Timing-Driven, Over-the-Block Rectilinear Steiner Tree Construction with Pre-Buffering and Slew Constraints

Yilin Zhang
ECE Department
University of Texas at Austin
Austin, TX 78712, USA
yzhang1@cerc.utexas.edu

David Z. Pan
ECE Department
University of Texas at Austin
Austin, TX 78712, USA
dpan@ece.utexas.edu

ABSTRACT

In this paper, we study a fundamental and crucial problem of building timing-driven over-the-block rectilinear Steiner tree (TOB-RST) with pre-buffering and slew constraints. We pre-characterize the tree topology and buffer distribution to provide accurate timing information for our final RST construction. In most previous work, the routing resources over the IP blocks were simply treated as routing blockages. Our TOB-RST could reclaim the "wasted" over-the-block routing resources while meeting user-specified timing (slack and slew) constraints. Before fixing topology, a topology-tuning is performed based on location of buffers to improve timing without increasing buffering cost. Experiments demonstrate that TOB-RST can significantly improve the worst negative slack (WNS) with even less buffering and wirelength compared with other slack-driven obstacle-avoiding rectilinear Steiner tree (SD-OARST) algorithms.

Categories and Subject Descriptors

B.7.2 [**Hardware,Integrated Circuit**]: Design Aids

General Terms

Algorithms, Design

Keywords

Over-the-block; Timing-driven RST; Slew Constraints

1. INTRODUCTION

As the semiconductor technology scales into deeper submicron domain, interconnection delay has become the dominant factor in determining circuit speed, contributing up to 50% $\sim$ 70% of the clock cycle in high performance circuit [8]. Rectilinear Steiner tree (RST) is a fundamental tree structure to model the interconnection. Rectilinear Steiner minimum tree (RSMT) aims to minimize the wirelength. BOI [5], BI1S [10], RV-based RST [20] and FLUTE [7] are near-optimal RSMT heuristics while GeoSteiner [24] is optimal with reasonable runtime for small trees. Obstacle avoiding RSMT (OA-RSMT e.g. [2,12,17,18]) is one extension to RSMT, which avoids the pre-designed IP blocks and macros. More recently, [25] and [13] propose over-the-block RSMT (OB-RSMT) to properly use the routing resources over the pre-designed blocks in order to achieve better performance.

RSMT and related extensions produce good results regarding wirelength minimization, but they are not timing-optimal in deep sub-micron high-speed ICs. To help meet timing on critical paths, timing-driven RST is needed to optimize pin-to-pin delays on those paths. Approaches, such as [9,14,23], focus on the minimum delay routing tree (MDRT) problem which minimizes a linear combination of delays at sinks. Other approaches(e.g. [3, 6, 15]) are able to optimize the required arrival time at the driver as a more practical target. Besides, timing optimization and obstacle avoidance are simultaneously considered in [19], etc. However, most of the abovementioned timing-driven approaches have the following three problems:

1) In order to build an RST optimizing required arrival time at the driver, it is necessary to know the criticality at all sinks. The first problem is that most previous works (such as [3,19,21]) use simple estimation on arrival time and criticality for each sink, which is not accurate enough. For example in [3], an optimally buffered 2-pin direct connection from root to one node is used to estimate the potential delay; similarly in [21], the require arrival time is calculated based on distance from root to merging point, neglecting the coupling from other part of the tree. Estimation cannot fully capture interconnection delay, including delay on wires and buffers, decoupling effect by buffers and load capacitance from un-buffered branch, which would result in a sub-optimal timing-driven RST. One the other hand, a buffered tree with topology close enough to the final constructed tree could provide criticality at all sinks accurately. We propose a pre-buffering approach in place of estimation so as to provide more accurate timing information. During pre-buffering, a timing-driven RST is iteratively built and buffered to offer criticality information for the next generation of timing-driven RST until the tree topology converges.

As is shown in Fig.1, if only estimation is used, it would conclude that sink E is critical, resulting in the topology in Fig.1(a). However, if we insert buffers on the topology in Fig.1(a) and re-calculate criticality, we will find that sink D is as critical as E. Based on that finding, the new topology would re-clusters D with E with a direct connection to root S. Upon this new topology, a new buffer insertion is applied

ISPD'14, March 30–April 2, 2014, Petaluma, CA, USA.

http://dx.doi.org/10.1145/2560519.2560533.

to re-calculate criticality at each sink. In this example, we find the set of critical sinks is not changed anymore and thus the topology converges to Fig.1(b) which has a better WNS since the slack on D is improved.

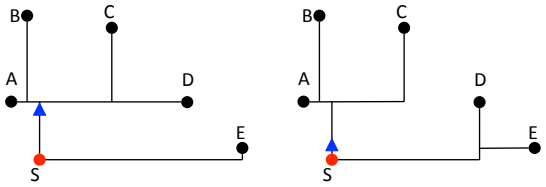

Figure 1: (a) estimates only sink E is critical. (b) groups sink E and D as critical cluster.

2) From [25] and [13], it has been demonstrated that over-the-block RSMT (OB-RSMT) outperforms OA-RSMT in terms of wirelength. Over-the-block routing resources should be used in timing-driven RST construction as well to replace obstacle-avoiding detours with shorter over-the-block connection. In the meantime, certain slew constraints have to be satisfied for over-the-block routing to ensure the solution will not fail buffering. Fig. 2 compares obstacle-avoiding tree construction with over-the-block algorithm, in which the latter shifts part of the inside tree outside and keeps the remaining inside the block. As is shown in Fig.2(b), the algorithm reduces two buffers, some detouring wirelength and delay of paths in the tree.

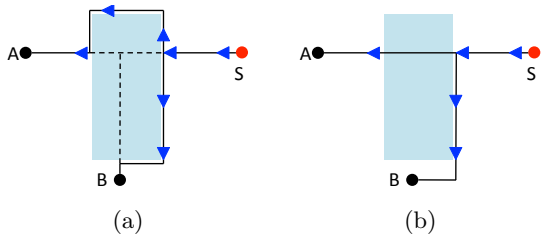

Figure 2: (a) is an OA-RSMT with root S and two sinks A, B. (b) uses part of the over-the-block routing resources.

3) Following topology generation and buffering, it has never been discovered or discussed that a buffer-location-based tuning can achieve considerable timing improvement without consuming additional buffering cost and noticeable wirelength. During the buffering, in order to obtain a legal buffering solution, some buffers are placed at positions without fully using up their power. The proposed post-buffering tuning algorithm could tune the locations of Steiner points based on the buffering information to further improve slack. In Fig. 3(a), we observe that buffer $b2$ is clamped under the Steiner point D to shield part of the downstream capacitance of D. We can change the position of the Steiner point (Fig. 3(b)) which makes the sequential buffers $b1$ and $b2$ parallel. The delay of the path from root S to A is notably reduced since the path becomes a decoupled direct connection and delay on buffer b_1 is taken away. However in a traditional flow, it is hard to accurately predict these better buffer locations via only topology generation and buffering.

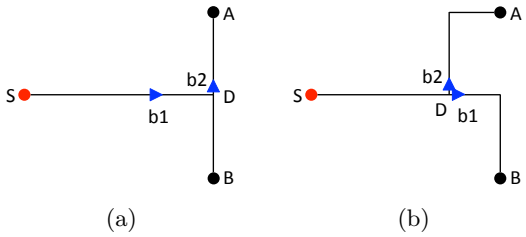

Figure 3: (a) is a buffered RST with root S and two sinks A, B. (b) exhibits the tuned topology and new buffering.

Our work makes the following major contributions:

1. We first propose a timing-driven, over-the-block RST construction algorithm which utilizes over-the-block routing tracks to reduce delay to critical sinks and shorten wirelength to non-critical sinks.
2. Our constructed RST satisfies the slew constraints everywhere with buffers placed at empty space.
3. During the tree construction, we use pre-buffering scheme to provide more accurate timing information, which helps explore better topologies for timing-driven RST.
4. We analyze the final buffered tree and relocate certain Steiner points to further improve the delay on paths to critical sinks.
5. We conduct our algorithm and observe significant improvements in WS, wirelength and buffering cost compared with existing works.

The rest of paper is organized as follow. We first introduce basic concepts and our problem formulation in Section 2. Our timing-driven, over the-block RST construction algorithm will be presented in Section 3, which includes three subsections. Section 3.1 discusses how to use pre-buffering to guide the tree construction. Section 3.2 discusses how to use over-the-block routing resources to reduce delays on critical paths without violating slew constraints. modify BOB-RSMT to ensure slew for over-the-block part. Section 3.3 introduces the post-buffering topology tuning algorithm which achieves considerable timing improvement without consuming noticeable wirelength and buffering cost. Experimental results will be shown in Section 4, followed by conclusions in Section 5.

2. NOTATIONS AND PROBLEM FORMULATION

In a two-dimensional routing region, we are given a net $N = \{s_0, s_1, s_2, \ldots, s_n\}$ with $n+1$ pins, where s_0 is the unique source and the rest are sinks. $L = \{b_1, b_2, \ldots, b_m\}$ is a set of non-overlapping rectilinear blocks in a two-dimensional space R. For $\forall\ s_i \in N$, s_i is not inside the two-dimensional space occupied by L. Any area with high-density logic cells not allowed for buffering is also taken as buffering blockage into L.

Our algorithm constructs a timing-driven buffered tree $T(V, E)$ to connect all the pins in N, where V is the set of nodes and E is the set of horizontal and vertical edges.

T might intersect with blocks in L, which confines a set of trees $S = \{T_1, T_2, \ldots, T_l\}$ inside blocks. We call trees in S *inside trees*. The outside-the-block part of T is defined as T_0. The buffered tree $T_b(V_b, E_b)$ is generated from T after we insert a set of nodes V' which corresponds to the buffers chosen from buffer library B, and $V_b = V \cup V'$.

The Steiner tree has a unique path $P(s_0, s_i)$ from s_0 to each sink s_i. The presence of buffers along the path could separate the path into *stages*, each of which consists of a driver, a set of driven nodes as well as edges connecting the driver and the driven nodes. The total delay on a path is the summation of the delay on each stage along that path, which can be computed in many ways. As in this discussion, we adopt the Elmore model for wires and a switch-level linear model for gates. The models we adopt are simple and informative enough to guide our approach, yet our formulation is by no means restricted to these models. The delay of each stage in the path is expressed as:

$$t(d(u), u) = \sum_{e=(i,j)\in p(d(u),u)} r_e l_e (0.5 c_e l_e + C_u(j)) + R_b C_d(d(u)) + D_b \quad (1)$$

Total delay of the path is the summation over all stages in the path:

$$d(s_0, s_i) = \sum_{u \in V' \cap p(s_0, s_i)} t(d(u), u) \quad (2)$$

The slack of sink s_i is defined as $slack(s_i) = RAT(s_i) - d(s_0, s_i)$. WS is defined as $WS(T) = min\{slack(s_i) | 1 \leq i \leq n\}$, and the worst negative slack is determined by $WNS(T) = min\{0, WS\}$. Notations amongst the formulation are as follows:

- l_e = length of edge e,
- r_e = unit length wire resistance on a chosen layer for edge e,
- c_e = unit length wire capacitance on a chosen layer for edge e,
- R_b = chosen buffer or source output resistance,
- C_b = chosen buffer or source input capacitance,
- D_b = internal buffer or source delay,
- $d(u)$ = the driver of node u,
- $t(u, v)$ = delay from node u to node v,
- $C_d(v)$ = total capacitance of the sub-tree rooted at node v down to the nearest downstream buffer or sinks, including the sink or buffer input capacitance,
- $C_u(v) = C_d(v)$ if v is not a buffer or source; C_b if v is a buffer or source node,

For slew calculation, we adopt the PERI model [16]:

$$S(v_j) = \sqrt{S(v_i)^2 + S_{step}(v_i, v_j)^2} \quad (3)$$

$S(v_j)$ is the slew at any node v_j, calculated as the root-mean square of the *step slew* from v_i to v_j and *output slew* at node v_i. The output slew at v_i is described by a 2-D lookup table of input slew and load capacitance. The experimental results in [16] show the error of PERI is within 1%, which is indistinguishable from what is obtained using SPICE simulation. For simplicity, we use Bakoglu's metric [4] for step slew calculation:

$$S_{step}(v_i, v_j) = \alpha * Elmore(v_i, v_j), \alpha = ln9 \quad (4)$$

The combination of Bakoglu's metric and the PERI model is shown to have error within 4% [16], which is, in general, accurate enough for RST construction purpose.

Our algorithm will construct a buffered RST T to connect all sinks and root while ensuring the slew rate on every point in the tree is within constraints. We use *slew mode* buffering as our buffering scheme as it is more predominantly used ([11, 22]) and saves buffering cost. The slew mode buffering satisfies the slew constraints on every point of the buffered tree with minimum buffering cost. Our buffered tree will have edges over the blocks but no buffers are allowed over the blocks. The object is to minimize the WNS of the tree with the lowest buffering cost.

3. TIMING-DRIVEN OVER-THE-BLOCK RST

Our approach constructs a timing-driven, over the-block RST with slew constraints. First, the approach uses coupled buffering and topology generation to provide AT and criticality at each sink. Then, a timing-driven RST is constructed based on pre-buffering. Second, the topologies of over-the-block trees are optimized to meet the slew constraints while maintaining the delay to critical sinks. Then, buffering is performed on the constructed tree structure. Finally, the constructed tree is tuned based on buffering information followed by buffering again. The overall algorithm of proposed approach is illustrated in Algorithm1.

Algorithm 1 *The overall algorithm*

Input: Set of pins N and blocks L
Output: Timing-driven over-the-block RST T
1: Construct timing-driven initial RST T with pre-buffering
2: Change the topology of T to meet the slew constraints
3: Perform buffering on T
4: Tune the topology of T based on buffering information
5: Perform buffering on T
6: **return** T

3.1 Initial Tree Generation with Pre-Buffering

Timing-driven RST requires the calculation of AT on each sink and might need RAT on internal nodes during the tree construction. Simple estimation of timing is inaccurate since there is no way to calculate the delay of un-constructed part of the tree or consider the final buffer distribution in the tree construction phase. Instead of using estimation, we apply pre-buffering to guide the tree construction.

Fig.4 depicts the proposed initial tree generation flow. We first generate a tree through any timing-driven RST algorithm. In this paper, we use state-of-the-art critical-trunk-based RST algorithm [19] to generate this initial tree (not considering blockages in this stage). Then pre-buffering part will buffer the RST and analyze timing. We save these topology and buffering if they are best-so-far. We calculate the real AT based on the buffered tree to substitute the pseudo time used in the tree topology generation algorithm as feedback information.

In the next iteration, all real critical sinks and critical trunks are re-determined because of the new timing information. In RST algorithm, we re-fix the critical trunks while

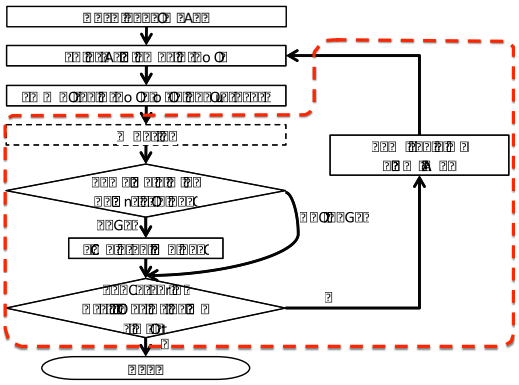

Figure 4: Flow of initial tree generation

the other two-pin nets are ripped up and re-routed by maze routing after the timing-driven critical trunk growth. Finally a post-process including rectilinearization and redirection is applied, which produces another RST. We will iterate

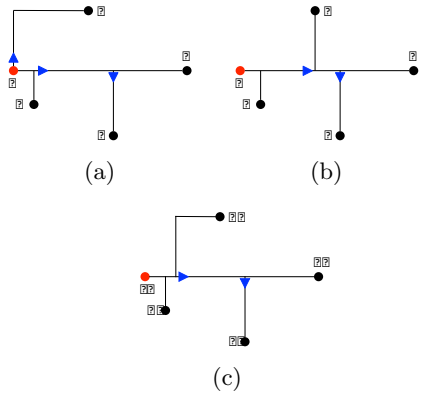

Figure 5: (a) is the initial critical trunk based tree with root S and sinks A,B,C,D. (b) reconstructs the tree according to the pre-buffering and timing information from (a). The tree topology converges in (c).

the whole procedure until the tree topology converges, or oscillates between several states, or the time limit is reached. Then we choose the best topology and WNS in our iterations as our initial tree. The new part of pre-buffering is indicated by dashed lines in Fig.4.

Example in Fig.5 shows that the topology and timing converge during the iterations. Initial structure in Fig.5(a) directly connects sink A to root as the RAT of A is small. In the next iteration, the topology generator decides to directly connect A to the trunk as in Fig.5(b), since according to Fig.5(a) the delay to A is small enough to meet the RAT, which in turn allows late branch. The late branch in Fig.5(b) leads to larger delay to sink A and eventually the topology converges to Fig.5(c) where the branch point of the path from root to A sits in the middle trunk leading to a star-like RSMT structure.

Table 1: 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 wirelength penalty of EP_1^t if EP_i^t moves to PPS_{ij}
C_i	estimated the timing criticality of EP_1^t

3.2 Buffering-Aware Over-the-Block Routing

We generate the initial tree without considering the blocks. The initial tree could cross over the blocks and break slew constraints even after buffer insertion. To prevent these violations, we change the topologies of over-the-block inside trees by approach similar to [25]. The objective in [25] is to minimize total wirelength only. Yet, in order to consider timing at the same time, we integrate criticality and slack into the objective function which minimize the wirelength of non-critical path as well as delay on critical path.

The initial tree confines a set of inside trees. For each inside tree, the ports, excluding the driver, on the boundaries of the block are called escaping points (EP). We use a mid-size hypothetical buffer at the driver and mid-size hypothetical buffers at each EP to determine if the tree has slew violation. Using mid-size hypothetical buffers instead of two extreme sizes will weaken the capability of utilizing more over-the-block routing resources, but the former turns out a more practical assumption and leads to less buffering cost as more solutions can propagate through this inside tree. If any inside tree violates the slew constraints, we apply three optimization primitives including parallel sliding, perpendicular sliding and EP merging [25] to fix the slew violations. Three optimization primitives are with different cost in our formulation since we consider timing as well.

For each inside tree t with slew violations, we first sort the illegal EPs per their slew violations. Next, in every iteration we choose the first illegal escaping point EP_1^t with the worst slew violation based on sorting. To improve slew for EP_1^t, each escaping point from $\{EP_1^t, EP_2^t, \ldots EP_{|EP^t|}^t\}$ may slide to a different position by taking a combination of primitives.

The combination of optimization primitives provides each escaping point a set of possible points. Each possible point in the set is a point on the boundary edge where escaping point may move to, which in turn improves the worst slew. Moving every escaping point to certain possible point guarantees $slew_1^t$ to meet slew requirement. In the extreme situation where maximum slew constraint is zero, EP_1^t can still become legal escaping point after we merge one escaping point to another until only the driver is left. 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} represent the step slew at EP_1^t and output slew reduction of the driver if EP_i moves to PP_{ij}. W_{ij} stands for the correspondingly estimated wirelength penalty. The possible point set associated 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.

In order to construct the inside tree under the slew constraint as well as meeting slack constraints, $\forall EP_i^t \in EP^t$ we need to decide which possible point to choose. The simultaneous point choice problem can be formulated in an optimization problem as follows (notation in Table 1):

$$\min. \sum_{i=1}^{|EP^t|} \sum_{j=1}^{|PPS_i^t|} X_{ij} W_{ij} (C_d(EP_i^t) C_i + \beta) \quad (5)$$

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

$$(S^t(D^t) + \sum_{i=1}^{|EP^t|} \sum_{j=1}^{|PPS_i^t|} X_{ij} B^t_{ij})^2 <= {slew^t_{spec}}^2 \quad (5a)$$

$$\sum_{j=1}^{|PPS_i^t|} X_{ij} = 1 \qquad \forall i \in \{1, 2, \ldots, |EP^t|\} \quad (5b)$$

The objective function (5) is to minimize the increase in delay on the critical paths and wirelength on non-critical paths. $W_{ij} C_d(EP_i^t)$ is the multiplication of resistance and total downstream capacitance, which estimates the amount of increase in delay for every sink downstream from EP_i^t. $C_i = \sum_{s_k} |slack(s_k)|$ is the weight for critical paths below EP_i^t, summing all absolute values of negative slacks of sinks downstream from EP_i^t. The weight β in the objective function selects solution with less estimated wirelength penalty on non-critical paths. The value of β is set remarkably smaller than $C_d(EP_i^t)C_i$ to avoid affecting critical paths. This objective function prefers less change on the critical paths while [25] can choose to increase the wire on critical path and exacerbate the WNS. Through the change of formulation, our new formulation considers the delay on critical paths and wirelength of non-critical paths.One example is that Fig.6(c) is preferred to Fig.6(b) because the former reserves the timing for critical sink by moving escaping point on non-critical path to satisfy slew constraints. Constraint (5a) restricts that the total slew reduction on EP_1^t has to be able to pull $slew_1^t$ down below requirement. Constraint (5b) is used to limit only one position chosen for each escaping point.

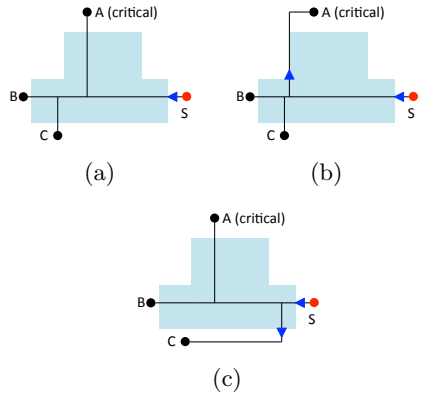

Figure 6: The root is S and three sinks are A, B, C. (a) is the initial timing-driven RST with slew violations. (b) fixes the slew violations with minimum wirelength penalty. (c) fixes the slew violations and considers the delay on critical path.

3.3 Timing-driven Buffer-location-based Tuning

We apply the slew mode buffering to the timing-driven, over-the-block RST, which satisfies slew constraints with minimum buffering cost. In the slew mode buffering, each buffer is desired to drive to its limit, implying that the worst slew rate among all receivers (buffers or sinks) should reach the slew limit. Similar to the concept of slack in timing calculation, we define *slew margin* which means the worst input slew rate among all receivers does not reach the slew limit. The existence of slew margin is because the driver or Steiner points in the tree topology may enforce the buffering solution to place one buffer to shield capacitance from one side.

3.3.1 Slew Margin

In a RST, a Steiner point is the joint point for at least two sub-branches to merge at. Before propagating buffer solutions through the Steiner point, each sub-branch will have an unbuffered segment connected to the Steiner point, such as OB, Ob_1 in Fig.7(a). These segments do not require buffers individually, but as a whole they may exceed the amount one large buffer can drive after propagating the Steiner point. The buffering tool has to place at least one buffer right below the Steiner point to shield one remaining segment to keep this solution legal. The buffering tool will place another buffer above the Steiner point to drive the unshielded parts along with the wire segment above the Steiner tree (this buffer can be saved if root is above the Steiner point with ability to drive). For instance, in fig.7(b), S is driver and O is a Steiner point. The segment OB is shielded by inserting a new buffer b_2. The shielding buffer b_2 will not drive to its limit as we already know that the length of driven segment is less than the optimal reach length. Therefore, the stage below b_2 ends up with slew margin. In Fig.7(a), the slew limit we adapt is 70ps, and the stage driven by b_2 exhibits slew margin with maximum slew 60ps at sink B. We also notice that the stage driven by driver S also has slew margin since the maximum slew is 65ps at the input of buffer b_1.

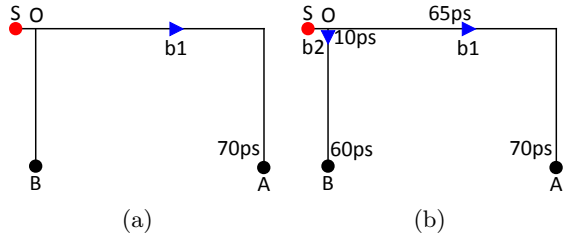

Figure 7: (a) bottom-up buffer solutions before merge at Steiner node O. (b) slew margin after propagation through Steiner node O

3.3.2 Buffer-location-based Tuning

Because the slew margin implies that the wire can be elongated to some extent without violating the slew constraints, the elongation of wires allows the change in topology without additional buffering cost. Per our approach, there exists a way of changing topology to improve timing on critical path by elongating the wire with slew margin. As the slew mar-

gin occurs below the Steiner point, we extracts the simplified pattern with one Steiner point and two buffers demonstrated in Fig.8(a). Buffer b_1 sits right below the Steiner point for shielding and buffer b_2 stays above the Steiner point as in Fig.8(a). We analyze this simplified pattern to generalize the way of changing topology used in our topology-tuning algorithm. We annotate the stage driven by b_1 as $stage_1$, that

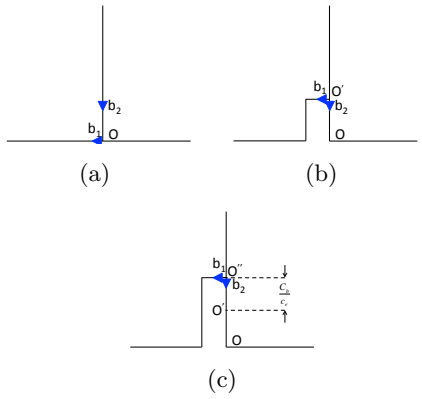

Figure 8: (a) depicts the pattern of slew margin. (b) shows buffer-location-based tuning if the input capacitance of buffers is negligible. (c) illustrates buffer-location-based tuning without neglecting the input capacitance of buffers.

driven by b_2 as $stage_2$ and that above b_2 as $stage_0$. Since $stage_1$ contains slew margin, we can calculate the elongation amount l to use up the slew margin. We denote the distance between b_2 and O as $l(b_2, O)$.

Observation I. If $l > l(b_2, O)$ and the input capacitance of buffers is negligible compared with wire capacitance, all slew constraints will be satisfied if we move the Steiner point to the location of b_2 and shift buffer b_1 up to the location right below the new location of the Steiner.

Fig.8(b) shows this buffer-location-based tuning. Under the assumption of negligible input capacitance of buffer, the load and slew of $stage_0$ are not changed. The slew of the $stage_1$ is still within constraints owing to $l > l(b_2, O)$.

Observation II. If $l > l(b_2, O) + C_b/c_e$ and the input capacitance of buffers is not neglected, we can keep all slew constraints satisfied by moving the Steiner point to C_b/c_e above b_2 and shifting buffer b_1 up to the location right below the new location of the Steiner node. (C_b is the input capacitance of buffer b_1 and c_e is the unit capacitance for the wire segment above b_2)

Fig.8(c) illustrates the topology and buffering after the relocation of Steiner point O to C_b/c_e above b_2 and buffer shifting. Because the wirelength above b_2 is curtailed by C_b/c_e, the downstream capacitance for $stage_0$ is reduced by $C_b/c_e * c_e$ accordingly. Buffer b_1 is attached to $stage_0$ during buffer-location-based tuning, including C_b into the downstream capacitance. Therefore the total downstream capacitance remains the same for $stage_0$. The amount of the downstream capacitance of $stage_2$ increases by $C_b/c_e * c_e$ as wire $O''O'$ is added below b_2. The input capacitance of b_1 is removed from $stage_2$ where the downstream capacitance is reduced by C_b. Hence the total downstream capacitance below b_2 stays the same. Under the assumption $l > l(b_2, O) + C_b/c_e$, the slew of $stage_1$ is still under slew constraints.

Algorithm 2 *Buffer-location-based Tuning*

```
Input: Buffered tree T
Output: Timing improved buffered tree T
Sort sinks in ascending order of slack
for each sink s_i with negative slack do
    node n = s_i
    while n! = s_0 do
        if n is Steiner point then
            if find buffer buffers b_1 right below n and b_2 above n
            then
                Calculate l based on slew margin
                if l > l(b_2, O) + C_b/c_e then
                    T_copy = T
                    Relocate n to C_b/c_e above b_2 and reconnect wires
                    Shift buffer b_1 up to right below n
                    if WNS(T) <= WNS(T_copy) then
                        T = T_copy
                    end if
                end if
            end if
        end if
        n = Parent(n)
    end while
end for
return T
```

3.3.3 Algorithms

Our proposed algorithm searches for the pattern which satisfies all the above assumptions. The algorithm scans the buffered topology in a bottom-up fashion. Once a pattern analyzed in Section3.3.2 is detected, we perform the above-mentioned buffer-location-based tuning. The search starts from the worst negative slack sink among the set of sorted negative slack sinks. We evaluate the newly generated topology and commit the potential improvements. The algorithm is described in Algorithm2.

4. EXPERIMENTAL RESULTS

We have implemented our algorithm in the C++ programming language. The experiments are conducted on an Intel Core 3.0GHz Linux machine with 32GB memory. We choose Gurobi Optimizer 5.10 as our solver for the integer linear programming.

RC01-RC12 are benchmarks in our experiments, same as those in [19]. We use two sizes of buffers in our experiment. The output resistances for two buffers are 450 ohms and 850 ohms, and the input capacitance are 3.8 fF and 1.9 fF respectively. Environment settings for wire and slew are calculated based on ITRS [1]. We use different resistance and capacitance for both horizontal and vertical layers. Each Steiner tree is placed on pre-selected layers. The slew constraint is set as 70 ps. Since the benchmarks do not comprise any timing information, to test the effectiveness of the slack optimization in our approach, we set RAT such that about 15% of the sinks are with negative slack in a buffered minimum spanning tree interconnection.

We will evaluate pre-buffering, over-the-block routing and post-buffering tuning individually. We use the algorithm in [19] as baseline for our comparison since as far as we know it possesses state-of-the-art performance driven RST construction with buffering while others (such as [25] and [13])

Table 2: Comparisons between TOB-RST-1, TOB-RST-2 and TOB-RST

Bench	Lin [19]			TOB-RST-1			TOB-RST-2			TOB-RST			
-marks	WNS (ps)	Buff	WL (um)	WNS (ps)	Buff	WL (um)	WNS (ps)	Buff	WL (um)	WNS (ps)	Buff	WL (um)	CPU (s)
RC1	-86	32	30220	-86	32	30220	-34	31	29370	-34	31	29370	0.52
RC2	-206	58	55700	-157	54	50880	0	52	48750	0	52	48750	0.89
RC3	-160	77	75730	-141	71	64270	-92	63	59530	-92	63	59530	0.82
RC4	-347	80	76340	0	84	79720	0	76	72920	0	76	72920	0.85
RC5	-305	95	92650	-177	102	97470	-108	96	96570	-108	96	96570	1.03
RC6	-722	134	130055	-722	134	130055	-521	123	118342	-423	123	119545	1.26
RC7	-605	179	185064	-574	174	182188	-249	162	178504	-162	162	179051	3.08
RC8	-418	189	185320	-220	191	190775	0	175	176920	0	175	176920	4.51
RC9	-787	182	177603	-517	186	180089	-126	168	162815	0	168	167240	7.70
RC10	-455	203	210040	-272	206	211910	-23	198	205650	0	198	209908	6.85
RC11	-1268	259	282338	-1142	265	287312	-1027	262	284077	-965	262	285290	11.41
RC12	-1221	885	1107538	-1008	912	1144662	-687	881	1101521	-245	881	1108324	27.38
Average	-548	1	1	-418	1.02	1.02	-239	0.96	0.97	-169	0.96	0.98	5.525

are not timing-driven RST. We notate the timing-driven OA-RST constructed with pre-buffering as TOB-RST-1, the timing-driven RST with both pre-buffering, over-the-block routing as TOB-RST-2, and the final tree with pre-buffering, over-the-block routing and post-buffering tuning as TOB-RST.

4.1 Effectiveness of Pre-Buffering

First, to solely evaluate pre-buffering, we compare the performance of TOB-RST-1 with that of OA-RSMT generated by [19] in Table 2. Columns 5, 6, 7 in the table list the WNS, buffering cost and total wirelength of TOB-RST-1, while columns 2 to 4 present those for [19]. Since the required time of each sink is different in our experiments, the wirelength in column 2 is different from that of SD-OARST in [19]. As we can see, WNS is improved for most test cases, and the average improvement is 130 ps, while the change of buffering and wirelength is within 2%. The similarity of wirelength (buffering cost) demonstrates that the different set of critical sinks selected by pre-buffering benefits the slack with little impact on wirelength (buffering cost). In the experiments, the topologies of most benchmarks converge while only the topology of RC4 oscillates between two states and the better one of the two states is returned. Also, all of the benchmarks converge or oscillate remarkably fast within four iterations at most.

4.2 Over-the-Block RST

To evaluate the effectiveness of over-the-block routing in TOB-RST-2, we compare TOB-RST-2 with TOB-RST-1. Columns 5 to 7 in Table 2 illustrate the WNS, buffering cost and total wirelength of TOB-RST-1 while the columns 8 to 10 are for TOB-RST-2. As shown in the table, over-the-block routing can improve WNS for all benchmarks. The average WNS improved from over-the-block routing is 179 ps with buffering cost and wirelength reduced by 6% and 5% respectively.

4.3 Post-buffering Topology Tuning

We compare TOB-RST with TOB-RST-2 to evaluate the effectiveness of post-buffering topology tuning. We only apply buffer-location-based tuning on critical paths with negative slack. Columns 11 to 13 in Table 2 present the WNS, buffering cost and total wirelength of TOB-RST. TOB-RST acquires about 70 ps improvements in WNS on average with less than 1% more wirelength. The buffering cost is the same since the post-buffering topology tuning does not consume buffering resources. We include total CPU runtime for TOB-RST algorithm in column 14 of Table 2, which contains total runtime of pre-buffering, over-the-block routing and post-buffering topology tuning. TOB-RST turns out to be fast since the maximum runtime is within one minute.

5. CONCLUSION

In this paper, we study a new class of RST problems, i.e., timing-driven over-the-block rectilinear Steiner minimum tree. We propose an effective and efficient algorithm which applies pre-buffering, over-the-block optimization and post-buffering tuning to optimize the slack on critical paths while saving wirelength on non-critical ones. Per our proposed approach, the generated topologies significantly improve WNS for all benchmarks along with 2% less wirelength and 4% less buffering cost than SD-OARST approach. Our proposed TOB-RST algorithm can be used in routing or post-routing stage to provide high-quality topologies to help close timing. This is the first work to solve timing-driven over-the-block RST problem crucial to high performance IC designs with multiple IP-blocks.

6. ACKNOWLEDGMENTS

This work is supported in part by Oracle. The authors would like to thank Dr. Salim Chowdhury and Dr. Akshay Sharma from Oracle for helpful discussions.

7. REFERENCES

[1] 2012 Overall Roadmap Technology Characteristics (ORTC) Tables. http://www.itrs.net/Links/2012ITRS/Home2012.htm.

[2] G. Ajwani, C. Chu, and W. Mak. FOARS: FLUTE Based Obstacle-Avoiding Rectilinear Steiner Tree Construction. In *Proc. ISPD*, pages 194–204, 2010.

[3] C. J. Alpert, M. Hrkic, J. Hu, A. B. Kahng, J. Lillis, B. Liu, S. T. Quay, S. S. Sapatnekar, A. J. Sullivan, and P. Villarrubia. Buffered Steiner Trees for Difficult Instances. In *Proc. ISPD*, pages 4–9, 2001.

[4] H. B. Bakoglu. Circuits, interconnections, and packaging for VLSI. Addison-Wesley, 1990.

[5] M. Borah, R. M. Owens, and M. J. Irwin. An edge-based heuristic for Steiner routing. *IEEE TCAD*, 13(12):1563–1568, 1994.

[6] Chung-Kuan Cheng, Ting-Ting Y. Lin, and Ching-Yen Ho. New performance driven routing techniques with explicit area/delay tradeoff and simultaneous wire sizing. In *Proc. DAC*, pages 395–400, 1996.

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

[8] J. Cong, L. He, K. Khoo, C. K., and D. Z. Pan. Interconnect Design for Deep Submicron ICs. In *Proc. ICCAD*, pages 478–485, 1997.

[9] J. Cong, K. Leung, and D. Zhou. Performance-Driven Interconnect Design Based on Distributed RC Delay Model. In *Proc. DAC*, pages 606–611, 1993.

[10] J. Griffith, G. Robins, J. S. Salowe, and T. Zhang. Closing the gap: Near-optimal Steiner trees in polynomial time. *IEEE TCAD*, 13(11):1351–1365, 1994.

[11] S. 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.

[12] T. Huang and E. F. Young. An Exact Algorithm for the construction of Rectilinear Steiner Minimum Trees among Complex Obstacles. In *Proc. DAC*, pages 164–169, 2011.

[13] T. Huang and E. F.Y. Young. Construction of rectilinear Steiner minimum trees with slew constraints over obstacles. In *Proc. ICCAD*, pages 144–151, 2012.

[14] k. D. Boese, A. B. Kahng, B. A. McCoy, and G. Robins. Rectilinear Steiner Trees with Minimum Elmore Delay. In *Proc. DAC*, pages 381–386, 1994.

[15] A. B. Kahng and B. Liu. Q-Tree: A New Iterative Improvement Approach for Buffered Interconnect Optimization. In *Proc. IEEE Annual Symp. on VLSI*, pages 183–188, 2003.

[16] C. V. Kashyap, C. J. Alpert, F. Liu, and A. Devgan. Closed Form Expressions for Extending Step Delay and Slew Metrics to Ramp Inputs. In *Proc. ISPD*, pages 24–31, 2003.

[17] L. Li, Z. Qian, and E. F. Young. Generation of Optimal Obstacle-avoiding Rectilinear Steiner Minimum Tree. In *Proc. ICCAD*, pages 21–25, 2009.

[18] L. Li and E. F. Young. Obstacle-avoiding Rectilinear Steiner Tree Construction. In *Proc. ICCAD*, pages 523–528, 2008.

[19] Y. Lin, S. Chang, and Y. Li. Critical-trunk-based obstacle-avoiding rectilinear Steiner tree routings and buffer insertion for delay and slack optimization. *IEEE TCAD*, 30(9):1335–1348, 2011.

[20] Ion I. Mandoiu, Vijay V. Vazirani, and Joseph L. Ganley. A new heuristic for rectilinear Steiner trees. *IEEE TCAD*, 19(10):1129–1139, 2000.

[21] T. Okamoto and J. Cong. Interconnect Layout Optimization by Simultaneous Steiner Tree Construction and Buffer Insertion. In *Proc. Asia and South Pacific Design Automation Conf.*, pages 44–49, 1996.

[22] P. J. Osler. placement driven synthesis case studies on two sets of two chips: hierarchical and flat. In *Proc. ISPD*, pages 190–197, 2004.

[23] M. Pan, C. Chu, and P. Patra. A Novel Performance-Driven Topology Design Algorithm. In *Proc. Asia and South Pacific Design Automation Conf.*, pages 244–249, 2007.

[24] D. M. Warme, P. Winter, and M. Zachariasen. Exact algorithms for plane steiner tree problems: a computational study, 2000.

[25] Y. Zhang, A. Chakraborty, S. Chowdhury, and D. Z. Pan. Reclaiming Over-the-IP-Block Routing Resources With Buffering-Aware Rectilinear Steiner Minimum Tree Construction. In *Proc. ICCAD*, pages 137–143, 2012.