Reclaiming Over-the-IP-Block Routing Resources With Buffering-Aware Rectilinear Steiner Minimum Tree Construction

Yilin Zhang[1], Ashutosh Chakraborty[2], Salim Chowdhury[2] and David Z. Pan[1]
[1] Department of ECE, University of Texas at Austin, Austin, TX, USA {yzhang1, dpan}@cerc.utexas.edu
[2] Oracle, Austin, TX, USA {ashutosh.chakraborty, Salim.Chowdhury}@oracle.com

***Abstract*—In this paper, we study an often overlooked but very important and practical problem of building Buffering-aware Over-the-Block rectilinear Steiner minimum tree (BOB-RSMT). In most previous works, the routing resources over the IP blocks were simply treated as routing blockages, resulting in significant waste of routing resources on higher metal layers not utilized by internal intra-block routing. On the other hand, routing over large IP blocks needs special attention as there is no way to insert buffers inside hard IP blocks, which can lead to unresolvable slew/timing violations. In this paper, we propose a novel BOB-RSMT algorithm which helps reclaim the "wasted" over-the-block routing resources while meeting user-specified slew constraints. Our algorithm incrementally and efficiently migrates initial tree structures with buffering-awareness to meet slew constraints while minimizing wire length. It can handle complex blocks including rectilinear shapes. Our experiments on various benchmarks demonstrate very promising results. By utilizing over-the-block routing resources intelligently, we can save the outside-block wire length as well as the total wire length significantly compared with the conventional obstacle-avoiding rectilinear Steiner minimum tree (OA-RSMT) algorithms. BOB-RSMT also reduces the repeater count/area needed to satisfy slew constraints, which is very important for modern design closure.**

I. Introduction

As the semiconductor technology scales into deeper sub-micron domain, billions of transistors can be used on a single system-on-chip (SOC). Routing becomes more and more challenging because of congestion, power, timing and buffering requirements. Rectilinear Steiner minimum tree (RSMT) construction is a fundamental physical design problem to achieve routing and buffering quality. This classical problem has long been proved as NP-complete [18] and many works have been performed including recent breakthrough, e.g. [8].

Because of extensively using IP-blocks to shorten turn around time, SOC designs nowadays are packed with IP blocks or macros. RSMT construction avoiding these blockages is well known as the OA-RSMT problem. OA-RSMT problem has been studied actively in the last few years (e.g., [2], [13], [15], [16]). Early approaches [15], [16] only deal with rectangular blockages, while a most recent study [13] can tackle rectilinear blockages without dissecting rectilinear blockages into rectangular ones. This approach can eliminate the infeasible solutions which put wires and buffers between adjoining blocks. However, all these OA-RSMT algorithms simply treat IP blocks as routing blockages, which would significantly waste routing resources over these IP blocks and cause more congestion issues.

Indeed, most IP blocks such as SRAMs only use certain lower metal layers. There are still considerable amount of routing resources available at higher metal layers over these IP blocks, even if we take into consideration the resources reserved for power/ground and clock routing. If we simply treat the IP blocks as routing obstacles, these over-the-block routing resources will be wasted, which leads to more routing demand elsewhere.

Besides blockage avoidance, other layout constraints are considered in [4], [5], [11], [12], [19], [21]. [4], [11], [19], [21] take timing, buffering, etc., into consideration in their tree construction. But slew constraint is not fully touched upon. It is reported that in reality, *slew mode buffering* is more predominant than timing mode buffering [12], [20]. Only a fraction (roughly $5\% \sim 10\%$) of nets needs to be buffered for delay optimization while for the remaining (roughly $90\% \sim 95\%$) are sufficient with slew mode buffering to meet the slew constraints. [5] extends the work in [4] with slew in consideration. However, the slew constraints are translated as length constraints, which may not guarantee meeting strict slew tolerances. [12] considers slew mode buffering and adopts the blockage avoidance algorithm in [3], [10] to benefit slew. But this approach either puts a Steiner node stationary in block or completely moves it out of block. This might bring unnecessary wiring detours and high buffering cost.

The blockage avoidance approach in [12] is shown by a 3-pin net example in Fig. 1(a). S is the source and A, B are the sinks; moving the Steiner node to right leads to the minimum-cost solution. Fig. 1(b) shows the same net if BOB-RSMT is adopted. In this case, BOB-RSMT saves two buffers as well as some detour wire length because it changes the structure of inside tree more efficiently. Buffer-aware tree construction has advantage over methods of tree construction which are independent of buffering.

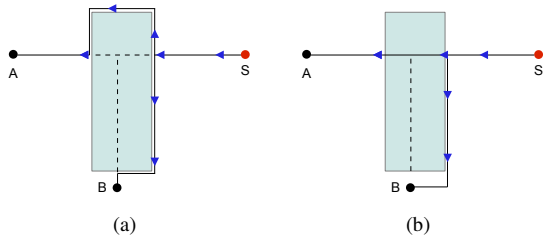


Fig. 1. A motivational example compares [12] and our proposed BOB-RSMT, which saves wire length and buffers.

In this paper, we propose to study a new class of BOB-RSMT problem and develop an effective algorithm which trys to intelligently reclaim the "wasted", over-the-IP-block routing resources by previous approaches while ensuring slew constraints for high quality buffering. Our algorithm incrementally updates the initial RSMT structure obtained from FLUTE [8] to satisfy slew constraints while minimizing wire length (FLUTE is chosen to be the initial RSMT generator because its low runtime and high quality). A *restricted length, over-the-block maze routing* algorithm is developed to reconnect any part of BOB-RSMT which is dissected during the optimization process. Our work has the following major contributions:

This work is partially supported by Oracle.

IEEE/ACM International Conference on Computer-Aided Design (ICCAD) 2012, November 5-8, 2012, San Jose, California, USA

1) This is the first work targeting this kind of BOB-RSMT problem. Our algorithm is able to integrate with a buffering tool to generate a low buffering cost BOB-RSMT without violating maximum slew constraint, which can be used in floorplanning, placement and routing stages. An incremental approach of fixing slew violation one by one is used to satisfy slew constraints on over-the-block part of BOB-RSMT.
2) Wire length outside blocks of our BOB-RSMT is remarkably less comparing with that in other algorithms which are not utilizing over-the-block routing resources. This will result in better timing, less power consumption and alleviate routing congestion. The total wire length, which includes the inside wire length as well, is also shorter than the results from OA-RSMT algorithms.
3) We formulate the incremental slew improvement problem into an integer linear programming (ILP) problem, which can be solved very fast as the number of variables are small.
4) A block-aware maze router is proposed to reconnect any part of BOB-RSMT dissected during the tree structure optimization.

The rest of paper is organized as follows. We first formulate the BOB-RSMT problem and analyze three optimization *primitives* in Section II. Our incremental approach of tree structures optimization will be presented in Section III, which includes five subsections. Section III-A discusses about how to find *possible point set*. Section III-B gives a method of shrinking search space. Section III-C formulates and solves the problem. Section III-D introduces a block-aware maze router to reconnect any dissected part of the tree. Section III-E describes a buffer insertion algorithm. Experimental results will be shown in Section IV, followed by conclusions in Section V.

II. Problem Formulation

A. What is BOB-RSMT?

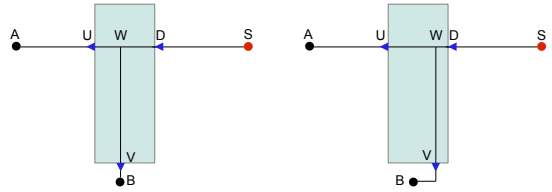
In this paper, we propose a new class of RSMT which utilizes the routing resource over the IP-blocks to improve wire length and congestion. In a two-dimensional routing region, we are given a net with a set of pins $P = \{p_1, p_2, \ldots, p_n\}$. Let $B = \{b_1, b_2, \ldots, b_m\}$ be a set of non-overlapping rectilinear blocks in the $2-$dimensional space. For $\forall\ p_r \in P$, p_r is not inside the $2-$dimensional space occupied by B. Any area with high-density placed logic cells is not allowed for buffering is also taken as buffering blockage into B.

Our algorithm constructs BOB-RSMT to connect all the pins in P. BOB-RSMT might intersect with blocks in B, which confine a set of trees $T = \{T_1, T_2, \ldots, T_l\}$ inside blocks. We call trees in T *inside trees*. The outside-the-block part of BOB-RSMT is defined as T_0. For each inside tree $T_i \in T$, the leaf nodes of T_i are on the boundaries of a block. Among all leaf nodes, one must be driving the signal and others are receiving. We name these leaf nodes which receive signals *escaping points* (EP), and the set of escaping points for T_i is $EP^i = \{EP^i_1, EP^i_2, \ldots, EP^i_{|EP^i|}\}$, in which $|EP^i|$ is the number of escaping points in EP^i. We denote the driver by D^i.

B. Basic Ideas and Optimization Primitives

For any inside tree $T_i \in T$, the worst slew part would occur at escaping points because no buffer is allowed to be inserted over the block. The best that a buffering tool can do to carry signal over the block to escaping points is to put the strongest buffer at D^i and a bunch of smallest buffers at EP^i to shield downstream capacitance. If for any $j, slew^i_j$ is still worse than $slew^i_{spec}$, then the slew from D^i to EP^i_j violates maximum slew constraint, which means that no buffering solution can be generated anyway. Further, because we want to leave more margin for buffering tool at critical timing path and buffer placement aspects, we use a middle size *hypothetical buffer* at D^i and middle size *hypothetical buffers* at EP^i to judge if thus the escaping points have slew violation. Using middle size hypothetical buffers instead of two extreme sizes will weaken the capability of utilizing more over-the-block routing resources, but the former will be a more practical assumption and leads to less buffering cost because more solutions can propagate through this inside tree. If any escaping point EP^i_j driven by a hypothetical buffer has $slew^i_j$ worse than $slew^i_{spec}$, then this EP^i_j is called *illegal escaping point*. Any inside tree with at least one illegal escaping point is an illegal inside tree.

In order to legalize any illegal inside tree, we will change positions of its escaping points as well as inside Steiner nodes. We move escaping points closer to the driver and then update the positions of corresponding Steiner nodes to improve slew. Fig. 2(a) is a three-pin net with source S and sinks A and B. Fig. 2(b) is the updated tree after a parallel sliding of escaping point V. Comparing Fig. 2(b) to Fig. 2(a), the downstream capacitance from W is closer to driver point due to the parallel sliding of V. The less capacitance burden to the driver reduces the slew on both escaping points U and V.



(a) (b)

Fig. 2. V moves to right in (b) compared to (a). This parallel sliding is providing slew improvement for escaping points U and V.

In this paper, we adopt the following PERI model for slew calculation at the escaping points [14]:

$$S(v_j) = \sqrt{S(v_i)^2 + S_{step}(v_i, v_j)^2} \tag{1}$$

$S(v_j)$ is slew at any node v_j, which is the root-mean square of the *step slew* from v_i to v_j and *output slew* at node v_i. The experimental results in [14] shows the error of PERI is within 1%, which is indistinguishable from what is obtained using SPICE simulation. For simplicity we use Bakoglu's metric [6] for step slew calculation:

$$S_{step}(v_i, v_j) = \alpha * Elmore(v_i, v_j), \alpha = ln9 \tag{2}$$

The combination of Bakoglu's metric and the PERI model is shown to have error within 4% [14]. It is, in general, accurate enough for RSMT construction purpose. If needed, we can use more accurate slew calculation tool for BOB-RSMT.

In this paper, we propose three slew optimization *primitives* including *parallel sliding*, *perpendicular sliding* and *EP merging* to improve the slew. The proposed primitives could guide illegal inside trees to migrate into legal ones with minimum wire length increase. The analysis demonstrates that the capability of using these three primitives can fix slew violations under any $slew_{spec}$.

We first analyze parallel sliding which performs sliding to a new position on one of the block boundaries. As the escaping point sliding on the boundary, if its first upstream Steiner node ancestor can also slide to keep the wire segment between escaping point and the ancestor Steiner node in translation, then this sliding on the boundary is called parallel sliding. The requirement of a meaningful parallel sliding is that the sliding should shorten the length of path from the

escaping point to D^i, i.e., sliding the escaping point closer to the driver.

The example in Fig. 3(a) provides an inside tree with the driver D and escaping points A, B, C, E. Fig. 3(b) shows that escaping point A performs a parallel sliding by a distance of Δl to new position A'. There will be a reduction of step slew on escaping point A and B. We adopt the following notations in Table I to calculate the slew improvement of parallel sliding in this example. The step slew reduction on A and B from Bakoglu's metric model will be:

$$\delta_A = -\alpha * r\Delta l(C_t(U) + 0.5 * c\Delta l)$$
$$\delta_B = -\alpha * r * \Delta l * (C_b + c * l(A, U))$$

The output slew of the driver D remains unchanged since the total downstream capacitance of the inside tree is the same. Then we can use (1) to calculate the corresponding slew change on escaping point A and B. The changes in slew of escaping point C and E are both zero because U is not on the path from these two escaping points to the driver D.

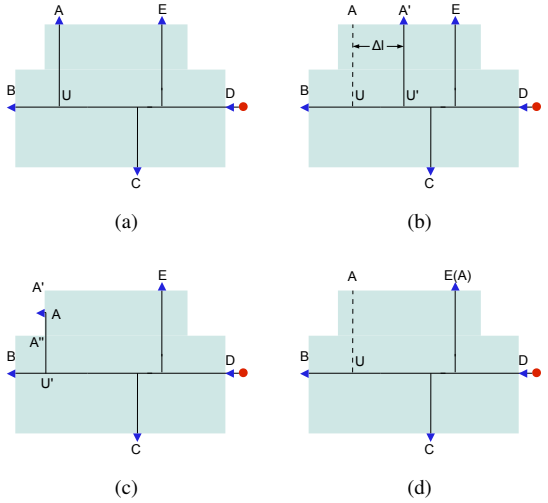


Fig. 3. An example shows slew reduction from three primitives. (b) shows escaping point A slides to A' parallelly to improve slew on A and B. (c) shows the vertical sliding of A from A' to A''. (d) shows EP merging of escaping point A to E.

TABLE I
NOTATION OF VARIABLES

r	unit length wire resistance on chosen layer
c	unit length wire capacitance on chosen layer
R_b	chosen buffer output resistance
C_b	chosen buffer input capacitance
$l(U, V)$	length of edges between node U and V
$C_t(V)$	total capacitance of the sub-tree rooted at node V down to the nearest downstream buffer, including the buffer input capacitance

With parallel sliding we can decrease slew at escaping points, but we may have wire length penalty because the position change of escaping points may need some additional wire connection from outside-the-block sub-tree. In the example shown in Fig. 3(a) to Fig. 3(b), escaping point A moves a distance of Δl to A' by a parallel sliding. The penalty of wire length is at most Δl because the outside connection to A' can go through A along the edge from A to A' with Δl more wire length and there is no change in wire length of inside tree.

Algorithm 1 *The overall BOB-RSMT Algorithm*

Input: Initial inside trees T, Slew required for the net: $slew_{spec}$
Output: BOB-RSMT
1: **for** each T_t **do**
2: Sort EP^t in descending order of slew
3: **while** $slew_1^t > slew_{spec}$ **do**
4: Build possible point set for all unfixed EP in EP^t
5: Formulate the problem by a ILP
6: Solve the ILP and update T_t
7: Remove EP_1^t from EP^t
8: **end while**
9: **end for**
10: **return** BOB-RSMT

Besides parallel sliding, we perform perpendicular sliding on edges which are not considered as parallel sliding edges. In Fig. 3(c), if A is sliding on the segment between A' to A'', the wire length penalty will be zero during the whole sliding process because slide of A from A' to A'' is just slipping wire from inside block to outside. It is observed that as A reaches A'', all escaping points will have the largest slew improvement due to the least downstream capacitance from U'. The calculation of slew reduction is similar as of parallel sliding.

Complementary to parallel sliding and perpendicular sliding, EP merging removes one escaping point and all edges from this escaping point up to the first Steiner point ancestor in the inside tree. This will also bring down the slew of all escaping points based on the fact that this escaping point and the upstreaming edges from it to next Steiner point in the inside tree will be removed. The above process will reduce the total capacitance burden of the driver and hence improve slew for all escaping points.

In tree T_i, if EP merging joins one EP_i with another EP_j, the outside connection to EP_i will be reconnected to EP_j or other closer part of BOB-RSMT by a restricted length, over-the-block maze routing algorithm, which will be introduced in Section III-D.

Considering the EP merging of escaping point A to E in Fig. 3(a) and Fig. 3(d), the wire length penalty will be at most the distance between A and E because the outside connection to A can go through original position of A and then along the edge to E as shown in Fig. 3(d). Actually due to the existence of tree outside this block, reconnecting to the outside part might have less wire length penalty. But here, we take the previous conservative estimate as the wire length penalty because it is guaranteed to be achieved. The calculation of slew reduction is similar.

III. BOB-RSMT Algorithms

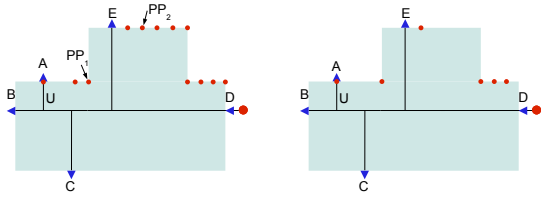
To construct a legal BOB-RSMT, we first generate an initial RSMT by using FLUTE-3.1, and then we apply primitives to all illegal inside trees to fix the slew of them. Finally a proposed restricted length, over-the-block maze routing algorithm is used to reconnect all these parts to form the final BOB-RSMT. The approach is described in Algorithm 1.

For each $T_t \in T$ as an illegal inside tree, three primitives are applied to decrease slew on illegal escaping points until T_t becomes a legal inside tree. The procedure starts from calculating slew of each EP_i^t. From the calculated result, we first sort EP^t in descending order of their slew violations as line 2 of Algorithm 1. Then we choose the first illegal escaping point, EP_1^t, which should have worst slew violation based on the sorting. To improve slew for EP_1^t, each escaping point from $\{EP_1^t, EP_2^t, \ldots EP_{|EP^t|}^t\}$ might slide to a different position by taking a combination of primitives discussed in section II-B. Taking these optimization primitives guarantees $slew_1^t$ to be within slew requirement. Because in the extreme situation where

maximum slew constraint is zero EP_1^t can still become legal escaping point by merging one escaping point to another until only the driver is left. This slew fixing procedure is elaborated through line 4 to 6 of Algorithm 1.

After $slew_1^t$ has decreased below the required slew, EP_1^t is fixed at the current position and removed from EP_t as in line 7. Next iteration will start from the rest of EP^t. The current iteration will not degrade the result of previous iterations as we will remove solution space from current solution space if it degrades slew of fixed escaping points. This solution space elimination happens rarely because moving one escaping point closer to driver usually does not degrade slew on other points. This slew improvement method will keep being applied on EP_1^t at each iteration until all EP^t are fixed.

A. Generating Possible Point Set



(a) (b)

Fig. 4. (a) is an inside tree with driver at D. It shows all possible points for E. (b) exhibits the refined possible point set for E.

At every iteration we try to improve the $slew_1^t$ of inside tree $T_t \in T, t \in \{1, 2, \ldots, l\}$. One important step in that is to generate *possible point set* for each non-fixed escaping points. Possible point set is a set of all *possible points* of one non-fixed escaping point. Each possible point in the set is a point on a boundary edge where escaping point might end up. For any non-fixed $EP_i^t \in \{EP^t\}$, the j^{th} possible point associated with EP_i^t is denoted as PP_{ij}. PP_{ij} is stored in a 3-tuple format $\{E_{ij}, B_{ij}, W_{ij}\}$. E_{ij} and B_{ij} denote the step slew at EP_1^t and output slew reduction of the driver if EP_i moves to PP_{ij}. W_{ij} is the corresponding estimated wire length penalty. The possible point set associating with EP_i^t in the current iteration is denoted as PPS_i^t. $PPS_i^t = \{PP_{i1}^t, PP_{i2}^t, \ldots, PP_{ir}^t\}$, where r is the number of possible points inside.

For each EP_i^t in current iteration, we generate the possible point for EP merging first. Assume the target escaping point for EP_i^t to merge with is EP_j^t. The estimated wire length penalty is the outside-the-block distance from EP_i^t to EP_j^t. Thus for EP merging, we always choose the EP_j^t with minimum outside-the-block distance from EP_i^t. The slew reduction and the estimated wire length penalty of this choice will be added to the PPS_i^t as the 3-tuple $\{E_{ij}, B_{ij}, W_{ij}\}$. For example in Fig.4(a), where EP_1^t is B and EP_i^t is E, the EP merging point for E is escaping point A.

Secondly, we consider the sliding for EP_i^t. We first search all edges on $block_t$ for sliding by the criteria discussed in SectionII-B. The $block_t$ here refers to the block confining T_t. Then for each parallel sliding edge, we chop it at a number of points. Moving EP_i^t to any one of these points can improve slew on EP_1^t. For each perpendicular sliding edge, we pick the possible point at one end of it, as discussed in SectionII-B. Each chop point is a possible point, which will be added into possible point set. We set distance between two chop points to be a fixed value depending on the scale of the chip. For example in Fig.4(b), D is the driver and A, B, C, E are escaping points. The possible point set for E are shown with red color dots.

TABLE II
NOTATION OF VARIABLES IN OUR FORMULATION

X_{ij}	binary variable denoting the choice of PPS_{ij}^t, $X_{ij} = 1$ if it is chosen, otherwise $X_{ij} = 0$
E_{ij}	step slew reduction at EP_1^t if EP_i^t moves to PPS_{ij}
B_{ij}	output slew reduction on D^t if EP_i^t moves to PPS_{ij}
W_{ij}	estimated wire length penalty of EP_1^t if EP_i^t moves to PPS_{ij}
Y_{rsij}	binary variable equals to one only if $X_{rs} = 1$ and $X_{ij} = 1$

B. Refinement of Possible Region Set

For any escaping point EP_i^t, after collecting PPS_i^t, we will do a refinement on $\forall PP_{ij} \in PPS_i^t$ to reduce the potential solution space. The refinement is based on Pareto efficiency [7].

The refined possible point set should form a Pareto frontier in the sense of estimated wire length penalty and slew reduction(both output slew reduction at the driver and step slew reduction at EP_1^t), which is restricting attention to the set of choices that either has less estimated wire length penalty or more slew reduction. After applying refinement on Fig.4(a), the possible points turn into Fig.4(b). One example of a pruned possible point in Fig.4(a) is: PP_2 is pruned by PP_1 as the latter has less estimated wire length penalty and more slew improvement.

C. Primitive Choice Based on a Fast ILP

In order to construct the inside tree under the slew constraint with minimum wire length as target, $\forall EP_i^t \in EP^t$ we need to decide which possible point to choose. We use an incremental way to update positions of all escaping points at each iteration. In each iteration, in order to meet the slew constraint for the worst violated escaping point, all escaping points in EP^t will move and the whole inside tree will be updated. Only through moving all $EP_i^t \in EP^t$ at the same time can we attain an optimal solution with minimum estimated wire length penalty. This stems from the reason that $\forall PPS_i^t, i \in \{1, 2, \ldots, |EP_i^t|\}$ has a Pareto frontier to choose one point from. The choice depends on what choices are made at other Pareto frontiers because the total slew reduction summed up from all these choices has to diminish the slew violation of EP_1^t. The new slew has to satisfy the slew constraints,

$$\sqrt{(S_{step1}^t + \sum_{i=1}^{|EP_i^t|} E_i^t)^2 + (S^t(D^t) + \sum_{i=1}^{|EP_i^t|} B_i^t)^2} < slew_{spec}^t$$

The simultaneous step slew reduction is same with calculating one by one, and the simultaneous output slew reduction is close enough to be represented by the summation of individuals. The simultaneous point choice problem can be formulated in an optimization problem as follows (notation in Table II):

$$\text{min.} \sum_{i=1}^{|EP^t|} \sum_{j=1}^{|PPS_i^t|} X_{ij} W_{ij} \tag{3}$$

$$\text{s.t.} (S_{step1}^t + \sum_{i=1}^{|EP^t|} \sum_{j=1}^{|PPS_i^t|} X_{ij} E_{ij}^t)^2 +$$

$$(S^t(D^t) + \sum_{i=1}^{|EP^t|} \sum_{j=1}^{|PPS_i^t|} X_{ij} B_{ij}^t)^2 <= {slew_{spec}^t}^2 \tag{3a}$$

$$\sum_{j=1}^{|PPS_i^t|} X_{ij} = 1 \qquad \forall i \in \{1, 2, \ldots, |EP^t|\} \tag{3b}$$

The objective function (3) is to minimize the total estimated wire length penalty. Constraint (3a) restricts that the total slew reduction

on EP_1^t has to be able to pull $slew_1^t$ down below requirement. Constraint ($3b$) is used to limit only one position chosen for each escaping point.

This formulation is a non-linear integer programming formulation (NLIP). We expand the step slew part in constraint ($3a$) as:

$$ {S_{step1}^t}^2 + 2S_{step1}^t \sum_{i=1}^{|EP^t|} \sum_{j=1}^{|PPS_i^t|} (X_{ij}^t)(E_{ij}^t) + \sum_{r=1}^{|EP^t|} \sum_{s=1}^{|PPS_i^t|} \sum_{i=1}^{|EP^t|} \sum_{j=1}^{|PPS_i^t|} X_{rs}^t E_{rs}^t X_{ij}^t E_{ij}^t $$

We observe that the only quadratic item is $X_{rs}^t X_{ij}^t$. We can substitute this item for a new binary variable Y_{rsij}. We constrain Y_{rsij} such that Y_{rsij} always behaves same as $X_{rs}^t X_{ij}^t$. The constraint needed is (for output slew part, it is similar):

$$ Y_{rsij} <= X_{rs}^t \quad \forall r,s,i,j \in \{1,2,\ldots,|EP^t|\} \quad (3c) $$
$$ Y_{rsij} <= X_{ij}^t \quad \forall r,s,i,j \in \{1,2,\ldots,|EP^t|\} \quad (3d) $$
$$ Y_{rsij} >= X_{rs}^t + X_{ij}^t - 1 \; \forall r,s,i,j \in \{1,2,\ldots,|EP^t|\} \quad (3e) $$

By adding constraint ($3c$) $\sim$ ($3e$) to (3), we turn the NLIP problem into integer linear programming formulation (ILP), which can be solved by solver Gurobi Optimizer [1] quickly. The formulation of ILP is shown as follows:

$$ \text{min.} \sum_{i=1}^{|EP^t|} \sum_{j=1}^{|PPS_i^t|} X_{ij} W_{ij} $$

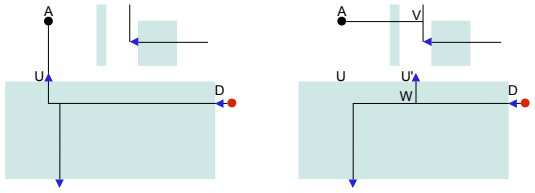
$$ \text{s.t.} {S_{step1}^t}^2 + S^t (D^t)^2 + 2S_{step1}^t \sum_{i=1}^{|EP^t|} \sum_{j=1}^{|PPS_i^t|} (X_{ij}^t)(E_{ij}^t) + 2S^t(D^t) \sum_{i=1}^{|EP^t|} \sum_{j=1}^{|PPS_i^t|} (X_{ij}^t)(B_{ij}^t) + \sum_{r=1}^{|EP^t|} \sum_{s=1}^{|PPS_i^t|} \sum_{i=1}^{|EP^t|} \sum_{j=1}^{|PPS_i^t|} (B_{rs}^t B_{ij}^t + E_{rs}^t E_{ij}^t) Y_{rsij} <= {slew_{spec}^t}^2 $$

$$ \sum_{j=1}^{|PPS_i^t|} X_{ij} = 1 \quad \forall i \in \{1,2,\ldots,|EP^t|\} $$
$$ Y_{rsij} <= X_{rs}^t \quad \forall r,s,i,j \in \{1,2,\ldots,|EP^t|\} $$
$$ Y_{rsij} <= X_{ij}^t \quad \forall r,s,i,j \in \{1,2,\ldots,|EP^t|\} $$
$$ Y_{rsij} >= X_{rs}^t + X_{ij}^t - 1 \; \forall r,s,i,j \in \{1,2,\ldots,|EP^t|\} $$

Due to the number of choices for each escaping point is limited by the number of possible sliding edges and their length, the total number of variables in our formulation is very limited. The ILP solver can get the solution very fast.

D. Block-aware Maze Routing Algorithm

After final positions of all escaping points are fixed, a restricted length, over-the-block maze routing will be applied. This maze routing features ability of routing over-the-blockage. The maximum length it can route over the block is decided by the distance a middle size buffer could drive itself over the block without slew problem. This restricted length, over-the-block maze router requires less wire length comparing with normal maze router because of its ability to route over the block. In Fig. 5(a), U is an escaping point and A is a sink of the tree. Escaping point U slides to U' to legalize the inside tree. The restricted length, over-the-block maze is applied to reconnect A, and it will choose connection from A to V instead of from A to U' because of shorter wire length, resulting in Fig. 5(b). Wire segment U' to W will be removed if no other part connects to U'.



(a) (b)

Fig. 5. Restricted length, over-the-block maze routing find a shortest path to reconnect pin A

The implementation of block-aware maze routing is based on the normal maze routing. But its multiple points to multiple points search is from all points of the tree rooted at the current escaping point to T_0 or an escaping point of any inside tree. Furthermore, the length of the over-the-block path is checked every step in the search to make slew safe. The details of the algorithm are skipped here due to page limit.

E. Min-cost Slew Mode Buffer Insertion

After BOB-RSMT is fully constructed, we insert buffers in a free-location way, which allows buffers at any unblocked space. Comparing with fixed-location buffer insertion algorithm, free-location buffering can freely choose position for buffering, which will result in lower buffer cost. We assume the input slew of each buffer is fixed at the slew constraint. Free-location buffering with fixed input slew will give a shorter runtime but conservative result [12]. It uses a dynamic programming framework to propagate a set solutions from bottom up to the source of the net. Each solution is characterized as a triple (C, W, S), where C stands for downstream capacitance, W denotes the total cost of the solution, and S is the worst downstream accumulated step slew degradation calculated from (2). Consider to propagate a solution from a node v_j to its parent v_i through edge $e = (v_i, v_j)$. One solution γ_j at node v_j propagates to v_i to become a solution γ_i as $C(\gamma_i) = C(\gamma_j) + C_e, W(\gamma_i) = W(\gamma_j), S(\gamma_i) = S(\gamma_j) + S_{step}(v_i, v_j)$.

In addition to unbuffered propagation, a buffer can be placed at v_i to generate a buffered solution. If a buffer is placed, the buffered solution at v_i is becoming $C(\gamma_{i,buf}) = C_b, W(\gamma_{i,buf}) = W(\gamma_i) + W_b, S(\gamma_{i,buf}) = 0$.

When two sets of solutions propagated by both left and right children reach a branching node, these two set of solutions are merged. The merge is performed on each solution in left child with each solution in right child. Assume γ_l is one solution from left side and γ_r is one solution from right side to be merged. The merged solution γ_p will have $C(\gamma_p) = C(\gamma_l) + C(\gamma_l), W(\gamma_p) = W(\gamma_l) + W(\gamma_r), S(\gamma_p) = \max\{S(\gamma_l), S(\gamma_r)\}$.

It is beneficial to prune useless solution at each node. As two solutions γ_{i_1} and γ_{i_2} are at same node, γ_{i_1} dominates γ_{i_2} only if $C(\gamma_{i_1}) <= C(\gamma_{i_2}), W(\gamma_{i_1}) <= W(\gamma_{i_2}), S(\gamma_{i_1}) <= S(\gamma_{i_2})$.

IV. Experimental Results

We have implemented our algorithm in the C++ programming language. The experiments are conducted on an Intel Core 3.0GHz

TABLE III
COMPARISONS BETWEEN OUR PROPOSED BOB-RSMT AND OA-RSMT

Bench -marks	n	m	20% slew			50% slew			80% slew			FLUTE -3.1	Huang [13]	Ajwani [2]
			WL_i	WL_o	WL	WL_i	WL_o	WL	WL_i	WL_o	WL			
RT1	10	500	385	1449	1834	296	1522	1818	296	1522	1818	1817	2146	2191
RT2	50	500	1216	43469	44685	1216	43469	44685	1186	43507	44693	44685	45852	48156
RT3	100	500	263	7420	7683	276	7390	7666	282	7379	7661	7652	7964	8282
RT4	100	1000	1196	6647	7843	872	6957	7829	882	6947	7829	7827	9693	10330
RT5	200	2000	6702	36474	43176	7277	35720	42997	7491	35458	42949	42943	51313	54598
RC1	10	10	740	24550	25290	740	24550	25290	740	24550	25290	25290	25980	25980
RC2	30	10	5220	36998	42218	8190	34520	42710	8190	33020	41210	39920	41350	42110
RC3	50	10	530	53950	54480	1190	53290	54480	4480	48430	52910	52880	54160	56030
RC4	70	10	3030	52420	55450	4490	50960	55450	5420	50027	55447	55300	59070	59720
RC5	100	10	3590	69810	73400	3590	69810	73400	4750	68980	73730	73220	74070	75000
RC6	100	500	12983	65667	78650	14613	61980	76593	15049	62432	77481	77171	79714	81229
RC7	200	500	13141	97109	110250	13785	95162	108947	14244	93565	107809	106743	108740	110764
RC8	200	800	23674	88136	111810	25515	84049	109564	25184	83385	108569	108495	112564	116047
RC9	200	1000	25689	83972	109661	25689	83972	109661	26026	82192	108218	107729	111005	115593
RC10	500	100	8372	156348	164720	9400	155370	164770	9400	155370	164770	163980	164150	168280
RC11	1000	100	3016	229519	232535	3498	228232	231730	3498	228282	231780	231730	230837	234416
Ave.			9.95	91.38	101.33	10.99	89.98	100.97	11.69	88.74	100.43	100	106.03	108.17

TABLE IV
CPU RUNTIME

Bench -marks	CPU (s)											
	20% slew				50% slew				80% slew			
	ILP	maze routing	C-SB buffering	BOB-RSMT buffering	ILP	maze routing	C-SB buffering	BOB-RSMT buffering	ILP	maze routing	C-SB buffering	BOB-RSMT buffering
RT1	0.02	0.1	270.09	0.03	0	0.04	281.62	0.02	0	0.04	285.03	0.01
RT2	0	0.13	1041.83	0.03	0	0.04	1056.27	0.07	0.01	0.1	1059.6	0.04
RT3	0.02	0.13	905.05	0.26	0	0.07	1023.39	0.19	0	0.07	1041.01	0.15
RT4	0.02	0.25	2859.06	0.64	0	0.07	2880.2	0.47	0	0.07	2896.48	0.43
RT5	0.03	2.61	>7200	1.25	0.01	1.43	>7200	1.03	0.01	0.35	>7200	1.03
RC1	0	0.01	0.05	0	0	0.01	0.05	0	0	0	0.05	0
RC2	0	0	0.45	0.01	0	0	0.53	0	0	0.01	0.63	0
RC3	0	0.02	0.50	0.01	0	0.03	0.66	0.01	0	0.01	0.63	0.01
RC4	0.01	0.04	2.72	0.01	0	0.01	2.40	0.01	0.02	0	2.40	0.01
RC5	0	0.01	4.44	0.02	0.01	0.01	4.44	0.02	0.01	0.01	4.50	0.02
RC6	0.04	0.91	1652.4	0.2	0.01	0.08	1643.11	0.12	0	0.09	1634.05	0.13
RC7	0.04	3.43	>7200	0.36	0.02	1.51	>7200	0.29	0	0.56	>7200	0.24
RC8	0.05	2.24	>7200	0.05	0.01	0.76	>7200	0.56	0.01	0.26	>7200	0.55
RC9	0.09	5.08	>7200	0.78	0.03	1.88	>7200	0.55	0.02	1.02	>7200	0.4
RC10	0.02	0.31	190.1	0.44	0	0.06	191.3	0.39	0	0.05	190.39	0.38
RC11	0.04	0.55	592.35	1.24	0.01	0.46	589.87	1.15	0.01	0.37	591.71	1.15

TABLE V
EXTRA BUFFERING COST COMPARISON

Bench -marks	20% slew					50% slew					80% slew				
	FLU -TE	[12]	extra % of [12]	BOB -RSMT	extra % of BOB-RSMT	FLU -TE	[12]	extra % of [12]	BOB -RSMT	extra % of BOB-RSMT	FLU -TE	[12]	extra % of [12]	BOB -RSMT	extra % of BOB-RSMT
RT1	151	221	46.36	158	4.63	94	137	45.74	98	4.26	74	109	33.78	77	4.05
RT2	349	409	17.19	352	0.86	218	225	3.21	218	0.00	16	19	18.75	16	0.00
RT3	761	897	17.87	767	0.79	470	557	18.51	473	0.64	376	442	17.55	376	0.00
RT4	300	448	49.33	306	2.00	180	270	50.00	184	2.22	139	208	49.64	144	3.60
RT5	378	673	78.04	386	2.12	228	413	81.14	237	3.95	174	326	87.36	183	5.17
RC1	21	23	9.52	22	4.76	13	14	7.70	14	7.70	10	11	10.00	10	0.00
RC2	33	37	12.12	33	0.00	21	24	14.29	21	0.00	16	19	18.75	16	0.00
RC3	96	113	17.71	99	3.12	59	71	20.34	60	1.69	45	54	20.00	47	4.44
RC4	65	76	16.92	67	3.08	41	48	17.07	41	0.00	30	36	20.00	32	6.67
RC5	62	70	12.90	62	0.00	36	41	13.89	39	8.33	28	32	14.29	29	3.57
RC6	237	272	14.77	248	4.64	143	166	16.08	151	5.59	112	128	14.29	118	5.36
RC7	458	504	10.04	465	1.53	278	307	7.67	287	3.24	217	242	11.52	224	3.23
RC8	282	342	21.28	304	7.80	172	211	22.67	187	8.72	135	162	20.00	143	5.92
RC9	425	506	19.06	451	6.12	259	313	20.85	278	7.34	202	247	22.28	213	5.45
RC10	394	412	5.08	395	0.25	246	261	6.10	248	0.81	189	201	6.35	191	1.06
RC11	1662	1695	1.99	1670	0.48	1023	1044	2.05	1025	0.20	789	804	1.90	792	0.38
Ave.			21.89		2.64			21.71		3.42			22.90		3.06

Linux machine with 32GB memory. We choose Gurobi Optimizer 4.60 as our solver for the integer linear programming.

RT1-RT5 and RC01-RC11 are benchmarks in our experiments. IND1-IND5 used in [2], [13] are not used in our experiments, because they require routing/buffering between adjoining blocks, which might be infeasible for real designs. RT1-RT5 are randomly generated circuits used in [17]. RC01-RC11 are test cases used in [9]. Because these benchmarks are widely different in scale and do not carry timing and physical information, we first apply predetermined resistance and capacitance to all of them. We use different resistance and capacitance for horizontal and vertical wires respectively. If a congestion map is considered, we can assign each wire segment to a proper layer by pruning possible points in congestion.

For each benchmark, after FLUTE-3.1 finishes generating inside trees, we collect slew on every escaping point for all inside trees. The range of value of collected slew is $[slew_{min}, slew_{max}]$. Then

we test each benchmark under three slew constraints:

1) 20% slew: $slew_{min} + 20\%(slew_{max} - slew_{min})$
2) 50% slew: $slew_{min} + 50\%(slew_{max} - slew_{min})$
3) 80% slew: $slew_{min} + 80\%(slew_{max} - slew_{min})$.

These three tests of each benchmark can test the performance of our algorithm under tight, medium and loose slew constraints, respectively.

Table III compares the performance of our algorithm with some recently published OA-RSMT algorithms. Columns 4, 5, 6 list the over-the-block wire length, outside-the-block wire length and total wire length of our algorithm under 20% slew constraint. Columns 7 to 12 are for same types of wire length under 50% and 80% slew constraints. The row at bottom illustrates the average performance from all benchmarks listed above. We normalize the performance in such a way that the total wire length of FLUTE-3.1 is 100. The outside-the-block wire length from Huang [13] is 14.27% more than our BOB-RSMT algorithm under 20% slew constraint and 17.29% under 80% slew constraint. The free over-the-block routing resources reclaimed by BOB-RSMT are between 10% to 12% of the total wire length. Even for the total wire length, since BOB-RSMT can intelligently use over-the-block wires, it can reduce about 5% of total wire length compared with [13] and [2]. The runtime of our proposed algorithm BOB-RSMT is divided into two parts: solving ILP and block-aware maze routing, which are listed in the columns 2, 3, 6, 7, 10, 11 in Table IV. Our runtime is much shorter than both reported in [13] and [2].

Table V carries out the buffering results on *FLUTE*, approach in [12] and *BOB-RSMT*. For simplicity we only use one type of buffer, and total buffering cost is the number of buffers used. From the table we have minimum buffering cost associated with $20\%, 50\%, 80\%$ slew constraint respectively for all benchmarks. We use buffering on FLUTE as the baseline for our comparison. Buffering on FLUTE is performed without considering any block in the two-dimensional routing region. We implement the approach in [12] and the results are in columns 3, 8, 13 in the Table V. The penalty parameter α for over-the-block routing wires in [3], [12] is set between 10 to 100, which increases if no solution can propagate to the source. Columns 5, 10, 15 in the Table V are the minimum buffering costs from BOB-RSMT. Columns after the buffering cost are the percentages of extra buffers used to overcome blocks by that approach. As we can see, buffering on BOB-RSMT only uses around 3% more buffers than FLUTE to propagate through blocks, while the approach in [12] uses more than 20%. The CPU runtime comparison between buffering on BOB-RSMT and the approach in [12] is in Table IV. Columns 5, 9, 13 illustrate the runtime for buffering on BOB-RSMT under three slew constraints while columns 4, 8, 12 are for approach in [12]. Buffering on BOB-RSMT is much faster because during the buffering stage, the tree structure of BOB-RSMT has no need to be changed to meet slew constraint, but in contrast [12] needs to find LeastBlockedPath([3], [10]) during every step.

V. Conclusion

In this paper, we study an important new class of RSMT problems, i.e., buffering-aware over-the-IP-block rectilinear Steiner minimum tree. We propose an effective and efficient algorithm which can reclaim the over-the-IP-block routing resources and is beneficial to buffering. With our proposed approach, we can reduce the outside-the-block wire length for more than 14% and use about 19% less buffer cost than the approach in [12] to ensure slew correct RSMT with blocks. Our proposed algorithm BOB-RSMT can be used in both pre-routing and global routing stage to provide high quality routing solutions. One example is to pre-route certain persistent critical signals in large complex chips, such as a microprocessor, using higher metal layers. Since this is the first work of this kind, we expect more follow-up works to push the state-of-the-art of BOB-RSMT, which is crucial for large SOC designs with many IP-blocks.

References

[1] Gurobi Optimizer 4.52. http://www.gurobi.com/.

[2] G. Ajwani, Chris Chu, and Wai-Kei Mak. FOARS: FLUTE Based Obstacle-Avoiding Rectilinear Steiner Tree Construction. In *Proc. ISPD*, 2010.

[3] Charles J. Alpert, G. Gandham, J. Hu, J.L. Neves, S.T. Quay, and S.S. Sapatnekar. Steiner tree optimization for buffers, blockages. and bays. In *Proc. IEEE Int. Symp. on Circuits and Systems*, 2001.

[4] C.J. Alpert, G. Gandham, M. Hrkic, J. Hu, S. T. Quay, and C. N. Sze. Porosity aware buffered steiner tree construction. *IEEE TCAD*, 23(4):517–526, 2003.

[5] C.J. Alpert, M. Hrkic, J. Hu, and S. T. Quay. Fast and flexible buffer trees that navigate the physical layout environment. In *Proc. DAC*, 2004.

[6] H. B. Bakoglu. Circuits, Interconnections, and Packaging for VLSI. Addison-Wesley, 1990.

[7] Stephen P. Boyd and Lieven Vandenberghe. Convex Optimization. Cambridge University Press, 2004.

[8] Chris Chu and Yiu-Chung Wong. FLUTE: Fast Loopup Table Based Rectilinear Steiner Minimal Tree Algorithm for VLSI Design. *IEEE TCAD*, 27(1):70–83, 2008.

[9] Zhe Feng, Yu Hu, Tong Jing, Xianlong Hong, Xiaodong Hu, and Guiying Yan. An O(nlogn) algorithm for obstacle-avoiding routing tree construction in the -geometry plane. In *Proc. ISPD*, 2006.

[10] Jiang Hu, C.J. Alpert, S.T. Quay, and G. Gandham. Buffer insertion with adaptive blockage avoidance. *IEEE TCAD*, 22(4):492–498, 2003.

[11] Jiang Hu and Sachin S. Sapatnekar. Simultaneous buffer insertion and non-Hanan optimization for VLSI interconnect under a higher order AWE model. In *Proc. ISPD*, 1999.

[12] Shiyan Hu, C.J. Alpert, J. Hu, S.K. Karandikar, Z. Li, W. Shi, and C.N. Sze. Fast algorithms for slew-constrained minimum cost buffering. *IEEE TCAD*, 26(11):2009–2022, 2007.

[13] T. Huang and Evangeline F.Y. Young. An Exact Algorithm for the construction of Rectilinear Steiner Minimum Trees among Complex Obstacles. In *Proc. DAC*, 2011.

[14] Chandramouli V. Kashyap, Charles J. Alpert, Frank Liu, and Anirudh Devgan. Closed Form Expressions for Extending Step Delay and Slew Metrics to Ramp Inputs. In *Proc. ISPD*, 2003.

[15] L. Li, Z. Qian, and Evangeline F.Y. Young. Generation of Optimal Obstacle-avoiding Rectilinear Steiner Minimum Tree. In *Proc. ICCAD*, 2009.

[16] L. Li and Evangeline F.Y. Young. Obstacle-avoiding Rectilinear Steiner Tree Construction. In *Proc. ICCAD*, 2008.

[17] Chung-Wei Lin, Szu-Yu Chen, Chi-Feng Li, Yao-Wen Chang, and Chia-Lin Yang. Efficient obstacle-avoiding rectilinear steiner tree construction. In *Proc. ISPD*, 2007.

[18] M.R.Garey and D.S.Johnson. The Rectilinear Steiner Tree Problem is NP-Complete. *Proceedings SIAM Journal on Applied Mathematics*, 32(4):826–834, 1977.

[19] Takumi Okamoto and Jason Cong. Simultaneous routing and buffer insertion with restrictions on buffer locations. In *Proc. ICCAD*, 1996.

[20] Peter J. Osler. placement driven synthesis case studies on two sets of two chips: hierarchical and flat. In *Proc. ISPD*, 2004.

[21] Hai Zhou, D. F. Wong, I-Min Liu, and Adnan Aziz. Buffered Steiner tree construction with wire sizing for interconnect layout optimization. In *Proc. DAC*, 1999.