A Place and Route Aware Buffered Steiner Tree Construction*

C. N. Sze, Jiang Hu and Charles J. Alpert†
Dept. of Electrical Engineering, Texas A&M University, College Station, TX 77843, USA; {cnsze,jianghu}@ee.tamu.edu
† IBM Austin Research Lab, Austin, TX 78758, USA; alpert@us.ibm.com

Abstract—In order to achieve timing closure on increasingly com-
plex IC designs, buffer insertion needs to be performed on thousands
of nets within an integrated physical synthesis system. In most of
previous works, buffers may be inserted at any open space. Even
when there may appear to be space for buffers in the alleys between
large blocks, these regions are often densely packed or may be useful
later to fix critical paths. In addition, a buffer solution may inadvert-
ently force wires to go through routing congested regions. Therefore,
within physical synthesis, a buffer insertion scheme needs to be aware
of both placement congestion and routing congestion of the existing
layout and so it has to be able to decide when to insert buffers in dense
regions to achieve critical performance improvement and when to uti-
lize the sparser regions of the chip. With the proposed Steiner tree
adjustment technique, this work aims at finding congestion-aware
buffered Steiner trees. Our tree adjustment technique takes a Steiner
tree as input, modifies the tree and simultaneously handles the objec-
tives of timing, placement and routing congestion. To our knowledge,
this is the first study which simultaneously considers these three ob-
jectives for the buffered Steiner tree problem. Experimental results
confirm the effectiveness of our algorithm while it achieves up to 20×
speed-up when comparing with the state-of-the-art algorithm [5].

I. INTRODUCTION

It has been widely recognized that interconnect becomes a
dominating factor for modern VLSI circuit designs. A key
technology to improve interconnect performance is buffer in-
sertion. A recent study by Intel [17] speculates that for 35nm
technology, 70% of the cells on a chip will be buffers.

Early works on buffer insertion are mostly focused on im-
proving interconnect timing performance. The most influen-
tial pioneer work is van Ginneken’s dynamic programming
algorithm [19] that achieves polynomial time optimal solution on
a given Steiner tree under Elmore delay model [8]. In [14],
Lillis et al. extended van Ginneken’s algorithm by using a
buffer library with inverting and non-inverting buffers, while
also considering power consumptions.

The major weakness of the van Ginneken approach is that
it requires a fixed Steiner tree topology which makes the fi-
nal buffer solution quality dependent on the input Steiner tree.
Even though it is optimal for a given topology, the van Gin-
neken algorithm will yield poor solutions when it is fed with
a poor topology. To overcome this problem, several works have
proposed to simultaneously construct a Steiner tree while per-
forming buffer insertion [15, 10, 9]. Although the simulta-
neous algorithms generally yield high quality solution, their
time complexities are very high. A different approach to solve
the weakness of van Ginneken’s algorithm is proposed by
Alpert et al. [4]. They construct a “buffer-aware” Steiner tree,
called C-Tree for van Ginneken’s algorithm. Despite being
a two-stage sequential method, it yields solutions comparable
in quality to simultaneous methods, while consuming signifi-
cantly less CPU time.

Recent trends towards hierarchical (or semi-hierarchical)
chip design and system-on-chip design force certain regions
of a chip to be occupied by large building blocks or IP
cores so that buffer insertion is not permitted. These con-
straints on buffer locations can severely hamper solution qual-
ity, and these effects have to be considered in the buffered
path [12, 13, 20] class of algorithms. Though optimal, they
are only applicable to two pin nets. Works that handle re-
strictions on buffer locations while performing simultaneous

Steiner tree construction and buffer insertion are proposed in
[7, 18], which can provide high quality solutions though the
runtimes are too exorbitant to be used in a physical synthesis
system. In [2], a Steiner tree is rerouted to avoid buffer block-
ages before conducting buffer insertion. This sequential ap-
proach is fast, but sometimes unnecessary wiring detours may
result in poor solutions. An adaptive tree adjustment technique
is proposed in [11] to obtain good solution results efficiently.
The huge number of nets that require buffering means that
resources have to be allocated intelligently. For example,
large blocks closely placed together create narrow alleys that
are magnets for buffers since they are the only locations that
buffers can be inserted for those routes that cross over these
blocks, but competition for resources for these routes is very
fierce. Inserting buffers for less critical nets can eliminate
space that is needed for more critical nets which require gate
sizing or other logic transforms. Further, though no blockages
lie in the alleys, these region could already be packed with
logic and feasible space may not exist for the buffers. If the
buffer insertion algorithm cannot recognize this scenario, after
a buffer is inserted into this congested space, placement legal-
ization may shift it out too far from its original location due
to cell overlap. Hence, whenever possible one should avoid
denser regions unless it is absolutely critical. For example,
Figure 1(a) shows a multi-pin net which is routed through a
dense region or “hot spot”, and (b) shows that the Steiner point
is moved outside of the dense region in order to obtain an im-
proved buffer insertion result.

Similarly, a buffer solution may inadvertently force wires to
go through routing congested regions such as in Figure 2(a).
Such solution causes wire detours in routing stage. Gener-
ally speaking, an L-shaped wire has flexibilities on avoiding
congestions without increasing wirelength. These flexibilities
may be ripped off when inserted buffers make the L-shaped
wire into a set of straight connections. If we let the buffering
algorithm be aware of the routing congestions, these flexibili-
ties can be kept for congestion avoidance in later routing stage.
To the best of our knowledge, the only published work about
placement congestion aware buffer insertion is the porosity
aware buffered Steiner tree problem addressed in [5]. This
work integrates the length-based buffer insertion [3] with
a plate-based tree adjustment to obtain a Steiner tree at regions
with greater porosity. However, the routing congestion is not
considered and the runtime overhead is too large.

This work adopts the sequential method in constructing a
Steiner tree and the tree is then fed into a van Ginneken style
buffer insertion algorithm. However, before buffer insertion, a
timing-driven plate-based tree adjustment algorithm is ap-
is applied so that both the placement and routing congestion are considered. Buffered paths between nodes are found through utilizing the analytical form buffered path solution and a congestion cost driven maze routing. In [5], each solution during path search is characterized by its cost and downstream capacitance, thus, the solution set is a two-dimensional array while at the same time, the tree adjustment is not driven by timing optimization. In the maze routing of this work, only the congestion cost is considered for each candidate solution. Therefore, the candidate solution set is a one-dimensional array and this smaller-sized solution set enables a faster computational speed. In fact, our experiment shows that, when comparing with [5], our algorithm achieves 7% better timing, 9% lower total placement and routing congestion but runs up to 20 times faster.

II. PROBLEM FORMULATION

In this paper, we use a tile graph to capture the placement and routing congestion information and at the same time reduce the complexity of our problem. A tile graph is represented as $G = (V_G, E_G)$ such that $V_G = \{v_1, v_2, \ldots, v_n\}$ is a set of tile and $E_G$ is a set of boundaries each $(g_i, g_j)$ of which is between two adjacent tile $g_i$ and $g_j$.

If a tile $g_i \in V_G$ has an area of $A(g_i)$ and its area occupied by placed cells is $a(g_i)$, the placement density is defined as the area usage density $d(g_i) = \frac{a(g_i)}{A(g_i)}$. Let $W(g_i, g_j)$ be the maximum number of wires that can be routed across the tile boundary $(g_i, g_j)$ and $w(g_i, g_j)$ be the number of wires crossing $(g_i, g_j)$. Similarly, the boundary density is $d(g_i, g_j) = \frac{w(g_i, g_j)}{W(g_i, g_j)}$.

A net is represented as a set of sinks $V_{sink} = \{v_1, v_2, \ldots, v_n\}$ and a source node $v_0$. Each sink $v_i \in V_{sink}$ at location $(x_i, y_i)$ is an area and is associated with a load capacitance $C_{l}$ and a required arrival time $q(v_i)$. The source node is at $(x_0, y_0)$ and is associated with a driver with driver resistance $R_d$. For simplification, a buffer type with input capacitance $C_b$, intrinsic delay $t_b$ and output resistance $R_b$ is used. The unit wire resistance is $r$ and the unit wire capacitance is $c$. We use Elmore model [8] for interconnect delay and RC switch model for driver and buffer delay.

Problem Definition: (Buffered Steiner Tree for Placement and Routing Congestion Mitigation) Given a net $N = \{v_0, v_1, \ldots, v_n\}$ with source $v_0$ and sinks $\{v_1, \ldots, v_n\}$, load capacitance $C_{l}$, and required arrival time $q(v_i)$ for $1 \leq i \leq n$, tile graph $G(V_G, E_G)$, and a buffer type $b$, construct a Steiner tree $T(V, E)$, in which $V = N \cup V_{Sink}$ and edges in $E$ span every node in $V$, such that a buffer insertion solution that satisfies $q(v_i)$ is obtained with a minimum congestion cost $S$.

The congestion cost can be formulated based on the application but the algorithm should have the flexibility to take any kind of congestion cost formulation. In this paper, we adopt the following definition. The placement cost $p(g_i)$ of placing a buffer in a tile $g_i$ is the square of the density $d(g_i)$; while the routing cost $p(g_i, g_j)$ crossing a tile boundary $(g_i, g_j)$ is the square of the boundary density $d(g_i, g_j)$. With this cost definition, we do not use an infinite cost for overflow. In reality, if the placement or routing on a dense region really helps improving slack or other design objectives, moving the previously placed and routed elements in the next design cycle would be more beneficial.

III. THE ALGORITHM

A. Methodology Overview

Since simultaneous Steiner tree construction and buffer insertion is computationally expensive for practical circuit designs, we propose to solve the congestion aware buffered Steiner tree problem through the following three stages: 1) Initial timing-driven Steiner tree construction; 2) Tree adjustment for congestion improvement; 3) Van Ginneken style buffer insertion.

Stage 1 can be accomplished by any heuristics while we apply “buffer aware” C-Tree algorithm[4]. Stage 2 contains the key ideas behind the algorithm. The tree adjustment phase modifies the existing timing-driven Steiner tree in an effort to reduce congestion cost while maintaining the tree’s high performance. It allows Steiner points to migrate out of higher-congested tiles into lower-congestion tiles while maintaining (if not improving) performance. Finally, in Stage 3 the routing tree topology is fixed for van Ginneken style buffer insertion. Since we use known algorithms for Stages 1 and 3, the rest of the discussion focuses on stage 2, which is the main contribution of this work.

B. Algorithm Motivation

The basic idea for the tree adjustment is to perform a simplified simultaneous buffer insertion and local tree topology modification so that the Steiner nodes and wiring paths can be moved to less congested regions without significant disturbance on the timing performance obtained in Stage 1. Also for the sake of simplification, we assume a single “typical” buffer type, the Elmore delay model for interconnect and a switch level RC gate delay model for this tree adjustment. The tree adjustment traverses the given Steiner topology in a bottom-up fashion similar to van Ginneken’s algorithm. During this process, candidate buffering and routing solutions are propagated from leaf nodes towards the source. At a Steiner node, candidate solutions from its two child branches are merged. Therefore, we can consider the propagation and merging process separately.

The buffered Steiner tree problem is inherently very difficult to solve and including congestions into account makes the complexity even more formidable. In this problem, three major factors (1) timing (2) load capacitance and (3) congestion cost have to be considered simultaneously on a two-dimensional Manhattan plane. Note that load capacitance needs to be evaluated and maintained for delay calculation even though it is not a part of objectives. In order to make the computation time practical, solution quality has to be sacrificed to a certain degree. Of course, the sacrifice on solution quality need to be as small as possible while the computation cost reduction has to be substantial. In [5], a length-based buffer insertion [3] scheme is employed to reduce complexity. Instead of maintaining all the timing and load capacitance information, only the maximum driving load for each buffer/driver is enforced as a rule of thumb. Therefore, the number of factors is reduced from three to two and the computation speed is acceptable. However, the experimental results in [5] show that runtime is almost doubled just because of considering congestions. The runtime bottleneck is due to the fact that we choose a timing-driven Steiner tree algorithm here since it is fast and easy to implement with our proposed tree adjustment technique. However, any Steiner tree (e.g., congestion-aware Steiner tree) can also be fed into stage 2 of our overall algorithm.

Note that the “buffer insertion” in stage adjustment is for timing estimation. Actual buffer insertion is performed in stage 3 of our algorithm.
that buffering solution has to be searched along with node-to-node paths in a two-dimensional plane since low congestion paths have to be found at where the buffers are needed.

If we can predict where buffers are needed in advance, then we can merely focus on searching low congestion paths and the number of factors to be considered can be further reduced to one. If we diagnose the mechanism on how buffer insertion improves interconnect timing performance, it can be broken down into two parts: (1) regenerating signal level to increase driving capability for long wires and (2) shielding capacitive load at non-critical branches from the timing critical path. In a Steiner tree, buffers that play the first role are along a node-to-node path while buffers for the second purpose are normally close to a branching Steiner node. The majority of buffer insertion algorithms such as van Ginneken’s method are dynamic programming based and have been proved to be very effective for both purposes. However, optimal buffer solutions along a node-to-node path can be found analytically if the driving resistance for this path is known [1, 6]. This fact suggests that we may have a hybrid approach in which buffers along paths are placed according to the closed form solutions while the buffers at branching nodes are still solved by dynamic programming, i.e., analytical buffered path solutions replace both the wire segmenting [1] and candidate solution generations at segmenting points in the bottom-up dynamic programming framework.

The only problem of this approach is that the driving resistance of a path to be processed is not known in this bottom-up procedure. This can be solved by sampling a set of anticipated upstream resistance values and generating candidate buffering solutions for each anticipated value. Different sampling rate may result in different solution quality and runtime trade-offs. Computing candidate buffered paths analytically is faster than dynamic programming, because the complexity of a dynamic programming approach has a quadratic dependence on the segmenting size while analytical approach has only linear dependence on the upstream resistance sampling size.

C. Steiner Node Adjustment

For a Steiner node, we find a few nearby tiles with the least congestion. In each of these tiles, we consider an alternative Steiner node there. Therefore, the candidate solutions from child branches are propagated to not only the original Steiner node but also these alternative Steiner nodes. For a node \( v \), we define expanded node set as its alternative nodes as well as the node itself and denote this set as \( P(v) \). The selection of alternative nodes in \( P(v) \) can be controlled by the placement cost and wiring cost of the tiles\(^4\) or for different objectives of steiner node adjustment, other selection schemes may be applied. After candidate solutions from child branches are merged at the original Steiner node and each of the alternative

\(^3\)See Section IV for the \( P(v) \) selection details adopted in our experiment.

Steiner nodes, the merged solutions are propagated further towards the source. This process is illustrated in Figure 3 where the tiles for the expanded node set are shaded. The alternative Steiner nodes enable alternative tree topologies and only the topology that is part of the best solution at the root will be finally selected. Therefore, this tree adjustment is a dynamic selection.

Usually, we restrain the alternative Steiner nodes to be close to the original Steiner node so that the perturbation to the original timing driven Steiner tree is limited. We define the adjustment flexibility which represent what the maximum distance of the alternative Steiner nodes can be away from the original Steiner node. For example, in Figure 3, all alternative nodes are enclosed in 3 x 3 tiles and so we refer it as the 3 x 3 tiles adjustment flexibility. By this definition, we can have a larger flexibility for the selection of expanded node set if, for instance, we adopt the 5 x 5 tiles adjustment flexibility in our program. A larger adjustment flexibility would lead to a higher possibility for the original Steiner node to migrate outside of the congested area.

The main difference between our proposed technique and the work of [5] is that a regular array of tiles are considered for alternative Steiner node in [5] while our selection on alternative Steiner nodes is according to the congestion for nearby tiles. This is based on our observation that only the nearby tiles with relatively low buffer placement or routing congestion cost worth considering to be the alternative Steiner node. Moreover, due to the irregularity and flexibility of expanded node set in our algorithm, we have more choices and can pick alternative Steiner nodes other than the immediate neighbor nodes for better congestion reduction and higher efficiency. On the contrary, since [5] features a regular array of tiles, a larger flexibility for alternative Steiner nodes is desired, the runtime would become unbearable. This can be shown by our experiments in Section IV.

D. Minimum Cost Buffered Path

Our work distinguishes from [5] significantly on the candidate solution propagation between two nodes. In [5], a length based buffer insertion is integrated with the minimum congestion cost path search, so the process is not timing-driven. However, our algorithm has the required arrival time information in an intermediate solution during the bottom-up propagation and pruning process, and hence our work is capable of handling not only the congestion reduction but also the timing optimization. In order to achieve these two objectives, we separate the buffer insertion for timing from the minimum congestion cost path search. For a path of length \( l \) with driver resistance \( R_d \)}
at one end and a load capacitance \( C_L \) on the other end, the number of buffers \( k \) that minimizes the path delay is obtained in [1] as:

\[
k = \left\lfloor \frac{1}{2} + \frac{1}{2} \sqrt{1 + \left(\frac{2rcL - r(C_k - C_L) - c(R_h - R_d)}{rc(R_h - R_d)}\right)^2} \right\rfloor
\]

The \( k \) buffers separate the path into \( k + 1 \) segments of length \( l_0, l_1, ..., l_k \) as illustrated in Figure 4. According to [1], the length of each segment can be obtained through:

\[
l_0 = \frac{1}{k+1} \left( l + \frac{k(R_h - R_d)}{r} + \frac{C_L - C_k}{c} \right)
\]

\[
l_j = \frac{1}{k+1} \left( l - \frac{R_h - R_d}{r} + \frac{k(C_L - C_k)}{c} \right)
\]

Then, we explain our buffered path routing technique by an example. For the thickened path in Figure 5(a), if we know the driving resistance at \( v_1 \) and load capacitance at \( v_4 \), we may obtain the optimal buffer positions at \( v_2 \) and \( v_3 \). However, if we connect \( v_1 \) and \( v_4 \) in a two-dimensional plane, there are many alternative paths between them and the optimal buffer locations form rows along diagonal directions. The tiles for the optimal buffer locations are shaded in Figure 5(a). Therefore, if we connect \( v_1 \) and \( v_4 \) with any monotone path and insert a buffer whenever this path passes through a shaded tile, the resulting buffered path should have the same minimum delay. The thin solid curve in Figure 5(a) is an example of an alternative minimum delay buffered path. Certainly, different buffer paths may have different congestion cost. Then the minimum congestion cost buffered path can be found by running the Dijkstra’s algorithm on the tile graph which is demonstrated in Figure 5(b). In Figure 5(b), each solid edge corresponds to a tile boundary and its edge cost is the corresponding wiring congestion cost (square of its boundary density). There are two types of nodes, the empty circle nodes that have zero cost and filled circle nodes that have cost equal to the placement congestion cost in corresponding tile. In conclusion, the shortest path obtained in this way produces a buffered path with both good timing and low congestion cost.

An issue need to be handled by this approach is that the upstream resistance \( R_u \) is unknown in the bottom-up solution propagation process. However, we are aware that the lower bound on the upstream resistance is \( R_u = \min(R_a, R_b) \) and the upper bound \( \overline{R_u} \) is \( \max(R_a, R_b) \) plus the upstream wire resistance\(^5\). Then, we can sample a few values between \( \overline{R_u} \) and \( \overline{R_u} \), and find the minimum cost buffered path for each value. Since the timing result is not sensitive to the upstream resistance, normally the sampling size is very limited.

### E. Overall Algorithm

In our algorithm, each intermediate buffer solution is characterized by a 4-tuple \( s(v, c, q, w) \) in which \( v \) is the root of

\(^5\)The maximum upstream wire resistance can be derived from the length of maximum buffer-to-buffer interval. This is also mentioned in [3].

---

**Procedure: FindCandidate \( s(v) \)**

**Input:** Current node \( v \) to be processed  
**Output:** Candidate solution set \( S(P[v]) \)  
**Global:** Steiner tree \( T[V, E] \)  
**Tile graph** \( G(V_G, E_G) \)

1. If \( v \) is a sink  
   \[ S(v) = \{ (v, c, q, w), 0 \} \]
   \[ S(P[v]) = \{ S(v) \} \]
   Return \( S(P[v]) \)
2. \( v \leftarrow \) left child of \( v \)
   \[ S(P[v_l]) \leftarrow \text{FindCandidate}(s(v_l)) \]
3. \( S(P[v]) \leftarrow \text{Propagate}(S(P[v_l]), P(v)) \)
4. If \( v \) has only one child
   Return \( S(P[v]) \)
5. \( v_r \leftarrow \) right child of \( v \)
   \[ S(P[v_r]) \leftarrow \text{FindCandidate}(s(v_r)) \]
6. \( S(P[v]) \leftarrow \text{Propagate}(S(P[v_l]), P(v)) \)
7. \( S(P[v]) \leftarrow \text{Merge}(S(P[v_l]), S_r(P(v)) \]
8. \( S(P[v]) \leftarrow \text{Prune}(S(P[v])) \)
9. Return \( S(P[v]) \)

---

**Procedure: Propagate \( S(P[v_l]), P(v) \)**

**Input:** Candidate solutions at \( P(v_l) \)  
**Expanded node set** \( P(v) \)  
**Output:** Candidate solution set \( S(P[v]) \)

1. \( S(P[v_l]) \leftarrow \emptyset \)
2. For each node \( v_l \in P(v_l) \)
   \[ s(v_l) \leftarrow \emptyset \]
3. For each node \( v_k \in P(v_l) \)
4. For each anticipated upstream resistance \( R_u \) at \( v_l \)
5. For each node \( v_k \in P(v_l) \)
6. For each anticipated upstream resistance \( R_u \) at \( v_l \)
7. \( S(P[v]) \leftarrow S(v_l) \cup \{ s(v_l, c, q, w) \} \)
8. \( S(P[v]) \leftarrow \text{Prune}(S(P[v])) \)
9. Return \( S(P[v]) \)

---

**Procedure: FindBufferPositions \( (R_u, v_n, v_d) \):** apply equations of (1) to find the set of buffer positions \( \mathcal{L} \) for the minimum delay from node \( v_n \) to \( v_d \) assuming driving resistance \( R_u \) at \( v_n \). The required arrival time \( q \) and downstream capacitance \( c \) at \( v_n \) are also returned.

**FindMinCostPath \( (v_n, v_d, G, \mathcal{L}) \):** apply Dijkstra’s algorithm to find the minimum cost path connecting \( v_n \) and \( v_d \) on tile graph \( G(V_G, E_G) \). On tile graph \( G \), the cost of each node \( q \in V_G \) is \( \ell(q) \) if \( q \in \mathcal{L} \); otherwise zero. The cost of an edge in \( G \) corresponding to boundary between tiles \( q_i \) and \( q_j \) is \( \ell(q_i, q_j) \). Return the path cost finally. (An example is shown in Section III.D.)
FindCandidate(s) (Figure 6) is a recursive procedure which is similar to van Ginneken’s algorithm such that at each node, we propagate the solutions from its children, merge and prune the solutions. However, the procedure Propagate(S(P(v_i)), P(v_j)) (Figure 7) for solution propagation adopts our analytical equations and shortest path algorithm described in Section III.D, which accelerates our algorithm when compared with the van Ginneken’s dynamic programming approaches. When our algorithm terminates, we obtain a set of solutions with different timing and congestion cost tradeoffs at the root.

IV. EXPERIMENTAL RESULTS

All experiments are performed on a Sun Ultra Sparc 450 machine running in 400 MHz. Our experiments adopt the following parameters: \( r = 0.184 \Omega /\mu m \), \( c = 0.124 fF/\mu m \), \( R_b = 246.3 \Omega \), \( C_b = 7.2 fF \), and \( t_s = 27.46 ps \). The sampling size of upstream resistance is set to 3.

We implement our algorithm “Place and Route Aware Buffered Steiner Tree Construction” (PRAB) and the algorithm “Porosity Aware Buffered Steiner Tree Construction” in [5] (namely POROSITY in this paper) with C++ and compare them based on a set of industrial nets which is also used in [4] and [11]. In the benchmark nets, the driver resistance ranges from 271.8Ω to 2557.5Ω while the loading capacitance ranges from 2.9 fF to 55.6 fF.

For each net, after we construct a tile graph, we randomly produce a set of buffer blockages and then calculate the placement density for each tile which range from 0 to 1. The boundary density for each tile boundary is also generated randomly ranging from 0 to 0.25 so that the importance of buffer placement cost and wiring congestion cost balance each other in the buffered Steiner trees.

In order to ensure a fair comparison, the buffered Steiner tree generation for both implementations follows the three-stage scheme mentioned in Section III-A and therefore we can focus on the comparison between the congestion-aware tree adjustment step for both algorithms. After the tree adjustment in stage 2, our PRAB algorithm generates a set of solutions with different timing and congestion cost tradeoff but POROSITY produces a single solution which only considers the minimization of estimated congestion cost. Therefore, before running the van Ginneken style buffer insertion, we only pick the one with least estimated cost in our solution set for the comparison with POROSITY.

A. Results for Real Multi-Sink Nets

In the first experiment, we pick the expanded node set from the 3 × 3 neighbor tiles of each Steiner node – 3 × 3 tiles adjustment flexibility. Each expanded node set consists of three tiles: the tile \( g_1 \) containing the original Steiner node, the tile \( g_2 \) with the smallest buffer placement cost \( p(g_2) \), and the tile \( g_3 \) with the smallest wiring cost, which is defined as \( \sum_{e \in \text{E}(g)} W(g,e) \) where \( N(g) = \{g'|g \neq g'|E(g') \} \). To achieve the same Steiner node adjustment flexibility, we implement POROSITY with the “plate size” to be 3 × 3. Table I shows the comparison between the POROSITY algorithm and our PRAB algorithm. For each net, the second column shows the number of sinks. “graph size” displays the number of columns and rows in the tile graph which encloses the routing region of the net. “Stage 1 req” represents the required arrival time at source node which is propagated from all sinks along the initial timing driven Steiner tree without buffer insertion (just after Stage 1). Columns 4-9 represents the results of POROSITY [5] while columns 10-15 shows our results. “req” represents the required arrival time at source node for the buffered Steiner tree generated by all three stages for both algorithms. Then, “imp” means the timing improvement by each three-stage implementation when comparing to the first-stage Steiner tree. (Note that both algorithms generate a negative improvement for the net mcus05s since in the process of congestion mitigation, the total wire-length increases and in turn decreases the required arrival time at source node.) “cost” shows the total congestion cost that is a sum of wiring cost “w.cost” and buffer placement cost “b.cost”, which are induced by the final buffered Steiner tree on the tile graph. Although we perform all three stages for both implementations, only the CPU time for second stage is shown for proper comparisons.

From Table I, we observe the following:

- Our algorithm for tree adjustment outperforms POROSITY [5] in the sense of both timing improvement (7%) and congestion cost evaluation(7%). Particularly, we are better than POROSITY by 15% in buffer placement cost.
- PRAB runs with about 15 times speed-up. The main reason for the efficiency is that we selectively pick the expanded node set according to the congestion nearby the node containing the original Steiner node; And, buffer location is determined by an analytical formula so that node-to-node routing becomes very fast.

B. Results when More Choices for Expanded Node Set

The second experiment is intended to show that our algorithm is capable in handling the situation when a greater flexibility is needed for the Steiner node adjustment. As stated in Section III-C, the set of alternative Steiner nodes is defined by the expanded node set. In Table II, we have a 5 × 5 tiles adjustment flexibility while keeping the size of expanded node set to be 3, which is the same as the first experiment. Similarly, POROSITY is implemented with each plate consisting of 5 × 5 tiles.

Table II consolidates our claims in the last experiment in the way that:

- PRAB generates better timing and congestion cost results than POROSITY.
- PRAB runs 16 times faster.

Table II demonstrates that our algorithm achieves a much faster Steiner tree adjustment with better solution quality and it also reveals the fact that for exploring a larger flexibility in Steiner node adjustment, our scheme of picking 3 tiles (irregularity) to be the expanded node set not only makes our algorithm very efficient but also accomplishes promising Steiner tree adjustment with congestion awareness.

C. Results for different adjustment flexibilities

We summarize Table I and II in Table III. It also includes the results for the 7 × 7 tiles adjustment flexibility setting with the size of expanded node set to be 3. The columns in Table III shows the corresponding total value of all 10 nets in previous experiments and the ratio of our PRAB algorithm compared to POROSITY. Table III shows that for larger adjustment flexibility, our PRAB algorithm together with the expanded node set selection scheme provides a larger speed-up in run-time while its solution quality is still better than [5] and the results agree with our claim that selecting a small number of tiles for the expanded node set according to congestion helps improving efficiency but it does not hurt the solution quality.

D. Summary

Experimental results confirm the effectiveness of our algorithm in producing solutions with better timing and less placement and routing congestion cost when comparing to the POROSITY algorithm. In our experiment, we use the Steiner
Null