Critical-Trunk-Based Obstacle-Avoiding Rectilinear Steiner Tree Routings and Buffer Insertion for Delay and Slack Optimization

Yen-Hung Lin, *Student Member, IEEE* , Shu-Hsin Chang, Yih-Lang Li, *Member, IEEE*

***Abstract*—For modern designs, delay optimization significantly facilitates success in design closure owing to its more realistic metric than wirelength in routing. Obstacle-avoiding rectilinear Steiner tree (OARST) construction is an essential routing problem. With the trends toward Internet protocol-block-based system-on-chip designs, OARST with buffer insertion has been surveyed to diminish the delay of long wires. Previous works on performance-driven (PD) OARST without and with buffer insertion can only handle small circuits. This paper develops a novel routing algorithm in obstacle-avoiding spanning graph to construct OARST with optimized delay efficiently. The proposed multisource single-target maze routing is first employed to identify the critical trunks, and the critical-trunk-based tree growth mechanism connects the unconnected pins to critical trunks under delay constraints of every sink. We apply the proposed critical-trunk-based tree growth mechanism to solve PD and slack-driven (SD) OARST problems. The proposed algorithms are extended to consider buffer insertion during PD and SD OARST constructions. Experimental results demonstrate that the proposed algorithms achieve an average 25.84% improvement in the maximum delay over obstacle-avoiding rectilinear Steiner minimal tree in the PD OARST problem and successfully solve 66.67% worst negative slack violations in the SD OARST problem. Compared to the simultaneous routing and buffer insertion approach, the proposed buffer-aware (BA) algorithm generates satisfactory timing results with almost identical wire length (WL). Moreover, the proposed BA SD OARST algorithm utilizes less WL than the BA rectilinear Steiner tree construction does by 17.99% on average. The runtime comparison with previous works shows the efficiency and scalability of this paper.**

***Index Terms*—Buffer insertion, Elmore delay model, obstacle-avoiding rectilinear Steiner tree, performance-driven routing, timing constraint.**

I. Introduction

EVER SINCE THE feature size of processes dramatically decreased, the interconnection delay has dominated the circuit performance in nanometer designs. Topology optimization is one of primary approaches used in performance-driven (PD) interconnect design [1]. Cong *et al.* [2] defined an A-tree as a rectilinear Steiner tree (RST) where every path connecting the driver and any sink is the shortest path. Cong derived the delay upper bound from a distributed resistor–capacitor (RC) circuit and exploited the derived bound to optimize the delay of a Steiner tree. Pan *et al.* [1] efficiently constructed an A-tree based on FLUTE [3]. Considering the sink position, required time, and load capacitance, Pan modified the constructed A-tree to improve its timing by branch moving. Alpert *et al.* [4] analyzed the delay in distributed RC tree structures, demonstrating how the tree cost and the tree radius affect the signal delay. They proposed that AHHK trees are an effective combination of the Prim minimum spanning tree (MST) algorithm and the Dijkstra shortest path tree algorithm. Alpert *et al.* [5] proposed a bounded radius-ratio Steiner minimum tree based on a rectilinear minimum spanning tree (RMST) and the radius-ratio for each sink. The radius of each sink was defined as the path length from the driver to the sink, and the radius-ratio of each sink was its radius divided by the Manhattan distance from the driver to the sink. If the ratio exceeded a given value, they disconnected the sink and reconnected all disconnected sinks with a direct path to the driver. Considering the varying importance of sinks, Boese *et al.* [6] presented the critical-sink routing tree problem. The criticality of sinks is regarded as a weight, if necessary, and the routing tree is constructed by finding the one with the minimum weighted sum of sink delays. They assumed that critical sinks are already known. Hentschke *et al.* [7] proposed AMAZE applying maze routing on a grid model to construct RSTs. In addition to introducing a sharing factor and a path-length factor to achieve a tradeoff between wire length (WL) and delay, similar to [6], that work assigned criticality to sinks as the priority of delay optimization.

In modern intellectual property (IP)-block-based system-on-chip (SoC) designs, IP cores, logic blocks, and prerouted wires placed in the core before routing are considered as obstacles, which significantly lengthen wires and induce increasing delays. The obstacle-avoiding spanning graph (OASG) adopted by obstacle-avoiding rectilinear Steiner minimal tree (OARSMT) constructions [8]–[10] has a global view in terms of pins and obstacles and reduces WL effectively.

SoC designs must consider delay minimization and obstacle avoidance simultaneously. Directly applying previous timing-driven algorithms on an OASG is infeasible. Cong *et al.* [2] analyzed the delay in distributed RC circuits and observed that the delay upper bound is minimal if the tree

Manuscript received February 11, 2011; accepted March 25, 2011. Date of current version August 19, 2011. This work was supported in part by the National Science Council of Taiwan, under Grants NSC 97-2220-E-009-036, NSC 98-2220-E-009-025, and NSC 99-2220-E-009-012. This paper was recommended by Associate Editor J. Hu.

The authors are with National Chiao-Tung University, Hsinchu 300, Taiwan (e-mail: homeryenhung@gmail.com; evan218@yahoo.com.tw; ylli@cs.nctu.edu.tw).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2011.2150222

is both a shortest path tree (SPT) with minimal path length from the driver to a sink as well as an optimal Steiner tree (OST) with minimal WL. Therefore, the A-tree algorithm proposed in [2] maintains a SPT while steadily approaching an OST. However, identifying a SPT or an OST under obstacle-avoiding constraint is difficult because the obstacles may induce detours and worsen the delay of the constructed Steiner tree. In OASG, a vertex may be an obstacle's corner or a pin. Directly applying the algorithms proposed by previous works to solve the problem of PD RST construction without obstacles cannot be expected to obtain good results because a routing connecting two vertices may connect a pin to an obstacle's corner rather than another pin.

Xu *et al.* [11] first considered delay minimization and obstacle avoidance simultaneously. A dynamic searching and computation procedure was proposed to solve the minimal delay tree from the driver to the critical sink recursively. They also assumed that a given net only has a known critical sink. AMAZE [7] handled obstacles well due to the nature of maze routing and assigned each sink a weight to determine their criticality for delay optimization.

In SoC designs, buffer insertion is the major approach to optimize the delay of a long wire. There are two categories of buffered PD interconnection designs: one is to consider buffer insertion and Steiner tree construction simultaneously [12]–[14], and the other is to insert buffers after Steiner tree construction [15]–[17]. Cong and Yuan [12] considered obstacles and constructed buffered routing trees with fixed buffer constraints. Hrkic and Lillis [13] synthesized buffered tree with additional considerations, i.e., temporal locality, sink polarity requirements, and congestion. Dechu *et al.* [14] developed a PD routing tree with simultaneous buffer insertion and wire sizing in the presence of obstacles. However, the sizes of nets that can be solved by these methods are very limited. The maximum pin numbers of the circuits used in [12]–[14] are very small (10 in [12], 21 in [13], and 25 in [14]). Alpert *et al.* [15] demonstrated that the two-step approach of constructing a timing-driven RST first and then adopting van Ginneken style buffer insertion [18] can achieve almost the same quality of the approach which considers Steiner tree construction and buffer insertion simultaneously. When constructing timing-driven Steiner tree, Alpert *et al.* [16] considered porosity to insert buffers with obstacle-avoiding capability. They first constructed a timing-driven Steiner tree and then adjusted it based on obstacles and porosity. Finally, they applied van Ginneken style buffer insertion to improve timing. Alpert *et al.* [17] enabled critical nets to pass through blockages for hole usage by routing non-critical nets around blockages.

This paper first proposes a novel *critical-trunk-based tree growth*, an effective method for optimizing the Steiner tree delay by avoiding existing obstacles. Moreover, given the driver's arrival time and the required times for sinks, the critical-trunk-based tree growth can also be applied to optimize the worst negative slack (WNS) for an OARST. A routing algorithm is applied instead of the two-stage tree construction to seek the Steiner tree and thus have more control over the efficiency of the routing procedure. For SoC designs, the proposed algorithms are enhanced to consider buffer insertion during OARST construction. Experiments reveal that the proposed algorithms achieve an average 25.84% improvement in maximum delay over OARSMT. The algorithms can solve 66.67% of WNS problems. The runtime comparison with previous works shows the efficiency and scalability of this work.

The rest of this paper is organized as follows. Section II presents problem formulations. The critical-trunk-based tree growth is presented in Section III. The algorithms for optimizing delay and WNS are presented in Sections IV and V, respectively. The buffer insertion extension is presented in Section VI. Timing complexity analysis and experimental results are presented in Sections VII and VIII, respectively. Conclusions are finally drawn in Section IX.

II. Problem Formulations

As the efficiency of solving the obstacle-avoiding routing problem using spanning graph has been certified, this paper utilizes the OASG generation procedure in [10]. Elmore delay model [19] is adopted to estimate the wire delay in this paper. The proposed critical-trunk-based tree growth can be effectively applied to solve the following two problems.

A. Problem 1: PD/SD OARST

Let $P = \{p_0, \ldots, p_m\}$ be a set of pins where p_0 is the driver, and the other m nodes are sinks. Let $B = \{b_1, \ldots, b_k\}$ be a set of rectangular blockages. Timing-related information, such as unit wire resistance, unit wire capacitance, sink output loading, driver resistance, and some technology parameters, are given. The objective of PD OARST problem is to identify a RST that connects all pins in P and evades all obstacles in B in order to minimize $\max(delay(i))$, where $delay(i)$ is the delay of p_i. For SD OARST, the given input additionally includes the driver arrival time $T_{arr}(0)$ and sink required time $T_{req}(i)$, $i = 1, \ldots, m$. The slack of sink i is defined as $slack(i) = T_{req}(i) - T_{arr}, (i) = 1, \ldots, m$, where $T_{arr}(i)$ is the arrival time of sink i. The objective of this problem is to identify a RST that connects all pins in P and evades all obstacles in B in order to minimize the WNS.

B. Problem 2: Buffer-Aware PD/SD OARST

Given P, B, and timing-related information, the default type of buffer, denoted as b_d, and the upper bound capacitance, denoted as C_{ub}, represents the driving power of b_d. The objective of buffer-aware (BA) PD problem is to identify a buffered RST that connects all pins in P and evades all obstacles in B in order to minimize the maximum delay of all sinks. For BA SD OARST, additional given the driver arrival time and sink required time. The objective of this problem is to identify a buffered RST that connects all pin in P and evades all obstacles in B in order to minimize WNS.

III. Critical-Trunk-Based Tree

The topology of a Steiner tree affects the delays of all sinks. The critical-trunk-based tree growth mechanism is proposed to control the tree's topology for minimizing the maximum delay.

A. Critical Trunk

An important relation between the radius and the circuit delay is observed. This relation is common when constructing

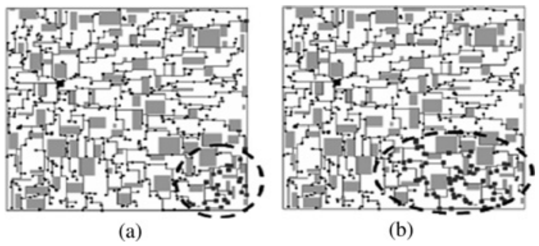


Fig. 1. Relation between radius and delay in SPT. (a) Enlarged rectangles within dotted circle are sinks with 80% of the worst radius in SPT. (b) Enlarged rectangles within dotted circle are sinks with 95% of the WD in SPT.

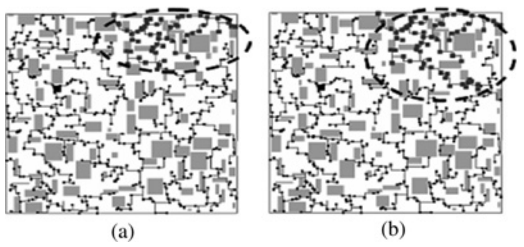


Fig. 2. Relation between radius and delay in MST. (a) Enlarged rectangles within dotted circle are sinks with 80% of the worst radius in MST. (b) Enlarged rectangles within dotted circle are sinks with 95% of the WD in MST.

Steiner trees with or without obstacles. In Figs. 1 and 2, the bold rectangular node in the upper left of each design is the driver. In Figs. 1(a)/2(a) and 1(b)/2(b), the circled and bold nodes in the lower/upper right are the sinks in which the radiuses are longer than 80% of the worst radius and those in which the delays are larger than 95% of the worst delay (WD) in a SPT/MST, respectively. Figs. 1 and 2 show that a long radius of a sink indicates a large delay for both a SPT and a MST. The long paths between the driver and the large radius sinks have higher possibility of connecting many subtrees than short paths do. Thus, the delays of sinks with large radiuses are easily raised by the increased downstream capacitance (DSC).

B. Subtree Topology

The topology of each subtree that subsequently grows starting at the path between the driver and the sinks with large radiuses must be well controlled to prevent from deteriorating the delays of these sinks. To grow subtrees properly, we utilize the following lemma.

Lemma 1: Given a set of pins P of a net. For a path ρ from the driver D to any sink s, the positions of large subtrees connecting to ρ affect the delay of s. If one large subtree approaches D, the delay of s decreases. If one large subtree approaches s, the delay of s increases.

Proof: In Fig. 3(a), there are n subtrees, $T_1, T_2, \ldots$, and T_n, connecting the path between the driver D and the sink s by pins, p_1, p_2, ..., and p_n, respectively. We derive the RC network in Fig. 3(b) from the tree in Fig. 3(a) using the π model. The delay of sink s, denoted as *delay* (s), is computed as follows:

$$\begin{aligned} delay(s) = R_{p_1} \cdot (C_{p_1} + C_{T_1} + C_{p_2} + C_{T_2} + \cdots + C_{p_n} + C_{T_n} \\ + C_s) + R_{p_2} \cdot (C_{p_2} + C_{T_2} + \cdots + C_{p_n} + C_{T_n} + C_s) \\ + \cdots + R_{p_n} \cdot (C_{p_n} + C_{T_n} + C_s) + R_s \cdot (C_s). \quad (1) \end{aligned}$$

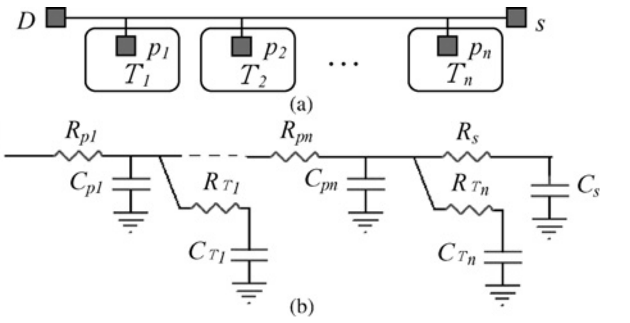


Fig. 3. Illustration of proof of Lemma 1. (a) $T_1, T_2, \ldots$, and T_n are subtrees connecting the path between driver D and sink s by pins, $p_1, p_2, \ldots$, and p_n, and p_n, respectively. (b) RC network transformed from the above tree using π model.

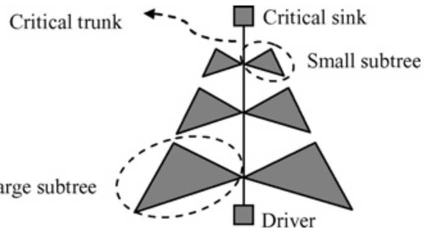


Fig. 4. Ideal tree topology for critical-trunk-based tree growth.

Then we can rewrite (1) as follows:

$$\begin{aligned} delay(s) = R_{p_1} \times (C_{p_1} + C_{T_1}) + (R_{p_1} + R_{P_2}) \times (C_{p_2} + C_{T_2}) \\ + \cdots + \sum_{i=1}^{n} R_{p_i} \times (C_{p_n} + C_{T_n}) \\ + \left(\sum_{i=1}^{n} R_{p_i} + R_s \right) \times C_s. \quad (2) \end{aligned}$$

We can note that the capacitance of each subtree contributes to the delay of the sink s based on its position. The Elmore delay multiplies the resistance of each pin by its DSC. When a subtree is closer to the sink, the subtree's capacitance needs to be multiplied more times. A subtree closer to the sink s has more impact on the delay of sink s than that closer to the driver D. In other words, if we have smaller subtrees close to the sink, a smaller delay can be obtained. ■

C. Critical-Trunk-Based Tree Growth Mechanism

From the above observation and Lemma 1, a critical-trunk-based tree growth is derived. Initially, to bound the delay of the sinks with the longest radius, the paths connecting the driver and these large-radius sinks are prerouted. The large-radius sinks are defined as critical sinks, and the prerouted paths are defined as critical trunks. For other unconnected sinks, subtrees are constructed to connect the unconnected sinks. These subtrees are then connected to the critical trunks according to the proposed tree topology to minimize delay in each sink. Fig. 4 depicts the ideal tree topology.

IV. PD OARST

PD OARST minimizes the delay for each sink by applying the critical-trunk-based tree growth mechanism. Fig. 5 displays the flow of the proposed performance-driven OARST algorithm. As mentioned above, the technique developed in

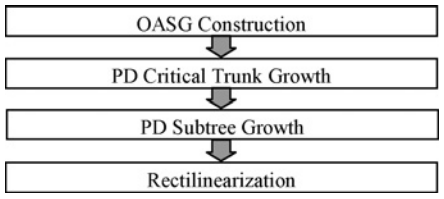


Fig. 5. Overall flow of PD OARST algorithm.

[10] is used to construct the OASG. The critical trunk in PD OARST is called the *PD critical trunk*. The growth of PD critical trunk on the OASG is the most crucial part of this algorithm, and the objective is to first connect the sinks with long radiuses due to their potential large delays. To conform to the ideal tree topology proposed in the critical-trunk-based tree mechanism, the size of each *PD subtree* connected to the PD critical trunks is well controlled. After constructing the Steiner tree, all slant edges in the OASG are transformed into horizontal and vertical edges, and a refinement operation removes redundant edges.

A. PD Critical Trunk Growth

An initial Steiner tree is constructed with adapted maze routing algorithm. Maze routing suffers from large runtime for large graphs. Line probe algorithm can better the runtime issue but the line extension approach to fit OASG that is not an array structure needs to be refined. An extended single-source single-target maze routing, called multisource single-target maze routing, is adopted to increase the flexibility of initial Steiner-tree construction. We then identify PD critical trunks of the initial Steiner tree and rip up the edges not in PD critical trunks.

1) *Initial Steiner Tree Construction:* A maze routing algorithm is adopted to construct the OARST; hence, the two-pin net generation is required for the PD critical trunk growth. The multiple-pin net is decomposed into several two-pin nets. The path in any two-pin net may generate detours to avoid obstacles, so the corners of each obstacle are also considered vertices of the OASG. The Prim algorithm starts at the driver and seeks a minimum-cost-spanning tree with m pin-to-pin connections (two-pin nets) in the OASG. The cost of an edge in the OASG is the Manhattan distance of its two end vertices. A two-pin net contains more than two OASG edges if it passes through at least one obstacle corner. The number of OASG edges is controlled by carefully selecting the OASG construction so that the two-pin net generation can be accomplished efficiently. Based on the generated two-pin nets, a maze routing with A* search is employed to construct an initial obstacle-avoiding Steiner tree, and the routing order is the same as that of the Prim algorithm.

2) *Multisource Single-Target Maze Routing:* For the increased flexibility of constructing an initial Steiner tree, each two-pin net is routed by the multisource single-target maze routing. Fig. 6 illustrates this method. In Fig. 6(a), node D is the driver of the routed net, the other gray nodes are the sinks and the corners of obstacles, the solid lines are the completed routing edges in the OASG, and the dotted line is the currently routed two-pin net where its source and target nodes are labeled S and T, respectively. In Fig. 6(b), a bounding box is

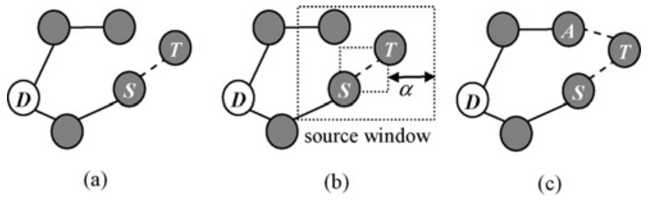


Fig. 6. Multisource single-target maze routing. (a) Dotted line is the currently routed two-pin net. (b) Source window is obtained by enlarging the enclosing bounding box of S and T with a factor α. (c) Node A is identified as another source for current routing.

constructed to enclose S and T, and then expand it by a user-defined parameter α to form a *source window*. In the source window, the vertex nodes within the source window which connect to D are identified as new source nodes. The source window is accomplished by R-tree [20] which is a spatial access method based on tree structures. Fig. 6(c) includes an additional source node connected to T.

3) *PD Critical Trunk Construction:* The first step to identify the PD critical trunks in a Steiner tree is recognizing PD critical sinks. Several important definitions for the PD OARST problem are given in the following.

Definition 1: PD criticality threshold factor (PDCTF): The PDCTF of a Steiner tree is the ratio of its average sink delay to its worst sink delay.

Definition 2: PD critical radius: The PD critical radius of a Steiner tree is the product of the maximum radius of the Steiner tree and the PDCTF.

Definition 3: PD critical sink: A sink is said to be PD critical if its radius exceeds the PD critical radius.

Definition 4: PD critical trunk: A path on the Steiner tree connecting the driver and any PD critical sink is a PD critical trunk.

The PDCTF represents the delay characteristics of the Steiner tree and determines the number of PD critical trunks to constrain the delay of PD critical sinks. The delay for Steiner trees with varying PDCTFs is analyzed as follows.

a) If the PDCTF approaches one, the average sink delay approximates the worst sink delay. In this case, all sinks have fairly consistent delays, and few critical trunks require prerouting. Prerouting PD critical trunks does not significantly enhance the performance and may worsen the average delay (AD) because the usage of PD critical trunks increases the total WL. Fig. 7(a) illustrates the PD critical trunks with a large PDCTF, where the bold paths are the PD critical trunks.
b) If the PDCTF approximates zero, the average sink delay is much smaller than the worst sink delay. Since the variations in sink delay are all significant, many PD critical trunks must be prerouted to reduce the longest radius and then the maximum delay, which are usually generated by the sinks with large radius. The AD can then be further improved. Fig. 7(b) illustrates the PD critical trunks with a small PDCTF. All paths originally connecting to the PD critical trunks are ripped up, and their reroutings are performed after the PD critical trunk growth. Fig. 8 displays the algorithm of the PD critical trunk growth.

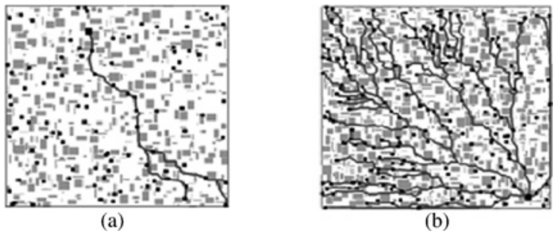


Fig. 7. Relation between PD critical trunk and PDCTF. (a) PD critical trunks for PDCTF = 0.854. (b) PD critical trunks for PDCTF = 0.473.

```
Algorithm: PD critical trunk growth
Input: An OASG (G), the driver (D), and sinks (S).
Output: two-pin nets and PD critical trunks.
begin
1.   generate two-pin nets N by the Prim algorithm;
2.   for each two-pin net n in N
3.       route n using the multi-source single-target maze
         routing with A* search;
4.   for each vertex v in G compute the radius R(v);
5.   for each sink s in S
6.       compute the delay using the Elmore delay;
7.   compute the PDCTF and the PD critical radius;
8.   for each sink s in S
9.       if (R(s) > PD critical radius) {
10.          mark s as PD critical sink;
11.          mark the path between D and s as PD critical trunk; }
12.  reserve PD critical trunks and rip up the other paths;
end
```

Fig. 8. Algorithm of PD critical trunk growth.

B. PD Subtree Growth

To control the delay of critical sinks, the critical-trunk-based tree favors large and small subtrees near the driver and critical sinks, respectively. Therefore, we propose a mechanism to constrain the topologies of subtrees connecting to the PD critical trunks. The *delay penalty factor* (DPF) is proposed to guide the maze routing to choose the proper connecting candidate, followed by how to adopt DPF in routing cost function.

1) *DPF:* The DPF is proposed to control the subtree topology. The DPF of node i is initially defined as follows:

$$DPF(i) = \begin{cases} \dfrac{R(i)}{R_{\max}}, & i \in N_{cr} \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

where $R(i)$ is the radius of node i on the Steiner tree, R_{max} is the maximum radius of the Steiner tree, and N_{cr} is the set of nodes on the critical trunks. The $DPF(i)$ increases as the distance between node i and the driver increases. During PD subtree growth, the ripped-up two-pin nets with the same order of previously generated two-pin nets by the Prim algorithm are rerouted. To control the tree topology during the PD subtree growth, DPF and *DPF inheritance* are employed in the multisource single-target maze routing. The DPF inheritance modifies the connected node, as Fig. 9 shows. Fig. 9 contains one driver and six sinks. In Fig. 9(a), the node labeled D is the driver, the node labeled s_c is the PD critical sink, the path labeled *PDCT* is the PD critical trunk, and the DPF of each node (except the driver) is self-labeled. Fig. 9(a) shows the initial DPF computed by (3).

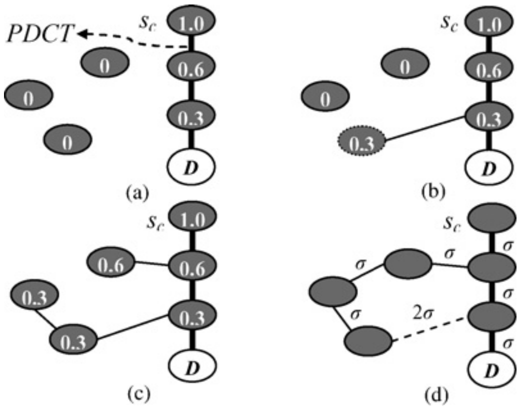


Fig. 9. DPF inheritance. (a) PD critical sink, PDCS, PD critical trunk, PDCT, and initial DPFs. (b) Updated DPF of newly connected node. (c) DPFs of all vertices after completing the routing tree. (d) Minimum-length routing tree excluding the dotted line and the length between corresponding vertices.

After completing one two-pin net routing, the newly connected node inherits the DPF from its neighboring node in the PD critical trunk as Fig. 9(b) shows. In the process of PD subtree growth, the DPF of the newly connected node inherits from its original neighboring node in the routing tree. Fig. 9(c) shows the final DPFs after all two-pin nets are routed. With the DPF inheritance, DPFs and the multisource single-target maze routing can be applied simultaneously to connect each disconnected node to the node in the routing tree with least routing cost integrating DPF, which is introduced in next section, to comply with the proposed tree topology. If multiple PD critical trunks are identified, for delay optimization, DPF guides each unconnected pin to connect to a proper vertex of some one growing critical trunk within the source window used in multisource single-target maze routing.

Fig. 9(d) shows a routing tree with the objective of minimizing the total WL. Assume the capacitance and resistance of every pin in Fig. 9 are C_p and R_p, respectively. The delay of critical sink s_c in Fig. 9(d) can be computed by the Elmore delay using the π model as follows:

$$\begin{aligned} delay(s_c) = {} & R_p \times (7 \cdot C_p + 6 \cdot \sigma \cdot c) + R_p \times (6 \cdot C_p \\ & + 5 \cdot \sigma \cdot c) + R_p \times (5 \cdot C_p + 4 \cdot \sigma \cdot c) + R_p \\ & \times (C_p + 0.5 \cdot \sigma \cdot c) = 19 \cdot R_p \cdot C_p + 15.5 \cdot R_p \cdot \sigma \cdot c \end{aligned} \tag{4}$$

where c is the wire unit-length capacitance and σ is the given WL. However, the delay of critical sink s_c in Fig. 9(c) can be computed similarly as follows:

$$\begin{aligned} delay(s_c) = {} & R_p \times (7 \cdot C_p + 7 \cdot \sigma \cdot c) + R_p \times (6 \cdot C_p + 6 \cdot \sigma \cdot c) \\ & + R_p \times (3 \cdot C_p + 2 \cdot \sigma \cdot c) + R_p \times (C_p + 0.5 \cdot \sigma \cdot c) \\ = {} & 17 \cdot R_p \cdot C_p + 15.5 \cdot R_p \cdot \sigma \cdot c. \end{aligned} \tag{5}$$

Although the WL of the routing tree in Fig. 9(c) is larger than that in Fig. 9(d), the delay of critical sink s_c in Fig. 9(c) is smaller than that in Fig. 9(d). With DPF inheritance, the adjusted topology of subtrees decreases the delay of s_c.

2) *Routing Cost Considering DPF:* In the multisource single-target maze routing, a delay-driven A* search-like method is applied for each two-pin net. The cost function for

```
Algorithm: PD subtree growth
Input: An OASG ($G$), two-pin nets ($N$), the driver ($D$), sinks ($S$),
and PD critical trunks ($T$).
Output: Obstacle-avoiding Steiner tree.
begin
  1. for each vertex $v$ in $G$ compute DPF;
  2. for each two-pin net $n$ of $N$
  3.     if (the target of $n$ is not in the routing tree){
  4.         route $n$ using the multi-source single-target maze
             routing with A* search-like method;
  5.         update DPF of two-pin net $n$; }
end
```

Fig. 10. Algorithm of PD subtree growth.

node x of A* search-like method is $f(x) = g(x) + h(x)$, where $g(x)$ is the cost function of the routed path, and $h(x)$ is the admissible heuristic distance estimation from node x to the target. The path-cost function $g(x)$ is reinforced to consider the delay effect by adding delay penalty cost as follows:

$$g(x) = dist_{sx} + dist_{ds} \times DPF(s)^2 \times ImpFactor \qquad (6)$$

where $dist_{sx}$ is the distance cost from the source to node x, $dist_{ds}$ is the distance cost from the driver to the source, $DPF(s)$ is the DPF of the source, and *ImpFactor* is the impact factor to balance total WL (total wire capacitance) and the WD. If the source is on a PD critical trunk and near the PD critical sinks, the subtree connecting this PD critical trunk at this source is imposed by high DPF upon the increase in WL. The DPF is used to obtain the trees that resemble the one in Fig. 4, and is reinforced in a square form because the delay is proportional to the square of WL. The $g(x)$ with a low impact factor lowers the importance of WD, and thus encourages reducing the total WL of Steiner tree with the penalty of increased worse delay. On the contrary, a high impact factor lowers the WD at the cost of increased total WL. Generally, AD increases as total WL increases. Fig. 10 shows PD subtree growth algorithm.

C. Rectilinearization

This stage transforms all edges into horizontal and vertical edges. For additional refinement, redundant edges are also removed. The U-shaped patterns are generated most frequently. A U-shaped pattern can be removed by breaking the L-shape in it. If a pin is not located at the corner of the L-shape, the L-shape can be replaced by a straight edge. Otherwise, the L-shape can be transformed into two parallel edges if the total WL after transformation is reduced. Fig. 11 shows these two cases. This paper adopts the Manhattan distance between two terminal vertices of an edge as the edge cost, thus the estimated delay before the rectilinearization is the same as that after rectilinearization without removing U-shapes. After performing the operation of removing U-shapes, the delay may be less than that before rectilinearization.

V. SD OARST

Minimizing the maximum delay in a Steiner tree may violate the timing constraint. To satisfy the timing constraints, the arrival time of each sink cannot exceed its required time: the slacks of all sinks should be at least zero. Figs. 12(a) and (b) presents two Steiner trees with different objectives—one to

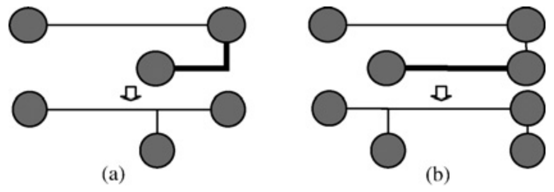


Fig. 11. Removal of redundant edges. (a) Remove the bold L-shape edge and connect the disconnected vertex with a vertical edge. (b) Remove the bold horizontal edge and connect the disconnected vertex with a vertical edge.

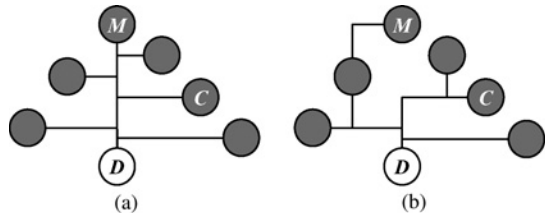


Fig. 12. Steiner tree with different objective where node D is the driver, node M is the sink with maximum delay, and node C is the sink with delay less than M. (a) Minimizing the maximum delay. (b) Satisfying the timing constraint.

minimize the maximum delay and the other to satisfy timing constraints. In Fig. 12(a), node D is the driver, node M is the sink with maximum delay, node C is the sink with delay less than M, and $T_{req}(C)$ is less than $T_{req}(M)$. If $T_{arr}(M) = delay(M)+T_{arr}(0)$ is less than $T_{req}(M)$, where $delay(M)$ is the delay of node M, and $T_{arr}(C)= delay(C)+T_{arr}(0)$ exceeds $T_{req}(C)$ where $delay(C)$ is the delay of node C, this Steiner tree violates the timing constraint at C ($T_{arr}(C) > T_{req}(C)$) rather than the worst-delay node M. In Fig. 12(b), $delay(M)$ is increased and $delay(C)$ is decreased, and the new Steiner tree satisfies the timing constraint. Unlike in the minimization of the maximum delay, the required time of every sink determines its importance while considering the timing constraints. A large required time of a sink corresponds to a large allowable delay of the sink. The slack is a good indicator of the allowable delay. If $slack(i)$ of sink i is less than zero, then $delay(i)$ is too large and should be reduced to satisfy the timing constraint. A positive $slack(i)$ implies that $delay(i)$ is legal and has a margin to be increased to reduce the other path delay and thereby improve related sink slacks.

This section presents a critical-trunk-based SD OARST algorithm to minimize the WNS of an OARST. Fig. 13 depicts the proposed SD OARST algorithm. The OASG construction is the same as that in the performance-driven OARST algorithm. The critical trunk in SD OARST is called the SD critical trunk. The growth of SD critical trunk is also the most important part of this algorithm, whose underlying concept is first to connect the sinks with small slacks to control their delays. The size of a SD subtree that is connected to the SD critical trunk is effectively controlled to satisfy the timing constraints. After the Steiner tree has been constructed, the same rectilinearization as that of the PD OARST algorithm is performed. Finally, the WNS of an OARST is minimized via a redirection mechanism.

A. SD Critical Trunk Growth

In PD OARST algorithm, the WL is used as the metric for Prim's algorithm because the delay is proportional to the

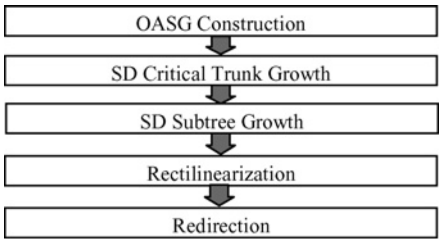


Fig. 13. Overall flow of SD OARST algorithm.

square of WL. The purpose of SD critical trunk growth is to well bound the delay of sinks with small required time to ensure the timing constraint can be satisfied, so the WL is not a proper metric. We can adopt required times to guide the two-pin nets generation in Prim's algorithm. Notably, only sinks in an OASG have required times while corners of obstacles do not. We propose the *pseudo required time computation* mechanism to guide the two-pin nets generation. According to the generated two-pin nets, the initial Steiner tree is constructed, and the critical trunks are identified according to the slack of each sink.

1) *Pseudo Required Time Computation:* To satisfy the timing constraint, the sinks with small required time prefer small delay in the constructed Steiner tree. Thus, the Prim algorithm is performed with edge cost simultaneously considering WL and the required time. But there is no information about required time except sinks in an OASG. Thus, we propose the *pseudo required time computation* mechanism to determine the possible required time for the vertices which represent the corners of obstacles in an OASG.

For the optimistic estimation of the required time for each vertex, the shortest delay path from the driver to each sink is first found. Then we estimate the possible required time for the vertex along the shortest delay path for each sink. The wave propagation used in the conventional routing algorithm is a good approach to find the shortest delay path. To obtain the upper bound of required time for each vertex, we adopt the modified wave propagation, called *potential delay propagation*, starting at the driver to compute the smallest delay for each vertex. Fig. 14 displays the algorithm of the potential delay propagation, where the $pdelay(i)$ is the potential delay for vertex i and $dist(i, j)$ is the distance from i to j. All edges in an OASG are initially enabled, and the *pdelay* of each vertex is set as infinite. In the beginning, the potential delay of the driver is set to zero, and the driver is inserted into a queue Q. In each iteration, vertex v is fetched from Q, and every edge e, connected to v, is checked whether e is enabled. If e is enabled and the potential delay of the other connected vertex v_{adj}, says $pdelay(v_{adj})$, exceeds $pdelay(v) + dist(v, v_{adj})^2$, then $pdelay(v_{adj})$ is updated as $pdelay(v) + distv(v, v_{adj})^2$ and v_{adj} records v as its source. If $pdelay(v_{adj})$ is updated, the edge e connecting v and v_{adj} is set as disable to prevent one path from passing e several times. Then v_{adj} is inserted into Q. This process continues until Q is empty. Then, each sink traces back by the source of each vertex to find the shortest path.

The *pseudo required time*, denoted as T_{p_req}, for each vertex in an OASG can be computed based on the potential smallest delay from the driver to any vertex. Initially, T_{p_req} s of every

```
Algorithm: Potential delay propagation
Input: An OASG (G), the driver (D), and sinks (S).
Output: Shortest paths for sinks.
begin
1.  for each vertex v in G  pdelay(v) = ∞;
2.  for each edge e in G enable e;
3.  pdelay(D)=0;   insert D into queue Q;
4.  while (!Q is empty)  {
5.      fetch vertex v from Q;
6.      for each adjacent vertex $v_{adj}$ of v
7.          if (edge e between $v_{adj}$ and v is enabled)  {
8.              if (pdelay($v_{adj}$)>pdelay(v)+dist($v_{adj}$,v) $^2$)  {
9.                  pdelay($v_{adj}$)=pdelay(v)+dist($v_{adj}$,v)$^2$;
10.                 source($v_{adj}$)=v;  disable e;   insert $v_{adj}$ into Q;  }  }  }
11. for each sink s in S construct $sp_s$ by back tracing source;
end
```

Fig. 14. Algorithm of potential delay propagation.

sink and every corner are set as its required time and infinity, respectively. For each sink i, its shortest path generated from the potential delay propagation is sp_i, and the *temporarily pseudo required time*, denoted as t_{p_req}, for every vertex j in s_{pi} is computed as follows:

$$t_{p\_req}(j) = T_{req}(i) - SMdist(i, j)^2 \times r \times c/2 \tag{7}$$

where $SMdist(i, j)$ is the sum of the Manhattan distance along sp_i from i to j and r is the wire unit-length resistance. After $t_{p\_req}(j)$ is computed, the pseudo required time of vertex j can be updated as follows:

$$T_{p\_req}(j) = \min(T_{p\_req}(j), t_{p\_req}(j)). \tag{8}$$

2) *Initial Steiner Tree Construction:* The pseudo required time for each vertex implicitly indicates the allowable delay. The vertex with a small pseudo required time needs connecting to the driver directly. If only pseudo required times are considered in the Prim algorithm, then the tree topology preferentially connects all sinks to the driver directly. The tree delay may increase dramatically, and the timing constraints are thus eventually violated. The WL of the tree and the pseudo required time should be considered simultaneously in the Prim algorithm to reach the objective of this algorithm. In the Prim algorithm, the OASG edge cost is computed dynamically as follows:

$$c_{i,j} = \begin{cases} dist(i, j), & if\ T_{p\_req}(j) = \infty, \\ dist(i, j) \times \dfrac{T_{p\_req}(j) - \min_{\text{prtV}}(T_{p\_req})}{\max_{\text{prtS}}(T_{p\_req})}, & \text{otherwise} \end{cases} \tag{9}$$

where $c_{i,j}$ is the edge cost, i is the original in-tree node, j is the connected node, $dist(i, j)$ is the Manhattan distance between i and j, $min_{prtV}(T_{p\_req})$ is the minimum pseudo required time among all vertices, and $max_{prtS}(T_{p\_req})$ is the maximum pseudo required time among all sinks. Initially, the edge costs of all edges connecting to the driver are set according to (9). The driver is set as visited, and the other vertices are set as unvisited. The edge with minimum cost and their connected vertices that are unvisited are included in the spanning tree, and the costs of all edges that connect the

```
Algorithm: SD critical trunk growth
Input: An OASG (G), the driver (D), sinks (S), the driver arrival
time, and sink required times.
Output: Two-pin nets, and SD critical trunks.
begin
 1.  for each corner vertex v in G  $T_{p\_req}(v)=\infty$;
 2.  for each sink s in S  $T_{p\_req}(s)=T_{req}(s)$;
 3.  generate shortest paths SP for every sinks by potential delay
     propagation;
 4.  for each path sp of sink s in SP
 5.     for each vertex v in sp update $T_{p\_req}(v)$ using (7) and (8);
 6.  generate two-pin nets N by required-time weighted Prim al-
     gorithm;
 7.  for each two-pin net n in N
 8.     route n using multi-source single-target maze
         routing with A* search;
 9.  for each sink s in S {
10.     compute the delay using Elmore delay;
11.     compute the priority priority(s); }
12.  compute SDCP;
13.  for each sink s in S {
14.     if (priority(s)<SDCP) {
15.        mark s as SD critical sink;
16.        mark the path between D and s as SD critical trunk; } }
17.  reserve SD critical trunks and rip up the other paths;
end
```

Fig. 15. Algorithm of SD critical trunk growth.

newly included vertex are assigned using (9) thereafter. The newly included vertex is also set as visited. The Prim algorithm repeats the identification of new edges and the assignment of vertex and edge costs until a minimum-cost spanning tree is identified. Based on the two-pin nets generated by required-time weighted Prim algorithm, multisource single-target maze routing with A* search can identify a good initial obstacle-avoiding Steiner tree to optimize slack.

3) *SD Critical Trunk Construction:* With the constructed Steiner tree, the priority of each sink, say i, is defined as $priority(i) = T_{p\_reqi}) - delayi)$. A negative priority implies a gap between the current arrival time and the ideal arrival time (pseudo required time) to satisfy the timing constraint. As in the PD OARST construction algorithm, the first step in identifying the SD critical trunks in the constructed Steiner tree is to identify the critical sinks. Several important definitions used in solving the slack-driven OARST problem are as follows.

Definition 5: SD critical priority (SDCP): The SDCP of a Steiner tree is the average priority of all sinks.

Definition 6: SD critical sink: A sink is said to be SD critical if its priority is smaller than SDCP.

Definition 7: SD critical trunk: A path on the Steiner tree that connects the driver to any SD critical sink is a SD critical trunk.

On the initial obstacle-avoiding Steiner tree, all paths that are connected to SD critical trunks are ripped up and rerouted after the SD critical trunk growth. Fig. 15 displays the algorithm of the SD critical trunk growth. As in Fig. 7(b), the algorithm may identify multiple critical sinks and their trunks.

B. Enhanced SD Critical Trunk Growth

The potential delay propagation used in the SD critical trunk growth is derived from the wave propagation in maze routing algorithm. The wave propagation is good at identifying the shortest delay path, but only considers the edge costs and ignores the number of passed vertices. Vertices of an OASG include pins and corners of obstacles. Obstacle corners are used to escape obstacles during routing and do not appear in physical paths, so we only consider pins in the following discussion. A shortest path may pass many pins and thus have worse delay as the number of passed pins increases. The following lemma shows the situation.

Lemma 2: For a fixed-length path connecting the driver to a sink, its delay increases as the number of passed pins increases.

Proof: Assume a path from the driver D to a sink s has a WL of l, and the capacitances of all sinks are equal. For simplicity and without loss of generality, n sinks are supposed to be evenly inserted in this path, the delay of s is shown to exceed that of the original path by $nlR_{\square}C_{pin}/2w$ in the following, where $R_{\square}$ is the sheet resistance [21], C_{pin} is the capacitance of all sinks, and w is the wire width.

First, we derive the delay of s in Fig. 16(a) by transforming it into a RC network using the π model as shown in Fig. 16(b), where R_s equals $lR_{\square}/N$, and C_s is the sum of the capacitances from the wire ($lwc/2$) and the sink s (C_{pin}). The delay of sink s in Fig. 16(b) is calculated as follows:

$$\begin{aligned} delay_{org}(s) &= R_s \times C_s = (lR_{\square}/w) \times (lwc/2 + C_{pin}) \\ &= l^2R_{\square}c/2 + lR_{\square}C_{Pin}/w. \end{aligned} \tag{10}$$

After inserting n sinks, we derive the RC network as shown in Fig. 16(d) from Fig. 16(c). The path in Fig. 16(c) are evenly divided into $(n + 1)$ segments, the resistances of all segments are equal. We derive the equations as follows:

$$R_{s1} = R_{s2} = R_{s3} = \cdots = R_{sn} = R_s = lR_{\square}/((n + 1)w) \tag{11}$$

$$C_{s1} = C_{s2} = C_{s3} = \cdots = C_{sn} = lwc/(n + 1) + C_{pin} \tag{12}$$

$$C_s = lwc/2(n + 1) + C_{pin}. \tag{13}$$

The delay of sink s in Fig. 16(d) is calculated as follows:

$$\begin{aligned} delay_{inserted}(s) = {} & R_{s1} \times (C_{s1} + C_{s2} + C_{s3} + \cdots + C_{sn} + C_s) + \\ & R_{s2} \times (C_{s2} + C_{s3} + \cdots + C_{sn} + C_s) + \cdots + \\ & R_{sn} \times (C_{sn} + C_s) + R_s \times C_s. \end{aligned} \tag{14}$$

After substituting (11) and (12) into (14), we can derive the equation as follows:

$$\begin{aligned} delay_{inserted}(s) = {} & R_{s1} \times [C_{s1} + 2C_{s2} + 3C_{s3} + \ldots + nC_{sn} \\ & +(n + 1)C_s] = R_{s1} \times [(n(n + 1)/2) \\ & \times C_{s1} + (n + 1)C_s]. \end{aligned} \tag{15}$$

Then substituting (11), (12), and (13) into (15), the delay of sink s in Fig. 16(c) is

$$\begin{aligned} delay_{inserted}(s) &= l^2R_{\square}c/2 + (n + 2)lR_{\square}C_{pin}/2w \\ &= l^2R_{\square}c/2 + lR_{\square}C_{pin}/w + nlR_{\square}C_{pin}/2w. \end{aligned} \tag{16}$$

■

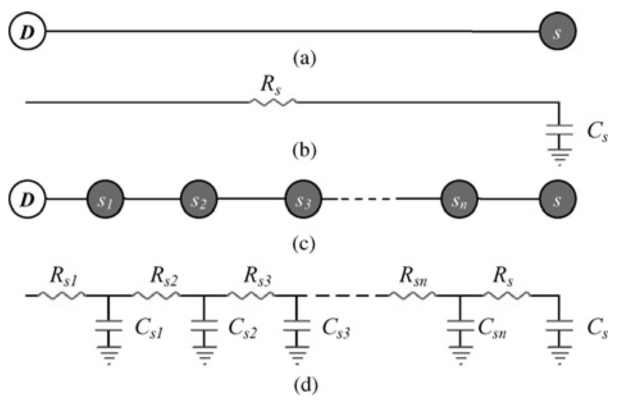


Fig. 16. Illustration of Lemma 2. (a) Path between driver D and sink s. (b) RC network of Fig. 16(a) using π model. (c) Path between D and s with averagely inserting n pins. (d) RC network of Fig. 16(c) using π model.

```
Algorithm: Potential delay propagation with pins consideration
Input: An OASG ($G$), the driver ($D$), and sinks ($S$).
Output: Shortest paths for sinks.
begin
1.  for each vertex $v$ in $G$ { $pdelay(v) = \infty$; $pin_{passed}(v) = \infty$; }
2.  for each edge $e$ in $G$ enable $e$;
3.  $pdelay(D)=0$;  $pin_{passed}(D) = 1$;  insert $D$ into queue $Q$;
4.  while (!$Q$ is empty) {
5.      fetch vertex $v$ from $Q$;
6.      for each adjacent vertex $v_{adj}$ of $v$
7.          if (edge $e$ between $v_{adj}$ and $v$ is enabled) {
8.              if ($pdelay(v_{adj})>pdelay(v)+dist(v_{adj},v)^2$
                    $\&pin_{passed}(v) < pin_{passed}(v_{adj})+\beta$) {
9.                  $pdelay(v_{adj})=pdelay(v)+dist(v_{adj},v)^2$;
10.                 $source(v_{adj})=v$;
11.                 if($v_{adj}$ is a pin) $pin_{passed}(v_{adj})=pin_{passed}(v)+1$;
12.                 else $pin_{passed}(v_{adj})=pin_{passed}(v)$;
13.                 disable $e$;  insert $v_{adj}$ into $Q$; } } }
14. for each sink $s$ in $S$ construct $sp_s$ by back tracing $source$;
end
```

Fig. 17. Algorithm of potential delay propagation with pins consideration.

Lemma 2 indicates that the delay of a sink increases as the number of passed pins from the driver to the sink increases with the used WL fixed. We propose an enhanced SD critical trunk construction by controlling the number of passed pins to prevent from obtaining a small edge-cost path but with many passed pins. Fig. 17 displays the algorithm of the potential delay propagation with pins consideration, where the $pdelay(i)$ is the potential delay for vertex i, $pin_{passed}(i)$ is the number of passed pins from D to$i$, and $dist(i, j)$ is the Manhattan distance from i to j. pin_{passed} of each vertex is initially set as infinity. Constant β is used to tradeoff the path length and the number of passed pins. In the experiments, β is set to 8. For an enabled edge e that connects currently visited vertex v with its adjacent vertex v_{adj}, if the potential delay of v_{adj}, says $pdelay(v_{adj})$, exceeds $pdelay(v) + dist(v,v_{adj})^2$ and the number of passed pins of v is smaller than that of v_{adj} plus β, then $pdelay(v_{adj})$ is updated as $pdelay(v)+dist(v, v_{adj})^2$, $pin_{passed}(v_{adj})$ is set as $pin_{passed}v) + 1$, or $pin_{passed}(v)$ depending on whether v_{adj} is a pin or not, v is set as the source of v_{adj}, edge e is disabled to avoid being re-visited, and finally v_{adj} is inserted into Q. This process continues until Q is empty. Then, each sink traces back by the sources of each vertex to find the minimum potential-delay path.

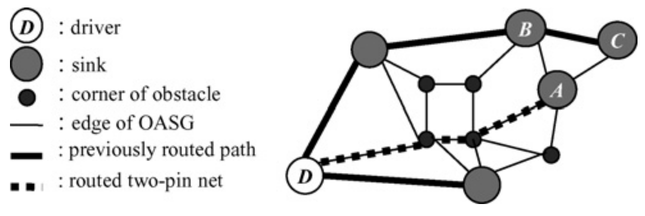


Fig. 18. Example of SD subtree growth.

C. SD Subtree Growth

After the growth of SD critical trunks, all disconnected pins are connected based on the two-pin nets generated by the required-time weighted Prim algorithm. To control the delay of each sink to fit its timing constraint effectively, each two-pin net is routed by single-source single-target maze routing. The required time determines principally the allowable delay of a sink in SD OARST problem. The SD critical trunk growth adopts the multisource single-target maze routing to increase the flexibility of constructing an initial Steiner tree. However, the source window of multisource single-target maze routing in a SD subtree growth may reduce the target delay of identified sources but increase the delays of other sinks. The timing constraint is eventually violated, and the WNS becomes worse. In Fig. 18, the dotted two-pin net between the driver D and the sink A are routed. If the multisource single-target maze routing is adopted, the sink B may become the extra source for connecting A and connect to A directly. The connection between A and B increases the DSC of C. Then the timing constraint for C may be violated due to the increased capacitance.

D. Redirection

After the Steiner tree has been rectilinearized, the Elmore delay model is applied to analyze the timing of SD OARST and then compute the arrival time and slack of each sink. If the slack of any sink is negative, then the redirect mechanism is exploited to improve the WNS. The redirection iteratively identifies the sink with the WNS, says s_{WNS}, disconnects s_{WNS}, and reconnects s_{WNS} to reduce its delay. s_{WNS} is disconnected by removing the routing path between s_{WNS} and its first upper stream node, which is a sink or a corner of an obstacle. The reconnection of s_{WNS} requires first selecting a candidate to be connected. To take full advantage of the previously constructed OASG, the nodes that are immediately connected to s_{WNS} are selected as candidates, and then the node with the smallest delay is chosen as the target to connect s_{WNS}. Following the reconnection, the newly connected edge is rectilinearized, and the timing is analyzed again to determine whether the redirection should be terminated. During the redirection, a sink may be reconnected to its original first upper stream node. In this case, further redirection will remove the paths between the sink and its second upper stream node, rather than its first node, to disconnect the sink from the Steiner tree. The redirection iterates until all negative slacks are solved or the runtime has reached a threshold value. Notably, the redirection mechanism does not change the tree topology because of the local modification. Experimental results show that the redirection mechanism improves the worst slacks in one and

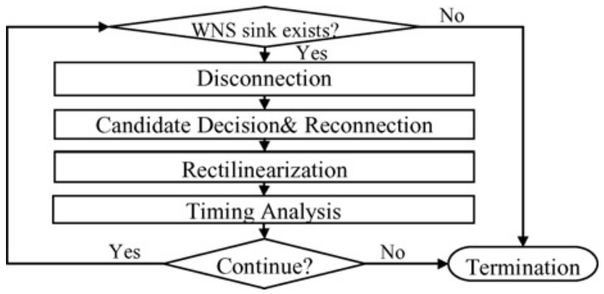


Fig. 19. Flow of redirection mechanism.

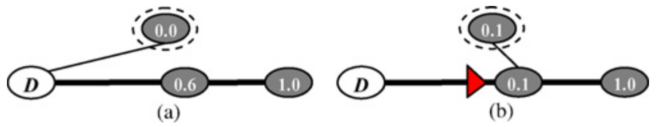


Fig. 20. Effect of buffer insertion to a PD OARST. (a) No buffer is inserted. (b) One buffer is inserted.

three cases of SD and BA SD OARST problems, respectively. Fig. 19 shows the redirection flow.

VI. Buffer-Insertion-Aware OARST

The PD/SD OARST algorithm assumes that one net contains one driving node. In the case with buffers inserted on a routing tree, the routing tree contains multiple driving nodes since each inserted buffer can be regarded as a driving node. Constructing PD/SD OARST without considering buffer insertion may lengthen the routing tree when OARST construction and buffer insertion are not considered simultaneously and performed sequentially. Fig. 20 illustrates the effect of inserting a buffer to a PD OARST. In Fig. 20(a), after constructing PD critical trunk (the bold lines), the circled vertex by dotted lines connects to the driver due to the DPF effect and inherits the DPF of the driver. In Fig. 20(b), one buffer is inserted, and the DPF of each vertex in the trunk is re-computed by regarding the inserted buffer as a driving node. The circled vertex connects to the middle vertex, resulting in decreasing the total WL. Thus, the ideal routing trees with and without inserted buffers for constructing PD/SD OARST might be quite different. The proposed algorithms are extended to consider buffer insertion during routing tree construction. The default type of buffer, denoted as b_d, is adopted, and the upper bound capacitance, denoted as C_{ub}, represents the driving power of b_d. The locations of inserted buffers (additional driving nodes) are predicted based on C_{ub}. A *pseudo buffer insertion* (PSBI) is proposed followed by the extensions of PD and SD OARST algorithms.

A. PSBI

An initial Steiner tree on the OASG, denoted as T_{init}, is constructed before PD/SD subtree growth in the same way as in PD/SD OARST. Since T_{init} significantly dominates the final outcome of PD/SD OARST, the proposed PSBI algorithm determines the positions of potential buffers on T_{init}. The DSCs of all nodes in T_{init} are first computed from each sink node toward the driver node. The DSC of one node v in T_{init}, denoted as $dsc(v)$, exceeding C_{ub} means one *pseudo buffer* needs to be inserted directly downstream to v. Fig. 21 depicts the algorithm of PSBI. The root (driver node) of T_{init} is the

```
Algorithm: PSBI
Input: A node $V_T$ in $T_{init}$
Output: Reduced downstream capacitance ($RDSC$).
begin
1. $RDSC = 0$;
2. for each child $ch$ of $V_T$ in $T_{init}$ $RDSC = RDSC + \text{PSBI}(ch)$;
3. $dsc(V_T) = dsc(V_T) - RDSC$;
4. if ($dsc(V_T) > C_{ub}$) { /* $C_{ub}$: Capacitance upper bound */
5.     insert a pseudo buffer at $V_T$;  $RDSC = RDSC + dsc(V_T)$; }
6. return $RDSC$;
end
```

Fig. 21. Algorithm of PSBI.

input of the algorithm. For each child ch of the input node V_T in T_{init}, if $dsc(ch)$ exceeds C_{ub}, one pseudo buffer is inserted near ch, and then $dsc(ch)$ is returned as the reduced DSC (RDSC) for neighboring upstream nodes. After computing the RDSC of each child, new $dsc(V_T)$ is obtained by subtracting the total RDSCs of all children from original $dsc(V_T)$ and then is examined if $dsc(V_T)$ is larger than C_{ub} and a pseudo buffer is required to be inserted around V_T. PSBI is a recursive procedure to insert buffers upward (from leaf to driver node) at the root of each subtree whose DSC after buffer insertion may exceed C_{ub}.

B. BA PD OARST

In BA PD OARST algorithm, pseudo buffers are inserted in the initial Steiner tree which is constructed to compute PDCTF. Only one driving node is considered using (3) to compute the DPF for controlling the subtree topology. With additional driving nodes, the *buffered radius* of node i, $R_{BUF}(i)$, is defined as the distance from i to its upstream driving node in the Steiner tree. The upstream driving node could be the driver or a pseudo buffer. The DPF in (3) is thus modified to consider pseudo buffers as follows:

$$DPF(i) = \begin{cases} \dfrac{R_{BUF}(i)}{R_{BUFmax}}, & i \in N_{cr} \\ 0, & \text{otherwise} \end{cases} \tag{17}$$

where R_{BUFmax} is the maximum buffered radius of the Steiner tree. The original path-cost function of maze routing in (6) is modified for pseudo buffers as follows:

$$g(x) = dist_{sx} + dist_{buf2s} \times DPF(s)^2 \times ImpFactor \tag{18}$$

where $dist_{buf2s}$ is the distance cost from the source to its upstream driving node and $DPF(s)$ is the source DPF computed by (17). Therefore, a BA PD OARST is constructed based on the flow of PD OARST.

C. BA SD OARST

In SD OARST algorithm, the potential delay propagation constructs an initial Steiner tree for pseudo required time computation. The BA SD OARST algorithm applies PSBI to the Steiner tree. The temporarily pseudo required time is computed using (7) for every node j in sp_i of each sink i. The BA SD OARST considers pseudo buffers by modifying (7) as follows:

$$t_{p\_req}(j) = T_{req}(i) - SMdist(i_{BUF}, j)^2 \times r \times c/2 \tag{19}$$

TABLE I
STATISTICS OF BENCHMARKS

Case	rc01	rc02	rc03	rc04	rc05	rc06
Pin	10	30	50	70	100	100
Obs.	10	10	10	10	10	500
Case	rc07	rc08	rc09	rc10	rc11	rc12
Pin	200	200	200	500	1000	1000
Obs.	500	800	1000	100	100	10 000

where $SMdist(i_{BUF}, j)$ is the sum of the Manhattan distance along sp_i from i_{BUF} to j. Sink i is the initial node i_{BUF}. If node j is pseudo buffered, i_{BUF} becomes j, which means that a pseudo buffer resets the delay. Therefore, the proposed algorithm can predict the delay considering inserted pseudo buffers to guide 2-pin net generation. The BA SD OARST can be constructed based on the original flow.

VII. Time Complexity Analysis

This paper adopts [10] for OASG construction, and its time complexity is $O(V \log V)$ by bounding the edge number of OASG where V is the vertex number of the OASG. The multiple-pin net with P_n pins is decomposed into (P_{n-1}) 2-pin nets by Prim's algorithm with the reduced time complexity of $O(E + V \log V)$ by using Fibonacci heap and adjacency list, where E denotes the edge number of the OASG. The time complexity of conventional maze routing in a $x \times y$ grid plane is $O(xy)$, i.e., $O(E)$, and the time complexity can be reduced as $O(\log E)$ by applying A* search. With (P_{n-1}) 2-pin nets, the time complexity of maze routing is $O(P_n \log E)$. Updating timing information for performance and SD OARST algorithms and PSBI require $O(V + E)$ to trace the Steiner tree. Therefore, the time complexity of this work is $O(P_n log E + E + V \log V)$.

VIII. Experimental Results

The algorithms herein were implemented in C++ language on a SUN UltraSPARC-III workstation with 1015 MHz central processing unit and 2 GB memory. A total 12 benchmarks (*rc01–rc12*) are adopted in OARSMT problems, and the corresponding statistics are shown in Table I. We adopt van Ginneken style buffer insertion (VGBI) [18], and the used buffer has 2.34 Ff load capacitance, 3.64 ps intrinsic delay, and 18 ohms resistance.

In (6), the *ImpFactor* is employed to balance total WL and WD. We vary the impact factor of the second term by setting *ImpFactor* to 20 different values, ranging from 0.05 to 1, and draw the variations of average improvement rates in WL, AD, and WD in Fig. 22. The base for comparison is the routing results using simplified (6) that only contains the first term ($dist_{sx}$). The average improvement rate for each item under a given *ImpFactor* is calculated as follows. For example, the WL improvement rate of each circuit is first computed as follows: $wire\_len\_imp = ((wire\_len_{base} - wire\_len_{impFact})/wire\_len_{base}) \times 100$. The average WL improvement rate can then be acquired by averaging the WL improvement rates of all circuits. As Fig. 22 shows, an ascending impact factor lengthens total WL and also increases AD while the approximated linear function of WD (dotted line) implies a slow increasing WD improvement rate is achieved

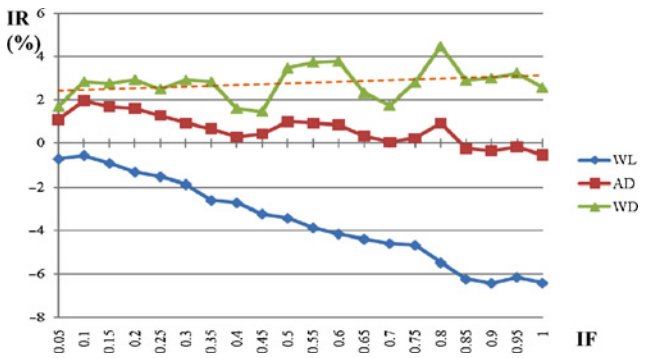


Fig. 22. In PD OARST problem, average improvement rates of WL, AD, and WD are impacted by different *ImpFactor* (IF) values.

TABLE II
COMPARISON OF WL, WDS, AND RUNTIMES AMONG MOARSMT (MO), PD OARST (PD), AND PD OARST WITHOUT U-SHAPE REMOVE (PDU)

Case	WL			WD			Runtime		
	MO (um)	PD *inc*	PDU *inc*	MO (ps)	PD *imp*	PDU *imp*	MO (s)	PD *imp*	PDU *imp*
rc01	26 810	8.69	9.55	3709.40	8.78	7.89	0.01	0	0
rc02	42 280	1.63	5.63	4757.91	14.14	11.50	0.01	−100	−100
rc03	56 160	10.88	14.26	8906.42	34.94	31.71	0.01	−200	−200
rc04	60 710	21.68	25.76	8124.20	27.27	25.59	0.02	−100	−100
rc05	77 330	12.92	14.82	11690.13	42.14	39.91	0.03	−66.7	−66.7
rc06	86 299	9.78	12.60	10685.59	−6.65	−11.99	0.21	−114	−110
rc07	116 801	10.34	13.13	13450.84	18.10	16.18	0.20	−240	−235
rc08	123 004	11.47	15.25	16169.9	29.66	27.05	0.29	−269	−266
rc09	120 062	24.33	27.68	20957.15	26.65	24.64	0.59	−227	−225
rc10	170 600	9.04	12.08	25946.16	39.05	34.94	0.11	−390	−364
rc11	238 905	5.92	8.42	36459.46	11.69	6.83	0.38	−208	−197
rc12	858 310	51.13	52.74	464903.00	64.25	63.97	15.95	−748	−743
Average		14.82	17.66		25.84	23.19		−222	−217

[PD/PDU] WL *inc* = ([PD/PDU]-MO)/ MO × 100.
[PD/PDU] WD *imp* = (MO-[PD/PDU])/ MO × 100.
[PD/PDU] runtime *imp* = (MO-[PD/PDU])/MO × 100.

as the *ImpFactor* increases. The best WD improvement rate occurs at the point *ImpFactor* = 0.8, thus we set *ImpFactor* to 0.8 in the following experiments.

To demonstrate the efficiency of the proposed algorithm, a maze-routing based OARSMT algorithm (MOARSMT) is implemented by removing the PD critical trunk growth and the PD subtree growth, and performing multisource single-target routings using A* search algorithm without timing information. Table II compares the proposed PD OARST algorithm (PD) with MOARSMT and the OARSMT (PDU), U-shapes are not removed while performing PD. The experimental environment settings are as follows: 440 ohms for driver resistance, 0.076 ohms/um for unit wire resistance, 0.118 Ff/um for unit wire capacitance, and 1 Ff for the loading capacitance of sink. The delay is calculated by the Elmore delay model. The increases in average WL and runtime are 14.82% and 222%, respectively. The significant drop in the routing speed of PD is primarily caused by the DPF inheritance, PD subtree growth, and timing analysis. The average 25.84% reduction in the WD indicates the efficiency of the critical-trunk-based tree growth. The results of PDU display the impact of removing the U-shapes. PD obtains lower WL and WD than PDU since they have the same topology, yet PDU has a longer WL.

Table III compares AMAZE and PD in terms of WL, the WD, AD, and runtime. In the used benchmarks, a pin may be located at the boundary of a blockage. However, in

TABLE III
COMPARISON OF WL, WDS, ADS, AND RUNTIMES BETWEEN AMAZE [7] (AM) AND PD OARST (PD)

Case	WL		WD		AD		Runtime	
	AM (um)	PD *inc*	AM (ps)	PD *imp*	AM (ps)	PD *imp*	AM (s)	PD *imp*
rc01	26 251	11.01	3716.0	8.94	3078.3	2.29	0.15	93.33
rc02	42 570	−1.17	4589.8	10.99	3583.1	−8.88	0.38	94.74
rc03	54 602	10.97	9300.6	37.69	6583.6	26.38	0.64	95.31
rc04	60 082	12.71	7979.9	25.96	6173.71	21.95	1.34	97.01
rc05	74 882	15.84	13471.9	49.79	10032.3	39.23	1.66	96.99
rc06	82 787	9.65	10418.5	−9.38	7400.1	−5.64	31.1	98.55
rc07	112 703	9.01	13933.0	20.94	10284.5	5.44	72.6	99.06
rc08	118 638	8.57	15457.0	26.41	12853.5	17.83	201.7	99.47
rc09	115 516	17.05	24551.9	37.39	17516.1	18.99	302.8	99.36
rc10	165 649	5.95	19136.6	17.36	14397.5	4.44	135.9	99.60
rc11	235 881	3.68	54890.6	41.35	40888.1	45.71	2079.2	99.94
rc12	–	–	–	–	–	–	–	–
Average		9.39		24.31		15.25		97.58

PD WL *inc* = (PD-AM)/AM × 100.
PD WD *imp* = (AM-PD)/AM × 100.
PD AD *imp* = (AM-PD)/AM × 100.
PD runtime *imp* = (AM-PD)/AM × 100

TABLE IV
COMPARISON OF WL, WDS AFTER VGBI, INSERTED BUFFER NUMBERS AFTER BUFFER INSERTION, AND RUNTIMES BETWEEN PD OARST (PD) AND BPD

Case	WL		WD af. VGBI		Buffer #		Runtime	
	PD (um)	BPD *imp*	PD (ps)	BPD *imp*	PD	BPD	PD (s)	BPD *imp*
rc01	29 140	8.10	288.66	56.27	15	19	0.01	0.00
rc02	42 970	1.61	66.42	62.52	1	2	0.02	0.00
rc03	62 270	9.81	81.44	63.26	3	5	0.03	0.00
rc04	73870	16.8	62.1	48.81	3	7	0.04	0.00
rc05	87 320	11.44	48.81	55.56	4	3	0.05	0.00
rc06	94 742	9.20	42.98	−44.00	9	15	0.45	40.00
rc07	128 875	9.85	28.32	29.80	4	3	0.68	42.65
rc08	137 116	10.8	32.66	1.75	6	9	1.07	45.79
rc09	149 274	19.57	33.17	−14.35	7	12	1.93	64.25
rc10	186 030	8.31	50.92	61.21	10	4	0.54	25.93
rc11	253 039	5.62	28.24	34.21	3	3	1.17	23.93
rc12	1 297 170	33.92	116.83	68.38	30	7	135.31	86.69
Average		12.03		35.29	7.92	7.42		27.44

BPD WL *imp* = (PD-BDP)/PD × 100.
BPD WD after VGBI *imp* = (PD-BPD)/PD × 100.
BPD runtime *imp* = (PD-BPD)/PD × 100.

AMAZE, the boundaries of blockages are not allowed for routing. We shrink the boundary of a blockage by one unit if a pin is located on it in order to satisfy the blockage constraint. Sharing and path-length factors are set to 0 and 1, respectively, as suggested in AMAZE. AMAZE involves performing experiments with generated nets and determined critical sinks randomly. For comparison, the Manhattan distance between the driver and each sink is used to determine the criticality of each sink for AMAZE, i.e., the sink with the longest/shortest Manhattan distance owns the highest/lowest criticality. AMAZE does not identify a feasible result within 12 h for the case *rc12*. Notably, PD increases in average WL by 9.39%. The average reduction rates in the worst and ADs, i.e., 24.31% and 15.25%, respectively, demonstrate the efficiency of the proposed algorithm. The runtime of PD is faster than AMAZE by 97.58% on average.

Table IV compares PD and buffer-aware PD (BPD) in terms of WL, the WD after VGBI, inserted buffer number after VGBI, and runtime. BPD effectively reduces the average WL and the WD after VGBI by 12.03% and 35.29%, respectively. The WDs after VGBI of BPD in *rc06* and *rc09* are worse than those of PD because the increasing number of obstacle corners affects the accuracy of pseudo buffer locations. The inserted buffer numbers of PD and BPD are almost the same on average. Exchanging $dist_{ds}$ in (6) by $dist_{buf2s}$ in (18) enlarges the routing cost difference of adjacent vertices and then accelerates the routing process. Therefore, the runtime of BPD is averagely faster than PD by 27.44%.

For the comparison of OARST construction with and without slack consideration, the benchmarks (*rc01–rc12*) for OARSMT are reinforced as new benchmarks (*rc01t–rc12t*) under the constraint of the required time of every sink. The required time of each sink is computed by finding the SPT on the OASG and then calculating the delay of each sink by the Elmore delay model. Table V compares the proposed SD OARST algorithm (SD) to PD in terms of WL, the WD, the WNS, and runtime. The SD exhibits an increase of 16.26% in WL on average and solves the WNS-violation problems in eight cases while PD does only in three cases. The runtime of SD is faster than PD by 23.89% on average. Moreover, the WD of SD is 1.35% better on average than that of PD, because the Elmore delay is implicitly adopted to find an SPT in the required-time weighted Prim algorithm, and the Elmore delay is more effective to estimate wire delay than WL.

Table V also lists the routing result of the SD OARST algorithm with the enhanced SD critical trunk construction (ESD). ESD constrains the growth of the number of passed pins to gain small delay, and the WL of EDS is thus shorter than that of SD. Moreover, the WD of ESD is almost the same with that of SD, and one WNS violation that cannot be solved by SD is solved by EDS. The runtimes of ESD and SD are almost the same.

Table VI compares SD, buffer-aware SD (BSD), and S-tree [13] in terms of WL, the worst slack after VGBI, inserted buffer number after VGBI, and runtime, where S-tree considers tree construction and buffer insertion simultaneously. The results of [13] are obtained by performing the tool provided by the authors. S-tree cannot complete *rc05t–rc12t* within 12 h. Compared to SD, BSD effectively reduces the average WL by 10.59% at the cost of 0.01% increase of the worst slack after VGBI. Moreover, BSD also reduces the inserted buffers on average. When the pins exceed obstacle corners in OASG, e.g., circuits *rc05t* and *rc11t*, BSD spends more runtime than SD in PSBI and related timing update. S-tree adopts dynamic programming approach, which can explore more solution space than those of the proposed heuristic algorithm, to obtain better WL and worst slack using less inserted buffers than BSD at the cost of large computation time. For instance, S-tree requires more than 220 min completing *rc04t* while BSD only demands 0.02 s. Table VI demonstrates that BSD obtains comparable quality to S-tree in very short runtime.

Herein, we also compare SD and BSD to C-tree [15], a combination of timing-driven Steiner tree construction and buffer insertion without considering obstacles. The results of C-tree are obtained by performing the tool provided by the authors. A comparison with the lookup-table-based mechanism [1] would also be of value, but the binary code and test cases are unavailable. A new set of benchmarks (*rc01tn–rc12tn*) is

TABLE V

COMPARISON OF WL, WDs, WORST SLACKS, AND RUNTIMES AMONG PD OARST (PD), SD OARST (SD), AND SD OARST WITH ENHANCED SD CRITICAL TRUNK CONSTRUCTION (ESD)

Case	WL			WD			Worst Slack			Runtime		
	PD (um)	SD *inc*	ESD *inc*	PD (ps)	SD *imp*	ESD *imp*	PD (ps)	SD (ps)	ESD (ps)	PD (s)	SD *imp*	ESD *imp*
rc01t	29 140	3.88	3.88	3383.65	9.08	9.08	−635.78	16.84	16.84	0.01	0.00	0.00
rc02t	42 970	12.27	12.27	4085.18	−11.01	−11.01	−567.09	−540.89	−540.89	0.02	50.00	50.00
rc03t	62 270	8.39	7.76	5794.79	−19.02	−19.02	−1081.18	−563.34	−563.34	0.03	66.67	66.67
rc04t	73 870	5.78	5.78	5908.49	1.38	1.38	−321.38	46.88	46.88	0.04	50.00	75.00
rc05t	87 320	11.48	11.48	6763.97	−1.54	−1.54	233.28	377.25	377.25	0.05	60.0	60.0
rc06t	94 742	28.47	28.47	11396.00	17.02	17.02	−1412.06	547.83	547.83	0.45	2.22	2.22
rc07t	128 875	37.92	37.92	11015.90	−14.91	−14.91	298.42	555.13	555.13	0.68	−7.35	−8.82
rc08t	137 116	32.33	32.33	11374.60	−14.70	−14.70	1595.35	1019.36	1019.36	1.07	−38.32	−37.38
rc09t	149 274	19.44	19.44	15371.10	3.55	3.55	−1097.07	129.21	129.21	1.93	28.50	28.50
rc10t	186 030	12.10	12.10	15815.30	4.17	4.17	−725.628	584.02	584.02	0.54	44.44	44.44
rc11t	253 039	13.28	13.04	32195.80	36.77	33.27	−11545.70	−489.24	63.69	1.17	32.48	38.46
rc12t	1 297 170	9.83	9.83	166205.00	5.43	5.43	−48206.60	−1135.19	−1135.19	135.31	−1.90	−1.06
Average		16.26	16.19		1.35	1.06					23.89	26.50

SD WL *inc* = (SD-PD)/PD × 100.
SD WD *imp* = (PD-SD)/PD × 100.
SD runtime *imp* = (PD-SD)/PD × 100.
ESD WL *inc* = (ESD-PD)/PD × 100.
ESD WD *imp* = (PD-ESD)/PD × 100.
ESD runtime *imp* = (PD-ESD)/PD × 100.

TABLE VI

COMPARISON OF WL, WORST SLACKS AFTER BUFFER INSERTION, INSERTED BUFFER NUMBER AFTER BUFFER INSERTION (VGBI), AND RUNTIMES AMONG SD OARST (SD), BUFFER-AWARE SD OARST (BSD), AND S-TREE (ST) [13]

Case	WL			Worst Slack After VGBI			Buffer #			Runtime		
	SD (um)	BSD *imp*	ST *imp*	SD (ps)	BSD *imp*	ST *imp*	SD	BSD	ST	SD (s)	BSD *imp*	ST *imp*
rc01t	30 270	13.25	12.19	1804.90	0.00	1.14	17	14	2	0.01	0.00	−15800
rc02t	48 240	8.17	10.12	2915.43	0.01	0.76	7	1	2	0.01	0.00	−211600
rc03t	65 620	5.75	13.99	4142.63	−0.05	−4.59	5	6	5	0.01	0.00	−39322900
rc04t	78 140	19.25	20.37	4456.62	0.00	0.36	7	3	2	0.02	0.00	−66143400
rc05t	97 350	8.33	–	6240.45	−0.05	–	15	2	–	0.02	−150.00	–
rc06t	121 633	9.56	–	7871.14	0.00	–	32	9	–	0.44	34.09	–
rc07t	177 741	11.01	–	10570.09	0.00	–	5	2	–	0.73	47.95	–
rc08t	181 445	9.18	–	11974.26	0.01	–	34	29	–	1.48	26.25	–
rc09t	178 289	13.97	–	11785.49	−0.04	–	25	14	–	1.38	30.43	–
rc10t	208 540	2.00	–	13609.72	−0.02	–	1	3	–	0.30	16.67	–
rc11t	286 377	5.85	–	17384.87	0.00	–	2	1	–	0.79	−31.65	–
rc12t	1 424 770	20.73	–	92672.51	−0.04	–	41	20	–	137.90	73.93	–
Average		10.59	14.17		−0.01	−0.58	15.92	8.67	2.75		3.98	−26423425

BSD WL *imp* = (SD-BSD)/SD × 100.
BSD worst slack *imp* = (BSD-SD)/SD × 100.
BSD runtime *imp* = (SD-BSD)/SD × 100.
ST WL *imp* = (SD-ST)/SD × 100.
ST worst slack *imp* = (ST-SD)/SD × 100.
ST runtime *imp* = (SD-ST)/SD × 100.

TABLE VII

COMPARISON OF WL, WORST SLACKS BEFORE/AFTER BUFFER INSERTION (VGBI), INSERTED BUFFER NUMBER AFTER BUFFER INSERTION, AND RUNTIMES AMONG C-TREE (CT) [15], SD OARST (SD), AND BUFFER-AWARE SD OARST (BSD)

Case	WL			Worst Slack Before VGBI			Worst Slack After VGBI			Buffer #			Runtime		
	CT (um)	SD *imp*	BSD *imp*	CT (ps)	SD (ps)	BSD (ps)	CT (ps)	SD *imp*	BSD*imp*	CT	SD	BSD	CT (s)	SD *imp*	BSD *imp*
rc01tn	30 270	6.77	6.31	−214.56	−30.64	−34.55	1785.29	0.00	3.26	13	6	15	0.02	50.00	50
rc02tn	49 830	3.63	16.42	−4474.01	−14.62	−487.00	2694.39	−0.31	0.03	35	3	1	0.07	85.71	85.71
rc03tn	69 820	10.24	13.91	−847.29	−55.22	−1701.3	3421.95	6.35	6.90	63	5	6	0.20	95.00	90.00
rc04tn	74 240	−4.53	2.56	−274.15	−69.69	−1153.08	4119.18	−0.27	0.02	73	2	3	0.26	92.31	88.46
rc05tn	103 640	14.65	25.35	−1186.67	−72.12	186.29	5449.05	0.69	3.01	110	2	7	0.58	96.55	96.55
rc06tn	111 690	24.48	24.84	−1551.37	364.35	−147.23	5131.22	−0.24	0.11	112	2	8	0.46	95.65	96.65
rc07tn	156 136	19.45	20.64	−2239.39	208.08	−2392.87	7710.04	3.27	3.41	201	3	1	1.91	97.38	93.19
rc08tn	156 036	20.82	20.91	−1670.44	213.6	−3710.27	7470.57	−0.45	0.01	207	1	2	1.7	97.06	91.76
rc09tn	167 048	20.05	20.74	−3037.44	−769.52	−2985.36	7932.97	−0.19	0.35	207	2	2	1.9	95.79	92.63
rc10tn	245 220	16.83	20.68	−3695.90	−1610.8	−9044.43	11814.4	1.36	0.01	407	3	3	13.93	97.70	96.20
rc11tn	341 752	19.71	21.28	−6749.56	−531.54	−10173.10	16733.9	−0.02	0.20	709	2	2	78.25	98.62	97.96
rc12tn	1 073 720	18.64	22.26	−49733.9	−38844.7	−273032.0	53324.4	0.39	0.00	1193	7	4	104.79	99.02	97.88
Average		14.23	17.99	−5972.89	−3434.40	−25389.57		0.88	1.44	277.5	3.17	4.5		91.73	89.67

SD WL *imp* = (CT-SD)/CT × 100.
SD worst slack *imp* = (SD-CT)/CT × 100.
SD runtime *imp* = (CT-SD)/CT × 100.
BISD WL *imp* = (CT-BSD)/CT × 100.
BSD worst slack *imp* = (BSD-CT)/CT × 100.
BSD runtime *imp* = (CT-BSD)/CT × 100.

generated here by removing the obstacles in *rc01t–rc12t*. The comparison to C-tree is based on the new benchmark set. Compared to C-tree, the SD and BSD averagely reduce the WL by 14.23% and 17.99% and improve the worst slack after VGBI by 0.88% and 1.44%, respectively. The average worst slack before VGBI of BSD is extremely worse than those of C-tree and SD. However, after VGBI, the worst slack of BSD in all cases is better than those of C-tree and SD, which shows the efficiency of PSBI. The number of inserted buffers of SD and BSD are less than those of C-tree by more than 98% on average. Without conducting global optimization, C-tree clusters sinks into several sets and then constructs a Steiner tree for each sink set, resulting in suboptimum in both terms of timing and wirelength. The quality of identified Steiner tree may degrade as the sink number increases. For instance, C-tree inserts 298.25 times the number of buffers required by BSD in *rc12tn*.

IX. Conclusion

This paper proposed a critical-trunk-based tree growth mechanism and applied it to construct an obstacle-avoiding Steiner tree with the objectives of minimizing the sink delay and maximizing the WNS of the Steiner tree, respectively. A multisource single target maze routing increased the flexibility of constructing an initial Steiner tree. In the PD OARST algorithm, a DPF inheritance controlled the topology of subtrees connected to the PD critical trunks. In the SD OARST algorithm, a pseudo required time computation guided the sinks with small required time to be connected. We also proposed an enhanced SD critical trunk growth that limited the number of passed pins in a routing to improve delay. A redirection mechanism improved the delay of the sink with the WNS as much as possible. Finally, the proposed algorithms were enhanced to consider buffer insertion during routing. The experimental results indicated that the proposed algorithms were very promising in terms of both solution quality and runtime.

References

[1] M. Pan, C. Chu, and P. Patra, "A novel performance-driven topology design algorithm," in *Proc. Asia South Pacific Des. Automat. Conf.*, 2007, pp. 244–249.

[2] J. Cong, K.-S. Leung, and D. Zhou, "Performance-driven interconnection design based on distributed RC delay model," in *Proc. 30th ACM/IEEE Int. Conf. Des. Automat.*, Jun. 1993, pp. 606–611.

[3] C. Chu and Y.-C. Wong, "FLUTE: Fast lookup table based rectilinear Steiner minimal tree algorithm for VLSI design," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 27, no. 1, pp. 70–83, Jan. 2008.

[4] C. J. Alpert, T. C. Hu, J. H. Huang, A. B. Kahng, and D. Karger, "Prim-Dijkstra tradeoffs for improved performance-driven routing tree design," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 14, no. 7, pp. 890–896, Jul. 1995.

[5] C. J. Alpert, A. B. Kahng, C. N. Sze, and Q. Wang, "Timing-driven Steiner trees are (practically) free," in *Proc. 43rd ACM/IEEE Des. Automat. Conf.*, Jul. 2006, pp. 389–392.

[6] K. D. Boese, A. B. Kahng, B. A. McCoy, and G. Robins, "Near-optimal critical sink routing tree constructions," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 14, no. 12, pp. 1417–1436, Dec. 1995.

[7] R. Hentschke, J. Narasimhan, and R. Reis, "Maze routing Steiner trees with delay versus wire length tradeoff," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 17, no. 8, pp. 1073–1086, Aug. 2009.

[8] Z. Shen, C. C. N. Chu, and Y.-M. Li, "Efficient rectilinear Steiner tree construction with rectilinear blockages," in *Proc. IEEE Int. Conf. Comput. Des.*, Oct. 2005, pp. 38–44.

[9] C.-W. Lin, S.-Y. Chen, C.-F. Li, Y.-W. Chang, and C.-L. Yang, "Obstacle-avoiding rectilinear Steiner tree construction based on spanning graphs," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 27, no. 4, pp. 643–653, Apr. 2008.

[10] J. Long, H. Zhou, and S. O. Memik, "EBOARST: An efficient edge-based obstacle-avoiding rectilinear Steiner tree construction algorithm," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 27, no. 12, pp. 2169–2182, Dec. 2008.

[11] J. Xu, X. Hong, T. Jing, and Y. Yang, "Obstacle-avoiding rectilinear minimum-delay Steiner tree construction towards IP-block-based SOC design," in *Proc. 6th Int. Symp. Qual. Electron. Des.*, 2005, pp. 616–621.

[12] J. Cong and X. Yuan, "Routing tree construction under fixed buffer locations," in *Proc. Des. Automat. Conf.*, 2000, pp. 379–384.

[13] M. Hrkic and J. Lillis, "Buffer tree synthesis with consideration of temporal locality, sink polarity requirements, solution cost, congestion and blockages," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 22, no. 4, pp. 481–491, Apr. 2003.

[14] S. Dechu, Z. C. Shen, and C. C. N. Chu, "An efficient routing tree construction algorithm with buffer insertion, wire sizing and obstacle considerations," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 24, no. 4, pp. 600–608, Apr. 2005.

[15] C. J. Alpert, M. Hrkic, J. Hu, A. B. Kahng, J. Lillis, and B. Liu, "Buffered Steiner trees for difficult instances," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 21, no. 1, pp. 3–14, Jan. 2002.

[16] C. J. Alpert, G. Gandham, M. Hrkic, J. Hu, S. T. Quay, and C. N. Sze, "Porosity-aware buffered Steiner tree construction," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 23, no. 4, pp. 517–526, Apr. 2004.

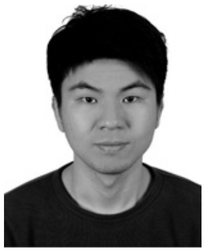
[17] C. J. Alpert, M. Hrkic, J. Hu, and S. T. Quay, "Fast and flexible buffer trees that navigate the physical layout environment," in *Proc. Des. Automat. Conf.*, 2004, pp. 24–29.

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

[19] J. Rubinstein, P. Penfield, and M. A. Horowitz, "Signal delay in RC tree networks," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 2, no. 3, pp. 202–211, Jul. 1983.

[20] A. Guttman, "R-Trees: A dynamic index structure for spatial searching," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 1984, pp. 47–57.

[21] N. H. E. Weste and D. Harris, "CMOS VLSI design: A circuit and systems perspective," 3rd ed. Reading, MA: Addison-Wesley, 2010.



Yen-Hung Lin (S'11) received the B.S. degree in computer science and information engineering from the National Chung Cheng University, Chiayi, Taiwan, in 2005. He is currently pursuing the Ph.D. degree at National Chiao Tung University, Hsinchu, Taiwan.

Since March 2011, he has been a Visiting Scholar/Student with the University of Texas at Austin, Austin. His current research interests include physical design automation, detailed routing, design for manufacturing, timing optimization, and formal verification.

Shu-Hsin Chang received the B.S. and M.S. degree from National Chiao Tung University, Hsinchu, Taiwan, in 2006 and 2008, respectively, all in computer science.

In 2008, he was employed at MediaTek Corporation, Hsinchu. His current research interests include intersection of the general fields of computer science and engineering and brain science.

Yih-Lang Li (M'04) received the B.S. degree in nuclear engineering in 1987 and the M.S. and Ph.D. degrees in computer science in 1990 and 1996, respectively, all from National Tsing-Hua University, Hsinchu, Taiwan.

In February 2003, he joined the faculty of the Department of Computer Science, National Chiao-Tung University (NCTU), Hsinchu, where he is currently an Associate Professor. Prior to joining the faculty of NCTU, he was a Software Engineer from 1995 to 1996 and an Associate Manager from 1998 to 2003 with Springsoft Corporation, Hsinchu, where he got heavily involved in the development of verification and synthesis tools for custom-based layout. His current research interests include physical synthesis, parallel architecture, and very large scale integrated testing.