this Chapter, we propose the first systematic method to size the wires of general non-tree RC networks. Our method consists of three steps: decompose a non-tree RC network into a tree RC network such that the Elmore delay at every sink remains unchanged; size wires of the tree; and merge the wires back to the original non-tree network. All three steps can be implemented in low order polynomial time. Using this method, previous wire sizing techniques for tree topology for various objectives, such as minimizing the maximum delay, minimizing the total area or power, reducing skew variability under process variations can be applied to non-tree topologies. For certain types of networks, such as the tree+link network [4], our method gives the optimal solution, provided the tree wire sizing is optimal.

A. Previous Work

Wire sizing plays an important role in achieving desirable circuit performance [39, 40, 41, 42]. In earlier work, the wire sizing problem under Elmore delay model is to minimize weighted average delay and optimal algorithm for discrete wire size is proposed in [39]. Later, sensitivity based heuristics and convex programming techniques are used to minimize the maximum delay [40]. For different objectives such as minimizing total area subject to delay bounds or minimizing maximum delay, these problems can be reduced to solve a sequence of weighted sink delay problems by Lagrangian relaxation [43]. Wire sizing under multiple input sources is handled in [44], and more accurate delay models are used in [41, 45]. For a single interconnect wire, the optimal wire sizing solution can be obtained in close form with or without constraints on the wire size [46, 47]. In [48, 49], wire sizing techniques are used to reduce the skew in the clock trees with minimum delay/area/power objective. However, all these work can only handle tree topology. One major reason is that Elmore delay in an RC tree can be modelled as a posynomial function in terms of wire width, similar to gate sizing problem that was identified as a posynomial function in [50], and in turn wire sizing problem is a convex program except for the skew problem. Note that the continuous wire sizing problem can be solved in linear time in each iteration [51] while the given fixed precision of the solution. the discrete wire sizing problem can be solved in $O(n^r)$ time, where r is the number of wire choices.

It has been shown that compared with tree topologies, non-tree topologies can effectively reduce the maximum delay [52, 53, 54], improve the yield [52, 55] and reduce the clock skew under process variations [56, 54, 4].

However, wire sizing for general non-tree topologies is more complicated since Elmore delay for non-tree topology cannot be modelled as a posynomial function. Some methods [53, 3] add multiple links to an existing interconnect and use sequential quadratic programming to optimize the wire size to reduce the maximum delay and skew. In [57], a delay model based on dominant time constant is used since the dominant time constant of a general RC circuit is a quasiconvex function of the conductances and capacitances. Semidefinite programming is used to minimize the total area and the dominant time constant. However, dominant time constant can only measure the delay of the node with slowest response, and it can not evaluate the weighted sum of the delay or the skew in the circuit, which is more important for the clock network. In [54], an iterative linear programming approach is used to size the wire on the top level edges of one-level mesh and a heuristic approach based on sensitivity information is used to size the tree edges. However, it is restricted to simple topologies. Generally, these approaches are inefficient for large circuits and general topologies due to the high computational complexity of quadratic programming and semidefinite programming.

B. Delay Models and Problem Formulations

The wire sizing problem for non-tree topology can be modelled as follows. A circuit is given as a graph $G = (V, E)$, where $V = \{v_1\} \cup V_s \cup V_n$, and $E \subseteq V \times V$. Vertex v_1 is the *source* node, V_s is the set of *sink* nodes, and V_n is the set of *internal* nodes. The source v_1 is associated with driver resistance $R(v_1)$, and each sink node $v_i \in V_s$ is associated with sink capacitance $C(v_i)$ and required arrival time $RAT(v_i)$. If v_i is an internal or source node, $C(v_i)$ is 0.

In general VLSI circuit, there could be multiple metal layers. Let the wire resistance for layer m be r_{0m} per square and wire capacitance be c_{0m} per square. Each edge e is associated with length $L(e)$ and width $W(e)$. Therefore, its lumped resistance is $R(e) = r_{0m}L(e)/W(e)$ and its lumped capacitance is $C(e) = c_{0m}L(e)W(e)$ if the edge is on layer l. The π model is used to model the R and C of each edge.

Following previous work [4, 3, 57, 58], we use Elmore delay to evaluate an RC network due to its high fidelity. According to [59], we have

Definition III.1. *For any pair of nodes v_i and v_j, define R_{ij} as the absolute value of the voltage at v_i when a unit current is injected to v_j. For any node v_j, define C_j as the node capacitance at v_j, or $C_j = C(v_j) + \sum_{e=(v_i,v_j)} C(e)/2$.*

It was shown in [59] that the vector of the Elmore delay at every node in the RC network is $R \cdot C$, where $R = \{R_{ij}\}_{n\times n}$ is the resistance matrix defined above and $C = \{C_j\}_n$ is a vector of node capacitance defined above.

If every node capacitance C_j in an RC network is treated as a current source whose current value is equal to C_j and the source v_1 is grounded, then the original RC network can be transformed to an equivalent DC network with only resistors and current sources. It was shown that the first moment of every node in the RC network, which is the *negative* Elmore delay, is equal to the voltage at every node in its corresponding DC equivalent network [60].

Once we decide a wire sizing solution $W(e)$ for every edge $e \in E$, the delay from source to any node v_i is $D(v_i) = \sum_{j=1}^{n} R_{i,j}C_j$. The slack at v_1 is $Q(v_1) = \min_{v_k \in V_s}\{RAT(v_k) - D(v_k)\}$, where $RAT(v_k)$ is the require arrival time at v_k. The weighted sum of sink delays is $T = \sum_{v_i \in V_s} \lambda_i D(v_i)$, where λ_i is the weight of the delay penalty to sink v_i. The greater λ_i is, the more critical sink v_i is. The total weighted area is $A = \sum_{e \in E} \beta(e)W(e)$, where $\beta(e)$ is weight of the each edge. The power of the circuit can be modelled as the total capacitance of the circuit. The skew between node v_i and v_j is $S(i,j) = D(v_i) - D(v_j)$ and the maximum skew of the circuit is defined as $\max_{v_i,v_j \in V_s} |S(i,j)|$.

Non-Tree Topology Sizing Problem: Given a circuit represented by routing graph $G = (V, E)$, driver resistance $R(v_1)$, sink capacitance $C(v_i)$ and $RAT(v_i)$ for each sink v_i, capacitance $C(e)$ and resistance $R(e)$ for each edge e, discrete wire width choices $\{W_1, W_2, \ldots, W_r\}$ or continuous range $[W_L(e), W_U(e)]$, find the wire width $W(e)$ for each $e \in E$ that minimizes the weighted sum of sink delays T, or maximizes source slack $Q(v_1)$, or minimizes the total area A or power subject to $Q(v_1) \geq 0$, or achieves zero skew for any two sinks.

C. Tree Decomposition

In the DC equivalent network of the original RC network, associate a current $I(e) = (-D(v_i) - (-D(v_j)))/R(e) = (D(v_j) - D(v_i)/R(e)$ with each edge $e = (v_i, v_j)$. A current path p from source v_1 to v_i is a sequence of nodes $v_{i_0}, \ldots, v_{i_l}$, where $v_{i_0} = v_1$ and $v_{i_l} = v_i$, and $I(e) > 0$ for every edge in the path. It is easy to see that the Elmore delay at vertex v_i can be expressed as $D(v_i) = \sum_{e \in p} R(e)I(e)$, where p is any current path from source v_1 to v_i.

From now on, we do not differentiate an RC network and its equivalent DC network. We always assume the capacitance is replaced by the corresponding current source.

For every node v_i, let $e_{i1}, e_{i2}, \ldots, e_{id}$ be its adjacent edges. Among these edges, call the edges with current flowing into v_i as *incoming edges* of node v_i and the other edges as *out-going edges* of node v_i. It is obvious that in an RC tree, there is exactly one incoming edge for every node, except for the source node. In a non-tree RC network, however, some nodes may have more than one incoming edge. Intuitively, if we can make every node in a non-tree RC network have only one incoming edge without changing the Elmore delay, then the non-tree RC network can be decomposed to an RC tree. Note that, we also want this RC tree realistic in order to perform wire sizing, which means the node capacitance must be non-negative and the edge capacitance correlated with the edge length and edge width. We now define three main operations that will be used in our algorithm.

Edge Cutting: For an edge $e = (v_i, v_j)$, if $0 \leq I(e) \leq 0.5C(e)$, then we can cut edge e at the point v_k such that the distance from v_i to v_k is $xL(e)$, where $x = I(e)/C(e) + 0.5$ and two nodes v_{k1} and v_{k2} that are equivalent to node v_k are generated. See Fig. 31.

Lemma III.2. *Edge cutting keeps the Elmore delay at all nodes in the original RC network unchanged and $D(v_{k1}) = D(v_{k2})$.*

Proof. From Fig. 31, in the original circuit the current flowing from A to node v_i is $I(e)+0.5C(e)$, the current flowing from B to v_j is $0.5C(e)-I(e)$. After edge cutting, the current flowing from A to node v_i is $xC(e) = I(e) + 0.5C(e)$, the current flowing from B to node v_j is $(1-x)C(e) = C(e) - (I(e) + 0.5C(e)) = 0.5C(e) - I(e)$. Since $I(e) \leq 0.5C(e)$, there must be another edge connected to v_j with current flowing into v_j and therefore after edge cutting there is still a current path from source to v_j. Since all edge currents and resistances except for edge e are not changed, and the paths from source to every node exist, the Elmore delay of every node in the original RC network are not changed.

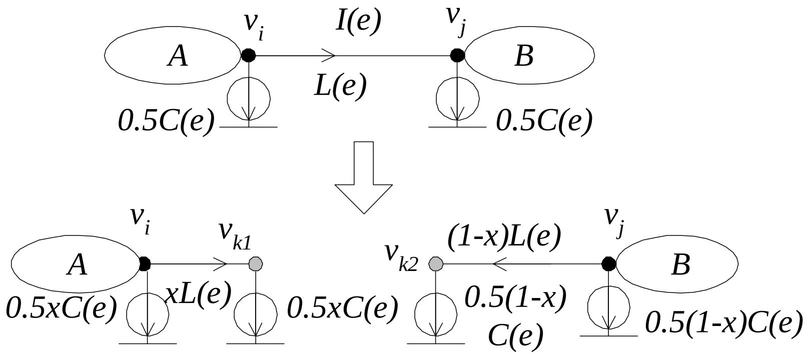


Fig. 31. Edge cutting. If $0 \leq I(e) \leq 0.5C(e)$, edge $e = (v_i, v_j)$ is cut into two edges (v_i, v_{k1}) and (v_{k2}, v_j), such that the Elmore delay for every node in the entire circuit is unchanged. A is the part of the circuit adjacent to v_i, and B is the part of the circuit adjacent to v_j. Symbol ⏀ represents a current source.

After edge cutting, we have

$$\begin{aligned} D(v_{k1}) &= D(v_i) + 0.5x^2 R(e)C(e), \\ D(v_{k2}) &= D(v_j) + 0.5(1-x)^2 R(e)C(e). \end{aligned}$$

Before edge cutting, $D(v_j) = D(v_i) + R(e) \cdot I(e)$. Since $D(v_j)$ is not changed after edge cutting, we have

$$\begin{aligned} D(v_{k2}) &= D(v_i) + R(e)(x - 0.5)C(e) + 0.5(1-x)^2 R(e)C(e) \\ &= D(v_i) + 0.5x^2 R(e)C(e) = D(v_{k1}). \end{aligned}$$

$\square$

Since after edge cutting, the initial current flowing from v_i to v_j is replaced by the current from v_j to v_{k2}, it is easy to see that the number of incoming edges of v_j is reduced by 1.

Node Splitting: For an edge $e = (v_i, v_j)$, if $0.5C(e) < I(e) \leq 0.5C(e) + C(v_j)$, then we can split node v_j to two nodes v_{j1} and v_{j2}. The sink capacitance of v_{j1} is $I(e) - 0.5C(e)$, and the sink capacitance of v_{j2} is $C(v_j) - (I(e) - 0.5C(e))$. The edge e is changed to $e = (v_i, v_{j1})$. Other connections are not changed. See Fig. 32.

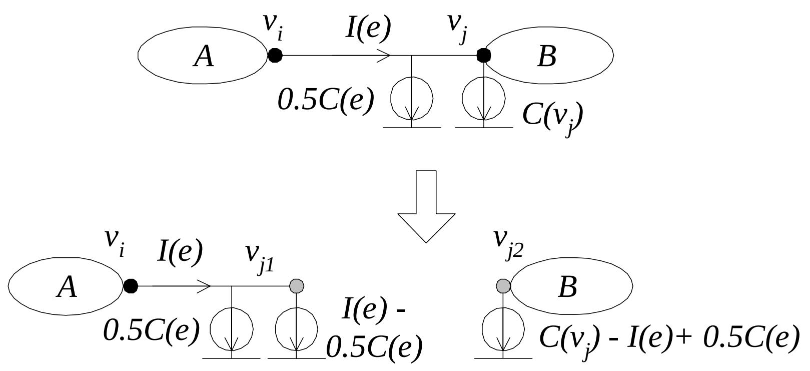


Fig. 32. Node splitting. If $0.5C(e) < I(e) \leq 0.5C(e) + C(v_j)$, node v_j is splitted to two nodes v_{j1} and v_{j2}, such that the Elmore delay at every node in the entire circuit is unchanged. A is the part of the circuit adjacent to node v_i, and B is the part of the circuit adjacent to node v_j.

Lemma III.3. *Node splitting keeps the Elmore delay at all nodes in the original RC network unchanged and* $D(v_{j1}) = D(v_{j2}) = D(v_j)$.

Proof. From Fig. 32, in the original circuit the current flowing from A (including another half edge capacitance 0.5C(e))to node v_i is $I(e)$, the current flowing from B to v_j is $C(v_j)+0.5C(e)-I(e)$. After node splitting, the current flowing from A to node v_i and the current flowing from B to node v_j are unchanged. Similar to the proof in Lemma III.2, there must be another edge connected to v_j with current flowing into v_j and the delay of every node in the original RC network is the same. Therefore, after node splitting we have $D(v_{j2}) = D(v_j)$ and $D(v_{j1}) = D(v_i) + R(e)I(e) = D(v_j)$. □

It is obvious that after node splitting, node v_{j2} has one less incoming edge than the original node v_j.

Edge Splitting: For an edge $e = (v_i, v_j)$, if $I(e) > 0.5C(e) + C(v_j)$ and the number of incoming edges of v_j is greater than 1, we will split v_j and outgoing edge(s) of v_j as follows. Let $e_1, e_2, \dots, e_d$ be adjacent edges of v_j, with the first q edges being incoming edges (assuming $e = e_1$) and the rest being out-going edges. It is easy to see that there must exist an edge $e_s = (v_j, v_k)$, $q < s \leq d$, such that

$$\begin{aligned}\sum_{l=q+1}^{s-1} (I(e_l) + 0.5C(e_l)) &< I(e) - 0.5C(e) - C(v_j) \\ &\leq \sum_{l=q+1}^{s} (I(e_l) + 0.5C(e_l)).\end{aligned}$$

We then split node v_j to two nodes v_{j1} and v_{j2}. The sink capacitance of v_{j1} is $C(v_j)$, and the sink capacitance of v_{j2} is 0. Then for all edges originally connected to v_j, we make edges $e_1(e), e_{q+1}, \dots, e_{s-1}$ connected to v_{j1}, edges $e_2, \dots, e_q, e_{s+1}, \dots, e_d$ connected to v_{j2}, and split edge e_s into two edges e_{s1} and e_{s2}, with $e_{s1} = (v_{j1}, v_k)$ and $e_{s2} = (v_{j2}, v_k)$. The width of all edges except for e_{s1} and e_{s2} is the same, while the width of edge e_{s1} is $b \cdot W(e_s)$ and the width of edge e_{s2} is $(1-b)W(e_s)$, where

$$b = \frac{I(e) - 0.5C(e) - C(v_j) - \sum_{l=q+1}^{s-1}(I(e_l) + 0.5C(e_l))}{I(e_s) + 0.5C(e_s)}.$$

The circuits before and after edge splitting is shown in Fig. 33.

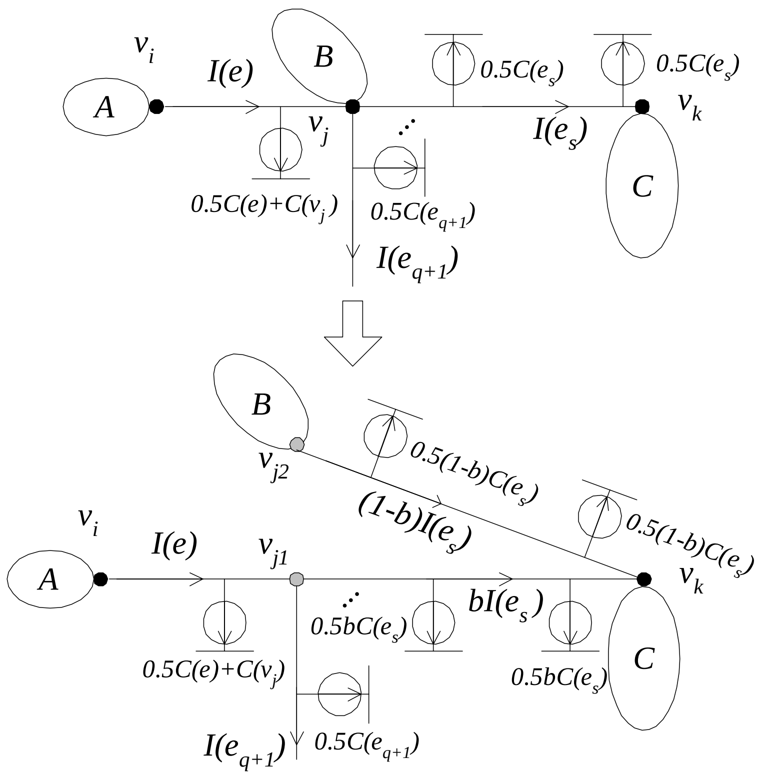


Fig. 33. Edge splitting. If $I(e) > 0.5C(e)+C(v_j)$, edge $e_s = (v_j, v_k)$ is splitted into two edges $e_{s1} = (v_{j1}, v_k)$ and $e_{s2} = (v_{j2}, v_k)$. A is the part of the circuit adjacent to node v_i. B is the part of the circuit adjacent to node v_j including edges $e_2, \ldots, e_q, e_{s+1}, \ldots, e_d$. C is the part of the circuit adjacent node v_k.

Lemma III.4. *Edge splitting keeps the Elmore delay at all nodes in the original RC network unchanged and* $D(v_{j1}) = D(v_{j2}) = D(v_j)$.

Proof. From Fig. 33, in the original circuit the current flowing from A (including another half edge capacitance $0.5C(e)$) to node v_i is $I(e)$, the current flowing from B

(including all half capacitances of edges connected to v_j except for edge e and e_s) to v_j is

$$I(B) = \sum_{l=q+1}^{s} (I(e_l) + 0.5C(e_l)) + 0.5C(e) + C(v_j) - I(e).$$

The current flowing from C to v_k is $0.5C(e_s) - I(e_s)$. After edge splitting, it is easy to see the current flowing from A to node v_i is unchanged and the current flowing from region C to node v_k is unchanged. The current $I(B)$ flowing from B to v_{j2} is

$$\begin{aligned} I(B) &= (1-b)(I(e_s) + 0.5C(e_s)) \\ &= I(e_s) + 0.5C(e_s) - (I(e) - 0.5C(e) - C(v_j) \\ &\quad - \sum_{l=q+1}^{s-1} (I(e_l) + 0.5C(e_l))) \\ &= \sum_{l=q+1}^{s} (I(e_l) + 0.5C(e_l)) + 0.5C(e) + C(v_j) - I(e). \end{aligned}$$

Since the current distribution is not changed, the Elmore delay of every node in the original RC network is not changed. Therefore, after edge cutting we have $D(v_{j2}) = D(v_j)$. For v_{j1}, $D(v_{j1}) = D(v_i) + R(e)I(e) = D(v_j)$. □

It is easy to see the node v_{j2} has one less incoming edge than the original node v_j, node v_{j1} has only one incoming edge. Node v_k has one more incoming edge than before.

Lemma III.5. *For any node with more than one incoming edge, we can use edge cutting, node splitting and edge splitting to reduce its number of incoming edges to be 1. Furthermore, we can reduce the number of incoming edges of every node to be one in $O(|E|^2)$ time.*

Proof. For any node with more than one incoming edge, consider one arbitrary incoming edge e. Its current $I(e)$ must fall into one of the following intervals:

$0 \leq I(e) \leq 0.5C(e)$, $0.5C(e) < I(e) \leq 0.5C(e) + C(v_j)$ or $I(e) > 0.5C(e) + C(v_j)$, and the corresponding operation will be performed. From previous analysis, we know that the incoming edges of this node will only decrease with these three operations. By checking all incoming edges of this node, we can reduce its number of incoming edges to be 1. It was mentioned that after edge splitting the number of incoming edges of node v_k will increase by 1. Therefore, we need to iteratively check every node and perform these three operations. In each iteration, the complexity is $O(|E|)$. Since for any node, the total number of incoming edges in all iterations is upper bounded by $|E|$ (the worst case is that all edges connect to this node and the currents are all flowing into this node), the maximum number of iterations is $|E|$. Therefore, we can reduce the incoming edge(s) of every node to be one in $O(|E|^2)$ time. $\square$

It should be pointed out that Lemma III.2 to III.5 are correct for any layer assignment.

Before decomposition, we also need to know the Elmore delay of every node in original non-tree RC networks. There are some methods that can get Elmore delay in non-tree RC networks efficiently instead of doing LU factorization directly. A tree/link partition method is introduced in [61]. The main idea is to remove edges (called links in the paper) in the network iteratively until there is a spanning tree left (note that this tree does not have the same Elmore delay as original topology). Then the Elmore delays of the original non-tree network are computed by adding each link back. The time complexity is m^2n, where m is the number of removed links and n is the number of nodes. For planar graphs, $m = O(n)$. For the non-tree topologies generated by adding links on a tree topology [3, 4], $m = O(1)$ and is much less than n. Then it is very efficient in terms of speed and memory than LU factorization. In [58], a relaxation method is introduced and the running time depends on the required accuracy. Even for a planar graph, since the G matrix is sparse, there are also some efficient numeric approaches to get the Elmore delay. Thus, we have the following decomposition algorithm and theorem.

```
Algorithm Decomposition
  Input: Non-tree RC network $G = (V, E)$.
  Output: Tree RC network $G' = (V', E')$ with source $v_1$.
1 Compute the Elmore delay for all nodes in $V$;
2 while there is a node with more than one incoming edge do
3     for every edge $e = (v_i, v_j)$ in $E$ such that $I(e) \geq 0$ and the number of
      incoming edges of $v_j$ is greater than 1 do
4         if $0 \leq I(e) \leq 0.5C(e)$ then
5             Edge Cutting;
6         else if $0.5C(e) < I(e) \leq 0.5C(e) + C(v_j)$ then
7             Node Splitting;
8         else if $I(e) > 0.5C(e) + C(v_j)$ then
9             Edge Splitting;
10        end
11    end
12 end
13 return the new graph;
```

Theorem III.6. *Given any non-tree topology, we can decompose this topology into an tree topology through edge cutting, node splitting and edge splitting such that the Elmore delay of every point remains the same and all node capacitances are greater than zero. The time complexity is $O(n^3)$ for a planar graph G.*

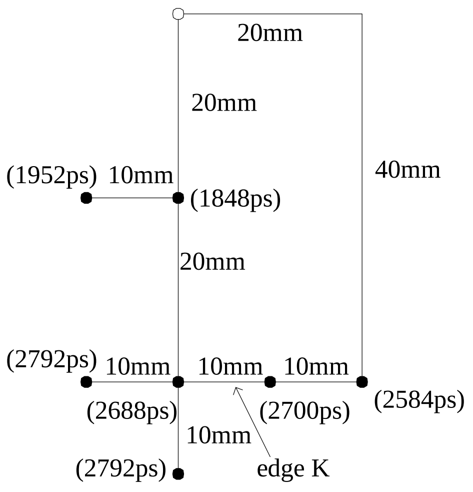
One example of our decomposition algorithm is shown in Fig. 34. The example circuit is net 1 used in [3] and the original topology is shown in Fig. 34(a). The number associated with each edge is the edge length and the number in the bracket is the Elmore delay of each node. We use the same interconnect and gate parameters: driving resistance $R(d) = 25\Omega$, unit resistance $r_0 = 0.008\Omega/\mu m$, unit capacitance $c_0 = 0.06fF/\mu m$, loading capacitance $C_s = 1000fF$. In this example, only edge k in original topology needs to be checked after checking all edge currents. Here edge cutting is performed. The new tree topology after decomposition is shown in Fig. 34(b) and the Elmore delay at all original nodes and new generated nodes are also shown.

D. Wire Sizing and Merging

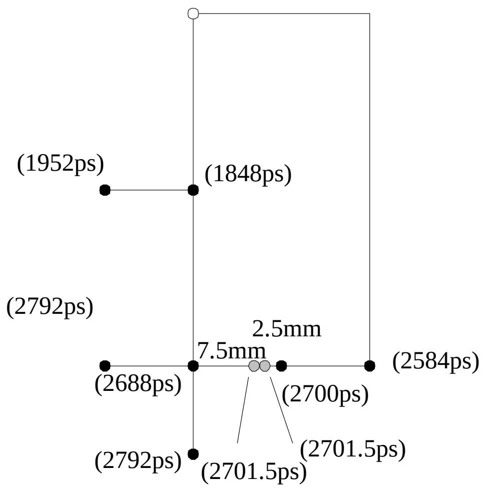
After the decomposition, the network is a tree and all node capacitance is non-negative. Therefore we can use existing wire sizing methods for the tree topology (or we call tree-targeted wire sizing methods) to optimize various objectives, such as minimizing weighted delay T, minimizing maximum delay, maximizing source slack $Q(v_1)$, minimizing area A under delay constraints, or achieve zero skew [42, 39, 44, 48, 49].

After we get the optimal wire sizing solution for the tree topology, we need to connect the edges or nodes to restore the original non-tree topology with a sized solution. The merging process could change the delay at every node. If the design targets are not satisfied, iterative process of tree decomposition (since the delay is changed, the decomposition could be different), wire sizing and merging need to be performed until the constraints are satisfied.

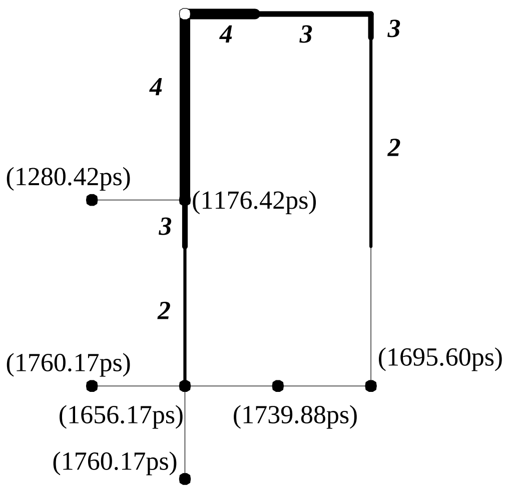
For the example circuit shown in Fig. 34, the non-tree topology after wire sizing and merging process is shown in Fig. 34(c). The TRIO software package is used on wire sizing for the tree topology [39, 44, 62]. We can see that the delay of the new



(a) Original Non-tree Topology



(b) New Tree Topology



(c) Non-tree Topology after wire sizing

Fig. 34. Example of our algorithm applied to a non-tree topology. The white node is source, black nodes are sinks, and grey nodes are new sink nodes generated from splitting an edge.

circuit is 37% less than original circuit.

Though generally several iterations may be needed to satisfy the design objective, Theorem III.7 to III.10 prove that for certain topologies and objectives, the merging does not change the solution quality, which means one iteration may be enough.

Theorem III.7. *After tree decomposition and wire sizing, if the delays of every pair of merging points are equal, then after merging the delay at every node remains unchanged.*

Proof. The proof is similar to the analysis in [61, 60], in which it is proved that the addition of a zero resistor between two nodes with same voltage does not change the voltages of all circuit nodes. □

Theorem III.7 implies that after tree decomposition, if the tree-targeted wire sizing algorithms can generate a solution, which satisfies the design constraints (delay or area) and at the same time guarantees zero skew between every pair of merging points, then the solution after merging also satisfies the design constraints. One application of Theorem III.7 is to use tree-targeted zero skew wire sizing algorithm to achieve a zero skew solution for clock network built via the tree+link method by Rajaram *et al* [4] under delay or area constraints. The main procedure of the tree+link method in [4] is as follows. First, an initial zero skew clock tree is constructed, and the links between sinks are identified. Then link capacitors are added to the tree and tree is tuned to get zero skew. Finally link resistors is added back and it can be proved the delay of every point does not change and new network still has zero skew. The method is mainly to reduce the clock skew under process variation. For tree-targeted zero skew wire sizing problems, some algorithms, such as [49, 48], can size the wire of a given zero skew clock tree under delay or area constraints while keeping zero skew after sizing.

Theorem III.8. *For clock networks built by the tree+link method in [4], if after tree decomposition the tree-targeted zero skew wire sizing algorithm generates a sized tree satisfying the delay or area constraints, then after merging the new network also has zero skew and satisfies the delay or area constraints.*

Proof. Since every sink of the clock network built by the tree+link method in [4] has the same delay, from Lemma III.2, by edge cutting at the middle of each link edge, the network can be easily decomposed to a new tree and two new nodes are generated at the middle of each link. Then for each sink in the original clock network to which a link is connected, the sink capacitance is increased by half of the link edge capacitance and two link edges (generated from cutting the original link) are omitted. Then the tree-targeted zero skew wire sizing algorithm is used to size the new tree under delay or area constraints. Since the delay between every sink in the tree is still same after wire sizing, every pair of two nodes created by edge cutting on a link still have the same delay, which equals to the sink delay plus half of interconnect delay of that link. From Theorem III.7, the merging process does not affect the delay of every node. Therefore, the new network after merging has the same delay and area with the network before merging. $\square$

When the merging points located in some specific positions, it is also possible to optimize the circuit in one iteration.

Theorem III.9. *Given D_{target} and the objective being $D(v_i) < D_{target}$ for every node v_i, for a single RC loop circuit, where the degree of every node is 2, if after tree decomposition the wire sizing solution for the tree topology satisfies the objective, then after merging the solution for the non-tree topology still satisfies the objective.*

Proof. It is easy to see that the RC loop circuit can be decomposed into a tree topology by only using node splitting or edge cutting, where two new points v_{j1}

and v_{j2} are created (for node splitting, original node v_j disappears). Let $D_t(v_i)$ and $D_m(v_i)$ represent the Elmore delay of node v_i in the tree topology before merging and non-tree topology after merging, respectively. If the wire sizing solution for the tree topology satisfies the constraint, which means $D_t(v_{j1}) < D_{target}$ and $D_t(v_{j2}) < D_{target}$. Then after merging, with the analysis of similar to [61], the delay at every node v_i changes to $D_m(v_i) = D_t(v_i) - (D_t(v_{j1}) - D_t(v_{j2})) \cdot r_i/(r_{j1} - r_{j2})$, where $D_t(v_i)$ is the delay of every node on tree topology before merging, r_i, r_{j1}, and r_{j2} are equal to the Elmore delay at i, $j1$ and $j2$ on tree topology, respectively, when $C_{j1} = 1$, $C_{j2} = -1$ and the other node capacitances are zero. In our case, $r_{j1} - r_{j2}$ is the whole loop resistance. The absolute value of r_i is the resistance from source to i, and it is positive (negative) when v_i locates on the path from source to $j1$ ($j2$) in the tree topology. Then it is not hard to prove that, no matter where v_i locates, $D_m(v_i) < D_t(v_{j1})$ or $D_m(v_i) < D_t(v_{j2})$. Therefore, $D_m(v_i) < D_{target}$. □

Theorem III.10. *Given D_{target} and the objective being $D(v_i) < D_{target}$ for every node v_i, if a non-tree circuit can be decomposed into a tree topology by only using node splitting and edge cutting, and each pair of new created points only locates in a single loop, where all nodes have degree 2 except for the jointing point of this loop to other parts of circuit, then if the wire sizing solution for this tree topology satisfies the objective, after merging the solution for the non-tree topology still satisfies the objective.*

Proof. Since every pair of merging points is in a single loop, then with the analysis similar to [61], merging of these two points does not affect the delay at the other part of the circuit out of the loop. Also, similar to the proof of Theorem III.9, the delay of the node inside of a single loop does not increase. Therefore, after merging the delay of every node in the non-tree topology still satisfies the objective. □

The whole algorithm Non-tree Topology Wire Sizing (NTWS) is as follows:

```
Algorithm $NTWS$
  Input: Non-tree RC network $G = (V, E)$.
  Output: Wire sizing solution for the edges in $E$..
1 Decomposition;
2 Compute the optimal wire sizing solution for the tree based on different
  objectives;
3 Compute the Elmore delay for all nodes in $V$;
4 if the merged non-tree topology does not meet the objective then
5     go to 1 with the current width assignment;
6 end
7 return wire sizing solution;
```

Theorem III.11. *NTWS can find the wire sizing in time $O(|V|^3 + S(G'))$ per iteration, where $S(G')$ is the time for wire sizing of RC tree G'.*

E. Experimental Result

All experimental results are run on a Sun SPARC workstations with 400 MHz and 2 GB memory. The running time of decomposition and merging for each circuit is less than 0.1 second.

1. Tree Decomposition Accuracy

We first run SPICE to verify the accuracy of our tree decomposition method. We tested the non-tree topologies r1, r2 and r3 generated by the algorithm in [4] with all snaking ignored to create nonzero skew. The technology parameters are as follows:

driving resistance $R(v_1) = 100\Omega$, unit wire resistance $r_0 = 0.03\Omega/\mu m$, unit wire capacitance $c_0 = 0.2fF/\mu m$, sink capacitance $C(s_i)$ ranges from $3.1fF$ to $8.4fF$. The maximum delay error between the original non-tree topology and the decomposed tree under SPICE simulation for r1, r2 and r3 are 0.05%, 0.11%, and 0.08% respectively. Fig. 35 also shows the delay for all sinks in r3 before and after tree decomposition. It is clear that our tree decomposition method is very accurate even under SPICE simulation.

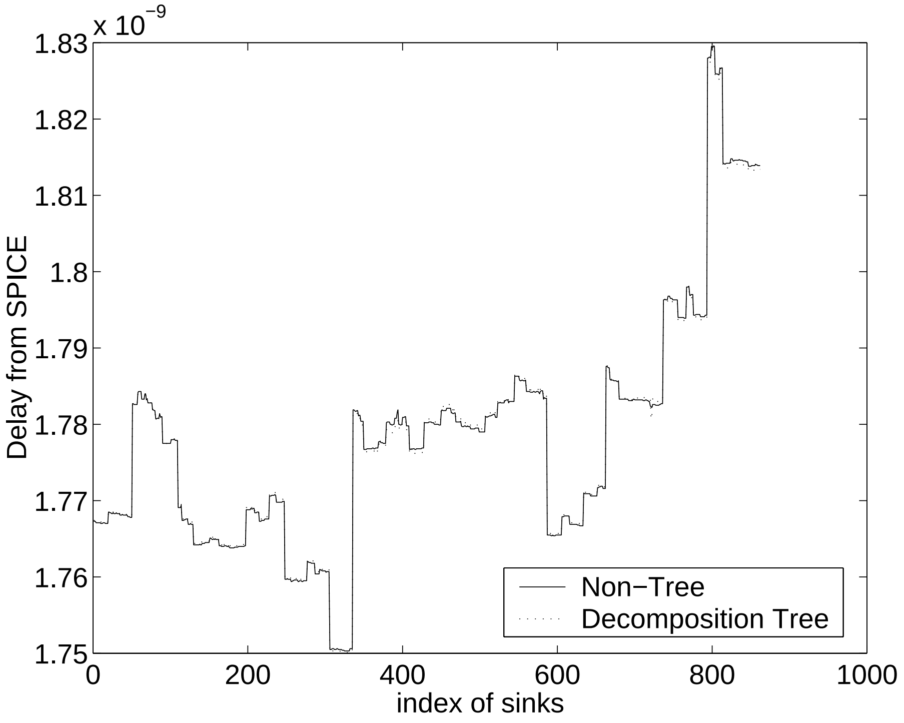


Fig. 35. SPICE delay for all sinks in original non-tree topology and decomposed tree tree topology

2. Delay Reduction via Wire Sizing

The second set of experiments are done for the non-tree topologies 1, 2 and 4 used in [3]. The technology parameters are the same as in [53], where driving resistance

$R(v_1) = 25\Omega$, unit resistance and capacitance are $0.008\Omega/\mu m$, and $0.06fF/\mu m$, and sink capacitance $C(s_i) = 1000fF$. We only run one iteration of NTWS and we use the TRIO package for tree wire sizing. Simulation results are shown in Table XV. In the table, total area is equal to the sum of link area and average wire width is defined as the ratio between the area of wire sized region and the length of wire sized region. Since the method in [3] only sizes the links, we only show the average wire width in the link region, which is equal to the area of links divided by the length of links. Smaller average wire width implies smaller coupling capacitance effect. Other nets in [3] are not simulated because the length of link is longer than Manhattan distance and the detail topology parameter was not published in the paper. In Table XV, LS represents the non-uniform link sizing method in [3] and the data is got from [3]. NEW represents our wire sizing method. The objective is set to minimize the weighted delay (every sink has the same weight) under the area constraints, which is set by limiting the width upper bound for each wire. Both LS and NEW use Elmore delay model. $SPICE$ shows the SPICE simulation results of our wire sizing method to verify the fidelity of Elmore delay model. The maximal delay, skew, average wire width and the total area shown in the table are normalized to minimum sizing solution.

With Elmore delay model, compared with non-uniform link sizing method [3], our method get 2% to 17% delay reduction, 14% to 30% area reduction, and 49% to 65% average wire width reduction. With SPICE simulation, our method achieve 15% to 25% delay reduction, and 20% to 35% skew reduction compared with minimum size solution. It also shows that there is strong correlation between SPICE results and the Elmore delay results. Since the wire sizing assignments are not give in [3], we can not show SPICE results for link sizing method in [3].

From Table XV, our algorithm can also get the skew reduction about 20% compared with the minimum sizing solution. Note that the skew reduction compared with the link sizing method is not obvious since the sizing algorithm that we use does not target on skew reduction. Also, we did some experiments without area constraints and we can get up to 28% delay reduction with 15% more area compared with non-uniform link sizing method [3].

Table XV. Normalized maximum delay (MD), maximum skew (MS), average wire width (AW) and total area (TA) results for 3 clock networks in [3]. *LS* represents the network built by [3], *NEW* represents the network built by our method, and *SPICE* represents SPICE simulation results of *NEW*.

Case	**Method**	**MD**	**MS**	**AW**	**TA**
1	*LS* [3]	0.878	0.724	3.017	1.807
	NEW	0.731	0.647	1.034	1.547
	SPICE	0.745	0.641	1.034	1.547
2	*LS* [3]	0.856	0.653	3.500	1.789
	NEW	0.834	0.857	1.472	1.268
	SPICE	0.847	0.798	1.472	1.268
4	*LS* [3]	0.768	0.639	3.825	2.076
	NEW	0.719	0.665	1.932	1.719
	SPICE	0.750	0.705	1.932	1.719

3. Zero Skew via Wire Sizing for Clock Network

In this set of experiments, we use ClockTune package [49] to construct the original clock tree r1, r2, r3 and r4. Then we generate the non-tree zero skew clock network by adding links with the minimum weight matching based selection in [4]. The number of links added for r1, r2 and r3 and r4 are 13, 4, and 27, and 9, respectively. After tree decomposition, we use ClockTune to size the wires with 256 sampling points and get zero skew solution by choosing min-delay embedding. The new network after merging still has the zero skew in terms of Elmore delay at nominal case.

It is stated in [48] that the wire sizing can also help to improve the skew variability under variations for tree structure. We perform the following experiments to test the wire sizing effect on skew variability for non-tree networks. The variation factors considered in the experiments include the clock driver gate length, the width of each wire segment in the network and each sink load capacitance. Following [4], each variable has $\pm 15\%$ variation following a normal distribution. Note that in this experiment we are using 0 correlation model for variation simulation, which assumes the width of each wire segment is an independent variable. In real applications, if two wires are on same layers they may be highly correlated and if they are closer, they are even more correlated. Even though our model is an extreme case, this assumption tends to give the worst case bound for skew variation. Also, we believe that the effectiveness of our method is still valid under the other variation models.

In the experiments, the skew variations, total wire capacitances are compared among clock trees, clock network including links, sized clock network. For each network, a Monte Carlo SPICE simulation of 500 trials is performed to obtain the maximum skew variation (MSV) and the standard deviation (SD) of skew variations. The size of benchmark circuits, nominal delay, maximum skew variations and total wire capacitance of original clock tree, tree+link [4] and our method are given in Table XVI. Compared with method in [4], our method results in 16% to 48% delay reduction, 54% to 66% maximum skew variation reduction and 57% to 65% standard deviation reduction with 15%-27% more wire capacitances. Compared with the leave leaf level meshes results reported in [4], our results even have much less total wire capacitances with better skew and standard deviation reduction.

Table XVI. Nominal delay (ND), maximum skew variation (MSV), standard deviation (SD) and total wire capacitances (WC) for tree, tree+link [4] and clock network built by our method (New).

Case	**Method**	**ND (ns)**	**MSV (ns)**	**SD (ns)**	**WC (pF)**
r1	Tree	1.774	0.346	0.068	30.72
	Link [4]	1.804	0.218	0.042	33.02
	New	1.518	0.099	0.018	42.10
r2	Tree	4.299	1.029	0.167	61.14
	Link [4]	4.304	0.481	0.085	61.60
	New	2.960	0.205	0.030	76.14
r3	Tree	6.165	1.422	0.263	79.22
	Link [4]	6.477	1.146	0.187	86.69
	New	4.125	0.391	0.069	104.34
r4	Tree	16.200	4.472	0.789	161.35
	Link [4]	16.339	2.034	0.447	162.62
	New	8.424	0.769	0.169	187.14

CHAPTER IV

CONCLUSIONS AND FUTURE WORK

We present several efficient algorithms for buffer insertion and wire sizing problems, which are two essential interconnect optimization techniques. For max-slack buffer insertion problem, three optimal algorithms for basic Van Ginneken's algorithm, large buffer libraries, and nets with small sinks have been proposed. The algorithms are so efficient that large industrial nets can be optimized in few seconds. For min-cost buffer insertion problem, we first prove that it is NP-complete. Then we propose speedup techniques to accelerate the min-cost buffer insertion algorithm even for nets of medium or small size. Extensive experiments on industrial designs show that dramatically speedup is achieved through these techniques. For the wire sizing problem, we present a new methodology to size the wires in circuits with non-tree topology. The main idea is to decompose the non-tree topology to a tree topology and then get the optimal size solution for the tree. By using edge cutting, node splitting and edge splitting, we can transform any non-tree topology circuit to a new tree circuit while keeping the Elmore delay at all nodes in the original non-tree topology unchanged. This approach offer a new way to study the optimization problems of non-tree topologies since well-developed optimization algorithms used on tree topologies can be applied.

Though several different algorithms have been proposed, they are inside connected and one algorithm is generally motivated from the technique originally proposed for the other algorithm. For example, the $O(mn)$ algorithm is motivated from the convex pruning proposed in $O(bn^2)$ algorithm and the data structure idea from $O(n \log^2 n)$ algorithm. Our new algorithms not only speed up the classical algorithms, also explore the instinct of the problem itself and find the implicit property, such as the candidates that generate new buffered candidates must lie on the convex hull of (Q, C). Therefore, many techniques proposed in this work can be used to more applications. For example, predictive pruning and convex pruning proposed for max-slack buffer insertion can be used for min-cost problem also, and they have potential to be used on some other buffer insertion problem, such as tree construction and high order delay models.

Following this work, one future direction could be extend the fast buffer insertion algorithm to buffered tree construction. All the algorithms presented in this work ares based on static topologies. In a popular two step approach, a good timing-driven tree, such as C-tree [63], is first constructed followed by buffer insertion. This process is repeated until the timing is achieved. This method is generally faster than performing tree construction and buffer insertion simultaneously, but the performance may be degraded. Our algorithms strongly speed up the two step methodology to let designer explore more potential trees. However, it is worthwhile to study the fast algorithms for simultaneous tree construction and buffer insertion to get both better running time and performance. Some of techniques proposed in this work have potential to be applied, such as predictive pruning, convex pruning and simple list data structure used in Section E. Also, one could consider buffer insertion and placement together. It may happen that after the placement, the part of circuits need buffers are too congested and only few buffers or even no buffers can be inserted, and in turn the timing is very bad. Such scenario could be alleviated by leaving buffer spaces at the placement stages. There are some works on this topic but still no complete solutions yet.

In stead of delay optimization, buffer insertion can also fix slew violation (electric violation), and in general, designers always first fix electric violation before any timing optimization. Fast algorithms for slew based buffer insertion is also very essential and we are working on this problem. Other extensions also include leakage power-aware buffer insertion, buffer insertion considering multiple V_{dd}s and V_{vth}s. Such problems are very important in the nano-meter era as the leakage power becomes dominant over the dynamic power.

In the nano-meter era, many variation effects, such as intra-die and inter-die process variations, power/ground noise, and temperature variations, start to manifest strongly. To address this problem, one way to address this issue is to study the statistical algorithms and methodologies instead of traditional deterministic methods. Recently, many traditional problems have been re-visited to study the statistical approaches. It is interesting to study the buffer insertion algorithms in the statistical environment, which may be applied in the noise reduction or the optimization with process variation or unpredictable design information (some early work has been shown in [64]). The work of our wire sizing approach, which has been shown effective to control the skew variations, will also be explored further to directly target on variation reduction for clock network. In such case, the decomposition of tree may not need to keep the same Elmore delay at every node and more efficient algorithms may exist. Combined with clock routing and buffer insertion, a new clock network with low power and variation control will be studied.

REFERENCES

[1] 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 Journal of Solid-State Circuits*, vol. 31, no. 3, pp. 437–447, March 1996.

[2] W. Shi, Z. Li, and C. J. Alpert, "Complexity analysis and speedup techniques for optimal buffer insertion with minimum cost," in *Proceedings of the Conference on Asia South Pacific Design Automation*, Yokohama, Japan, January 2004, pp. 609–614, IEEE Press.

[3] T. Xue and E. S. Kuh, "Post routing performance optimization via multi-link insertion and non-uniform wiresizing," in *Proceedings of the International Conference on Computer-Aided Design*, San Jose, California, November 1995, pp. 575–580, IEEE Computer Society Press.

[4] A. Rajaram, J. Hu, and R. Mahapatra, "Reducing clock skew variability via cross links," in *Proceedings of the 41st Conference on Design Automation*, San Diego, California, June 2004, pp. 18–23, ACM Press.

[5] P. Saxena, N. Menezes, P. Cocchini, and D. A. Kirkpatrick, "Repeater scaling and its impact on CAD," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 4, pp. 451–463, April 2004.

[6] J. Cong, L. He, C.-K. Koh, and P. H. Madden, "Performance optimization of VLSI interconnect layout," *The VLSI Journal of Integration*, vol. 21, pp. 1–94, November 1996.

[7] J. Cong, "An interconnect-centric design flow for nanometer technologies," *Proceedings of IEEE*, vol. 89, pp. 505–528, April 2001.

[8] J. Cong and Z. Pan, "Interconnect performance estimation models for design planning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 6, pp. 739–752, June 2001.

[9] P. J. Osler, "Placement driven synthesis case studies on two sets of two chips: Hierarchical and Flat," in *Proceedings of International Symposium on Physical Design*, Phoenix, Arizona, April 2004, pp. 190–197, ACM Press.

[10] L. P. P. P. van Ginneken, "Buffer placement in distributed RC-tree network for minimal Elmore delay," in *Proceedings of International Symposium on Circuits and Systems*, New Orleans, Louisiana, May 1990, pp. 865–868.

[11] W. C. Elmore, "The transient response of damped linear networks with particular regard to wideband amplifiers," *Journal of Applied Physics*, vol. 19, no. 1, pp. 55–63, January 1948.

[12] C. J. Alpert and A. Devgan, "Wire segmenting for improved buffer insertion," in *Proceedings of the 34th Conference on Design Automation*, Anaheim, California, June 1997, pp. 588–593, ACM Press.

[13] C. J. Alpert, R. G. Gandham, J. L. Neves, and S. T. Quay, "Buffer library selection," in *Proceedings of International Conference on Computer Design*, Austin, Texas, September 2000, pp. 221–226.

[14] C. J. Alpert, A. Devgan, and S. T. Quay, "Buffer insertion for noise and delay optimization," in *Proceedings of the 35th Conference on Design Automation*, San Francisco, California, June 1998, pp. 362–367, ACM Press.

[15] C. J. Alpert, A. Devgan, and S. T. Quay, "Buffer insertion with accurate gate and interconnect delay computation," in *Proceedings of the 36th Conference*

on Design Automation, New Orleans, Louisiana, June 1999, pp. 479–484, ACM Press.

[16] P. Cocchini, "Concurrent flip-flop and repeater insertion for high performance integrated circuits," in *Proceedings of the International Conference on Computer-Aided Design*, San Jose, California, November 2002, pp. 268–273, ACM Press.

[17] C. L. Berman, J. L. Carter, and K. F. Day, "The fanout problem: From theory to practice," in *Proceedings of the Decennial Caltech Conference on VLSI on Advanced Research in VLSI*, Cambridge, Massachusetts, June 1989, pp. 69–99, MIT Press.

[18] K. J. Singh and A. Sangiovanni-Vincentelli, "A heuristic algorithm for the fanout problem," in *Proceedings of the 27th Conference on Design Automation*, Orlando, Florida, June 1990, pp. 357–360, ACM Press.

[19] S. Lin and M. Marek-Sadowska, "A fast and efficient algorithm for determining fanout tree in large networks," in *Proceedings of the Conference on European Design Automation*, Amsterdam, The Netherlands, Febrary 1991, pp. 539–544, IEEE Computer Society Press.

[20] T. Okamoto and J. Cong, "Buffered Steiner tree construction with wire sizing for interconnect layout optimization," in *Proceedings of the International Conference on Computer-Aided Design*, San Jose, California, November 1996, pp. 44–49, IEEE Computer Society Press.

[21] M. Kang, W. W.-M. Dai, T. Dillinger, and D. Lapotin, "Delay bounded buffered tree construction for timing driven floorplanning," in *Proceedings of the International Conference on Computer-Aided Design*, San Jose, California, November 1997, pp. 707–712, IEEE Computer Society Press.

[22] H. Zhou, D. F. Wong, I. M. Liu, and A. Aziz, "Simultaneous routing and buffer insertion with restrictions on buffer locations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 7, pp. 819–824, July 2000.

[23] S. Hassoun, C. J. Alpert, and M. Thiagarajan, "Optimal buffered routing path constructions for single and multiple clock domain systems," in *Proceedings of the International Conference on Computer-Aided Design*, San Jose, California, November 2002, pp. 247–253, ACM Press.

[24] M. Hrkic and J. Lillis, "S-tree: a technique for buffered routing tree synthesis," in *Proceedings of the 39th Conference on Design Automation*, New Orleans, Louisiana, June 2002, pp. 578–583, ACM Press.

[25] M. Hrkic and J. Lillis, "Buffer tree synthesis with consideration of temporal locality, sink polarity requirements, solution cost and blockages," in *Proceedings of the International Symposium on Physical Design*, San Diego, California, April 2002, pp. 98–103, ACM Press.

[26] M. Hrkic, "Tree optimization and synthesis techniques with application in automated design of integrated circuits," Ph.D. dissertation, University of Illinois at Chicago, Chicago, Illinois, 2004.

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

[28] C. C. N. Chu and D. F. Wong, "A quadratic programming approach to simultaneous buffer insertion/sizing and wire sizing," *IEEE Transactions on Computer-*

Aided Design of Integrated Circuits and Systems, vol. 18, no. 6, pp. 787–798, June 1999.

[29] T. H. Corman, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithm*, MIT Press, Cambridge, Massachusetts, 2nd edition, 2001.

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

[31] R. L. Graham, "An efficient algorithm for determining the convex hull of a finite planar set," *Information Processing Letters*, vol. 1, no. 4, pp. 132–133, June 1972.

[32] W. Shi and Z. Li, "An $O(n \log n)$ time algorithm for optimal buffer insertion," in *Proceedings of the 40th Conference on Design Automation*, Anaheim, California, June 2003, pp. 580–585, ACM Press.

[33] W. Shi and Z. Li, "A fast algorithm for optimal buffer insertion," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 6, pp. 879–891, June 2005.

[34] R. Chen and H. Zhou, "A flexible data structure for efficient buffer insertion," in *Proceedings of the International Conference on Computer Design*, San Jose, California, October 2004, pp. 216–221, IEEE Computer Society Press.

[35] Z. Li and W. Shi, "An $O(bn^2)$ time algorithm for optimal buffer insertion with b buffer types," in *Proceedings of the Conference on Design, Automation and Test in Europe*, Munich, Germany, March 2005, pp. 1324–1329, IEEE Computer Society Press.

[36] Z. Li, C. N. Sze, C. J. Alpert, J. Hu, and W. Shi, "Making fast buffer insertion even faster via approximation techniques," in *Proceedings of the Conference on Asia South Pacific Design Automation*. January 2005, pp. 13–18, IEEE Press.

[37] M. R. Garey and D. S. Johnson, *Computers & Intractability: A Guide To The Theory Of NP-Completeness*, W. H. Freeman & Co., New York, New York, 1979.

[38] R. P. Stanley, *Enumerative Combinatorics*, vol. 2, Cambridge University Press, Cambridge, England, 1999.

[39] J. Cong and K.-S. Leung, "Optimal wiresizing under Elmore delay model," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 3, pp. 321–336, March 1995.

[40] S. S. Sapatnekar, "RC interconnect optimization under the Elmore delay model," in *Proceedings of the Conference on Design Automation*, San Diego, California, June 1994, pp. 387–391, ACM Press.

[41] N. Menezes, S. Pullela, F. Dartu, and L. T. Pillage, "RC interconnect synthesis — a moment fitting approach," in *Proceedings of the International Conference on Computer-Aided Design*. November 1994, pp. 418–425, IEEE Computer Society Press.

[42] C.-P. Chen, H. Zhou, and D. F. Wong, "Optimal non-uniform wire-sizing under the Elmore delay model," in *Proceedings of the International Conference on Computer-Aided Design*, San Jose, California, November 1996, pp. 38–43, IEEE Computer Society Press.

[43] C.-P. Chen, C. C. N. Chu, and D. F. Wong, "Fast and exact simultaneous gate

and wire sizing by lagrangian relaxation," in *Proceedings of the International Conference on Computer-Aided Design*, San Jose, California, November 1998, pp. 617–624, ACM Press.

[44] J. Cong and L. He, "Optimal wiresizing for interconnects with multiple sources," *ACM Transactions on Design Automation of Electronic Systems*, vol. 1, no. 4, pp. 478–511, October 1996.

[45] J. Cong and L. He, "An efficient technique for device and interconnect optimization in deep submicron designs," in *Proceedings of the International Symposium on Physical Design*, Monterey, California, April 1998, pp. 45–51, ACM Press.

[46] C. C. N. Chu and D. F. Wong, "Closed form solutions to simultaneous buffer insertion/sizing and wire sizing," *ACM Transactions on Design Automation of Electronic Systems*, vol. 6, no. 3, pp. 343–371, July 2001.

[47] C.-P. Chen, Y.-P. Chen, and D. F. Wong, "Optimal wire-sizing formula under the Elmore delay model," in *Proceedings of the Conference on Design Automation*, Las Vegas, Nevada, June 1996, pp. 487–490, ACM Press.

[48] S. Pullela, N. Menezes, and L. T. Pileggi, "Post-processing of clock trees via wiresizing and buffering for robust design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 6, pp. 691–701, June 1996.

[49] J.-L. Tsai, T.-H. Chen, and C. C.-P. Chen, "ε-Optimal minimum-delay/area zero-skew clock tree wire-sizing in pseudo-polynomial time," in *Proceedings of the International Symposium on Physical Design*, Monterey, California, April 2003, pp. 166–173, ACM Press.

[50] J. P. Fishburn and A. E. Dunlop, "TILOS: a posynomial programming approach to transistor sizing," in *Proceedings of the International Conference on Computer-Aided Design*, November 1985, pp. 326–328.

[51] C. C. N. Chu and D. F. Wong, "Greedy wire-sizing is linear time," in *Proceedings of the International Symposium on Physical Design*, Monterey, California, April 1998, pp. 39–44, ACM Press.

[52] B. A. McCoy and G. Robins, "Non-tree routing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 6, pp. 780–784, June 1995.

[53] T. Xue and E. S. Kuh, "Post routing performance optimization via tapered link insertion and wiresizing," in *Proceedings of the Conference on European Design Automation*, Brighton, England, September 1995, pp. 74–79, IEEE Computer Society Press.

[54] H. Su and S. S. Sapatnekar, "Hybrid structured clock network construction," in *Proceedings of the International Conference on Computer-Aided Design*, San Jose, California, November 2001, pp. 333–336, IEEE Press.

[55] A. B. Kahng, B. Liu, and I. I. Măndoiu, "Non-tree routing for reliability and yield improvement," in *Proceedings of the International Conference on Computer-Aided Design*, San Jose, California, Untied States, November 2002, pp. 260–266, ACM Press.

[56] S. Lin and C. K. Wong, "Process-variation-tolerant clock skew minimization," in *Proceedings of the International Conference on Computer-Aided Design*, San Jose, California, November 1994, pp. 284–288, IEEE Computer Society Press.

[57] L. Vandenberghe, S. Boyd, and A. El Gamal, "Optimal wire and transistor sizing for circuits with non-tree topology," in *Proceedings of the International Conference on Computer-Aided Design*, San Jose, California, November 1997, pp. 252–259, IEEE Computer Society Press.

[58] T.-M. Lin and C. A. Mead, "Signal delay in general RC networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 3, no. 4, pp. 331–349, October 1984.

[59] J. L. Wyatt Jr., "Signal delay in RC mesh networks," *IEEE Transactions on Circuits and Systems*, vol. 32, no. 5, pp. 507–510, May 1985.

[60] T. L. Pillage, R. A. Rohrer, and C. Visweswariah, *Electronic Circuit & System Simulation Methods*, McGraw-Hill Inc., New York, 1995.

[61] P. K. Chan and K. Karplus, "Computing signal delay in general RC networks by tree/link partitioning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 8, pp. 898–902, August 1990.

[62] J. Cong, L. He, C.-K. Koh, D. Z. Pan, and X. Yuan, "UCLA Tree-Repeater-Interconnect-Optimization Package (TRIO)," http://cadlab.cs.ucla.edu/software/_release/trio/htdocs, 1999.

[63] C. J. Alpert, G. Gandham, M. Hrkic, J. Hu, A. B. Kahng, J. Lillis, B. Liu, S. T. Quay, S. S. Sapatnekar, and A. J. Sullivan, "Buffered Steiner trees for difficult instances," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 3–14, January 2002.

[64] V. Khandelwal, A. Davoodi, A. Nanavati, and A. Srivastava, "A probabilistic approach to buffer insertion," in *Proceedings of the International Conference*

on Computer-Aided Design, San Jose, California, November 2003, pp. 560–567, IEEE Computer Society Press.

VITA

Zhuo Li received the B.E. and M.S. degrees in Electrical Engineering from Xi'an JiaoTong University, China, in 1998 and 2001 respectively.

In summer 2004, he worked at the IBM Austin Research Laboratory. His research interests lie in the general area of VLSI CAD with an emphasis on interconnect optimization, routability prediction, clock network synthesis, timing analysis and delay testing.

He received the Applied Materials Fellowship in 2002. He was a substitute member of Xi'an JiaoTong University Debating Team, who won the championship of the Chinese National College Debating Contest and the CCTV Invitational Debating Contest(CCTV is the official national TV station in China) in 1998. He won the championship of Xi'an JiaoTong University Speech Contest in 1996.

The typist for this dissertation was Zhuo Li.