Porosity Aware Buffered Steiner Tree Construction

Charles J. Alpert[1], Gopal Gandham[2], Milos Hrkic[3], Jiang Hu[4] and Stephen T. Quay[1]
1. IBM Corporation, Austin, TX 78758, {calpert,quayst}@us.ibm.com
2. IBM Corporation, Hopewell Junction, NY 12533, gopalg@us.ibm.com
3. University of Illinois at Chicago, CS Department, Chicago, IL 60607, mhrkic@cs.uic.edu
4. Texas A&M University, EE Department, College Station, TX 77843, jianghu@ee.tamu.edu

ABSTRACT

In order to achieve timing closure on increasingly complex IC designs, buffer insertion needs to be performed on thousands of nets within an integrated physical synthesis system. Modern designs may contain large blocks which severely constrain the buffer locations. Even when there may appear to be space for buffers in the alleys between large blocks, these regions are often densely packed or may needed later to fix critical paths. Therefore, within physical synthesis, a buffer insertion scheme needs to be aware of the *porosity* of the existing layout to be able to decide when to insert buffers in dense regions to achieve critical performance improvement and when to utilize the sparser regions of the chip.

This work addresses the problem of finding porosity-aware buffering solutions by constructing a "smart Steiner tree" to pass to van Ginneken's topology based algorithm. This flow allows one to fully integrate the algorithm into a physical synthesis system without paying an exorbitant runtime penalty. We show that significant improvements on timing closure are obtained when this approach is integrated into a physical synthesis system.

Categories and Subject Descriptors

B.7.2 [**Hardware**]: Integrated Circuits—*Design Aids*

General Terms

Algorithms, Performance

Keywords

VLSI, interconnect, physical design, buffer insertion

1. 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 insertion. Cong [4] speculates that close to 800,000 buffers will be required for 50 nanometer technologies.

ISPD'03, April 6-9, 2003, Monterey, California, USA.

1.1 Previous Work

Early works on buffer insertion are mostly focused on improving interconnect timing performance. The most influential pioneer work is van Ginneken's dynamic programming algorithm [16] that achieves polynomial time optimal solution on a given Steiner tree under Elmore delay model[7]. In [13], 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 has to be provided in advance which makes the final buffer solution quality dependent on the input Steiner tree. Even though it is optimal for a given topology, the van Ginneken algorithm will yield poor solutions when fed a poor topology. To overcome this problem, several works have proposed simultaneously constructing a Steiner tree while performing buffer insertion. Lillis *et al.* proposed the buffered P-Tree algorithm which integrates buffer insertion into the P-Tree Steiner tree algorithm[14]. Buffered P-Tree generally yields high quality solution, but its time complexity is also high because candidate solutions are explored on almost every node in the Hanan grid. In recently reported S-Tree[9] algorithm, alternative abstract topologies for a given Steiner tree are explored to promote solutions that are better at dealing with sink criticalities. This technique is integrated with P-Tree as SP-Tree algorithm in [8]. A different approach to solve the weakness of van Ginneken's algorithm is proposed by Alpert et al [3]. 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 significantly less CPU time.

Recent trends toward 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 constraints on buffer locations can severely hamper solution quality, and these effects need be considered. Thus buffer blockages are considered in the buffered path[17, 11, 12] class of algorithms. Though optimal, they are only applicable to two pin nets. Works that handle restrictions on buffer locations while performing simultaneous Steiner tree construction and buffer insertion are proposed in [5, 15]. Like buffered P-Tree, these approaches can provide high quality solutions though at runtimes too exorbitant to be used in a physical synthesis system. In [1], a Steiner tree is rerouted to avoid buffer blockages before conducting buffer insertion. This sequential approach is fast, but sometimes unnecessary wiring detours may result in poor solutions. An adaptive tree adjustment technique is proposed in [10] to obtain good solution results efficiently.

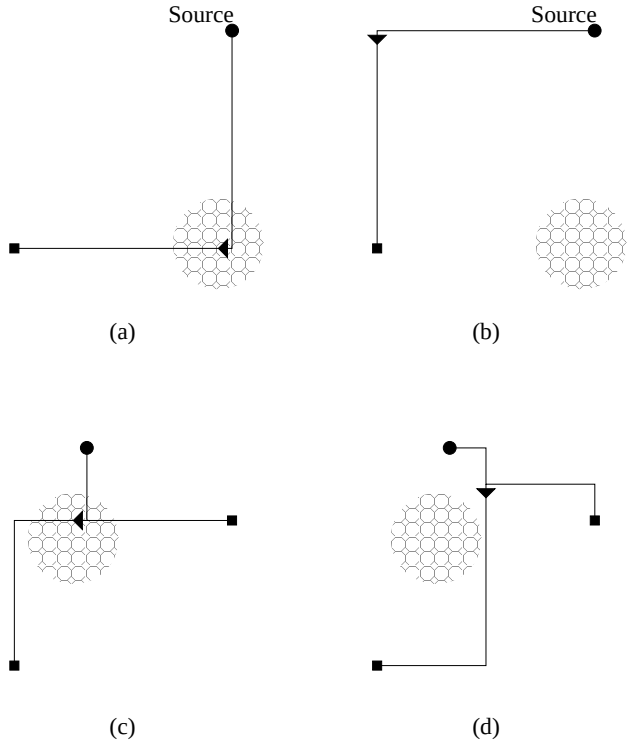


Figure 1: An arbitrary Steiner tree as shown in (a) and (c) may force buffers being inserted at dense regions which is indicated in blobs. A porosity aware buffered Steiner tree, which are shown in (b) and (d), will enable buffers solution at sparse regions.

1.2 The Importance of Porosity

In tradition design flows, buffer insertion is applied after placement to optimize interconnect timing. As interconnect effects become increasingly severe, buffer insertion needs to be pushed up earlier in the design flow to help logic optimization and placement algorithms on interconnect estimation and to reserve buffering resources. The sheer number of nets that require buffering means that resources need to be allocated intelligently. For example, large blocks placed close together create narrow alleys that are magnets for buffers since they are the only locations that buffers can be inserted for 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 that may require gate sizing or other logic transforms. Further, though no blockages lie in the alleys, these region could already be packed with logic and no feasible space for the buffer might exist. If the buffer insertion algorithm cannot recognize this scenario then inserting a buffer into this space may cause placement legalization to spiral it out to a region too far from its original location. Hence, whenever possible one should avoid denser regions unless absolutely critical.

For example, Figure 1(a) shows a 2-pin net which is routed through a dense region (denoted by circles) or "hot spot", and Figure 1(b) shows the net routed differently to avoid this congested region. Both solutions have the same wirelength and timing characteristics. Figure 1 (c) and (d) show another example for a multi-pin net where the Steiner point needs to be moved outside of the dense region to yield an improved buffer insertion result.

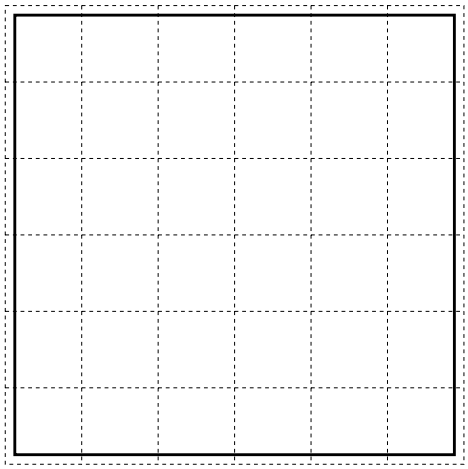


Figure 2: An example of tile graph.

The literature contains no buffering approach which addresses these issues. Note that several algorithms (such as S-Tree and SP-Tree) could be used to address porosity constraints since they can be run on arbitrary grid graphs. However, the extensions are hardly straightforward. One has to carefully consider the weighting scheme for edges in the grid graph, how to sparsify the graph to give reasonable runtimes, the appropriate cost functions, etc. Further, any simultaneous approach is likely to require too much runtime to be practically used within physical synthesis.

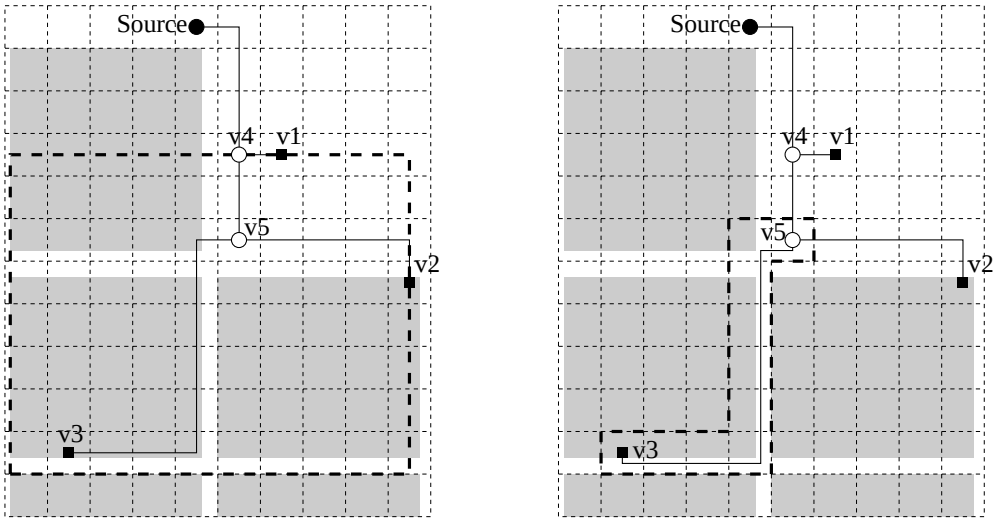
This work proposes a porosity aware buffered Steiner tree construction and is the first buffer insertion work to address porosity in the layout directly (as opposed to simply specifying a handful of allowable buffer insertion locations). The approach begins with a timing-driven but porosity-ignorant Steiner tree, then applies a plate-based adjustment guided by length based buffer insertion. After performing localized blockage avoidance, the resulting tree is then passed to van Ginneken's algorithm to obtain a porosity aware buffered Steiner tree. Physical synthesis experiments on real industrial circuits confirms the effectiveness of this framework.

2. PRELIMINARIES

The porosity is represented through a tile graph $G(V_G, E_G)$ where a node $g \in V_G$ corresponds to a tile and an edge $e_{uv} \in E_G$ represents a boundary between two neighboring tiles u and $v \in V_G$. An example of tile graph is provided in Figure 2. If the a tile $g \in V_G$ has an area of $A(g)$ and its area occupied by placed cells are $a(g)$, the placement density is defined as the area usage density $d(g) = \frac{a(g)}{A(g)}$ of tile g. We address the following problem:

Porosity-aware Buffered Steiner Tree Problem: *Given a net $N = \{v_0, v_1, \ldots, v_n\}$ with source v_0 and sinks $\{v_1, \ldots, v_n\}$, load capacitance $c(v_i)$ 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 $V = N \cup V_{Steiner}$ and edges in E span every node in V, such that a buffer insertion solution that satisfies $q(v_i)$ may be obtained with a minimal porosity cost.*

For porosity cost, one can sum the costs of the tiles that the routes cross. We use the following function, though certainly others can be used as well, depending on the application. For each buffer inserted in a tile g, the cost is the square of the density $d(g)$. Thus, the more buffers inserted, the higher the cost, though buffers are close to free in sparser regions.



(a) (b)

Figure 3: An example of local blockage avoidance. Shaded rectangles represent buffer blockages. The Steiner tree before blockage avoidance (a) and after blockage avoidance (b).

Note, that we do not recommend using an infinite penalty if inserting a buffer exceeds the density target. It may be better to insert a buffer into an over dense region and move other cells out of the region than to accept a significantly inferior delay result. Ultimately the right trade-off between porosity cost and timing depends on the criticality of the net and the design characteristics.

3. ALGORITHM DESCRIPTION

3.1 Overview

Since simultaneous Steiner tree construction and buffer insertion is computationally expensive for practical circuit designs, we prefer to first construct a Steiner tree followed by a van Ginneken style buffer insertion algorithm that employs higher order delay models, handles slew and load capacitance constraints, manages a library of inverting and non-inverting buffers and trades off buffer resource utilization with solution quality. Integrating a Steiner tree construction into this algorithm while maintaining all its features would be prohibitive. However, just constructing a Steiner tree without performing buffer insertions cannot possibly yield the best topology. Therefore, we propose to solve the porosity aware buffered Steiner tree problem through the following four stages:

1. Initial porosity ignorant timing-driven Steiner tree construction
2. Plate-based adjustment for porosity improvement
3. Local blockage avoidance
4. Van Ginneken style buffer insertion

For stage one, one can construct a timing-driven Steiner tree through any heuristics. We choose to apply the "buffer aware" C-Tree algorithm[3].

Stage 2 contains the key ideas behind the algorithm. The plate-based adjustment phase modifies the existing timing-driven Steiner in an effort to reduce congestion cost while maintaining the tree's high performance. One of the key elements is that it allows Steiner points to migrate outside of high-porosity tiles into lower-porosity tiles to reduce congestion while maintaining (if not improving) performance.

After Stage 2, the Steiner tree is correct in that it goes through the tiles that optimize performance while minimizing porosity cost, though the routes may overlap blockages. Performing local blockage avoidance in Stage 3, manipulates the routes within each tile to avoid blockages, thereby allowing more buffer insertion candidates. However, this stage does not disturb the tree topology uncovered from Stage 2. An example of the local blockage avoidance is illustrated in Figure 3.

Finally, in Stage 4 the resulting tree topology is fixed for van Ginneken style buffer insertion.

Since we use known algorithms for Stages 1 and 4, the rest of the discussion focuses on the other two stages.

3.2 Stage 2: Plate-based Adjustments

The basic idea for the plate-based adjustment is to perform a simplified simultaneous buffer insertion and local tree adjustment so that the Steiner nodes and wiring paths can be moved to regions with greater porosity without significant disturbance on the timing performance obtained in Stage 1.

The buffer insertion scheme employed here is similar to the length based buffer insertion in [2]. However, we allow the Steiner points and wiring paths to be adjusted. This approach is also distinctive from a simultaneous buffer insertion and Steiner tree construction approach[14] in which the Steiner tree is built from scratch. The plate-based adjustment traverses the given Steiner topology in a bottom-up fashion similar to van Ginneken's algorithm. During this process, Steiner nodes and wiring paths may adjusted together with buffer insertion to generate multiple candidate solutions.

The range of the moves are restrained by the *plates* defined as follows. For a node $v_i \in T(V, E)$ which is located in a tile g_k, a *plate* $P(v_i)$ for v_i is a set of tiles in the neighborhood of g_k including g_k itself. During the plate-based adjustment, we confine

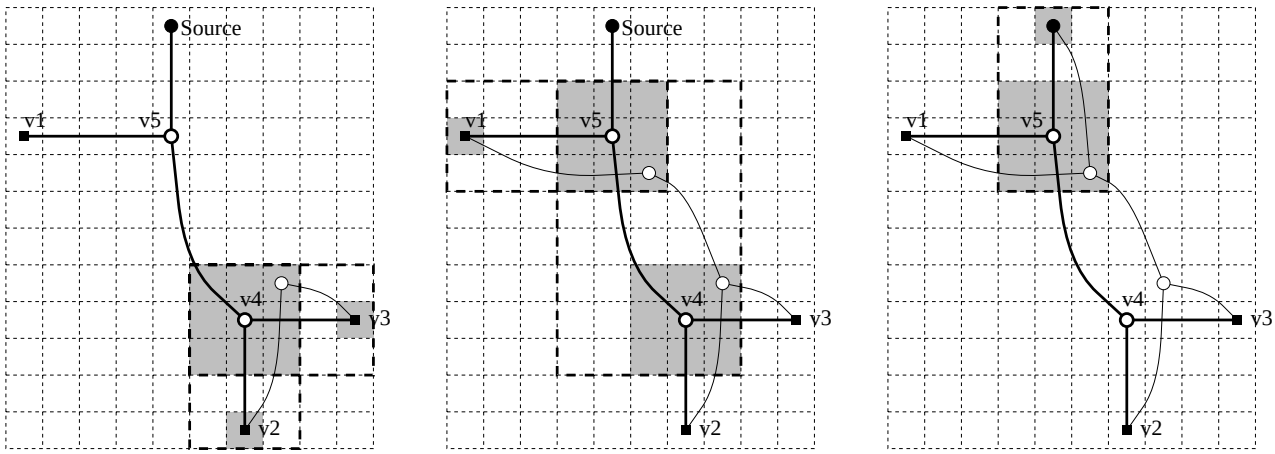


Figure 4: (a) Candidate solutions are generated from v_2 and v_3 and propagated to every tile, which is shaded, in the plate for v_4. Solution search is limited to the bounding boxes indicated by the thickened dashed lines. (b) Solutions from v_1 and every tile in the plate for v_4 are propagated to the plate for v_5. (c) Solutions from plate of v_5 are propagated to the source and the thin solid lines indicate an alternative tree that may result from this process.

the location change for each Steiner node within its corresponding plate. If v_i is a sink or the source node we let $P(v_i) = \{g_k\}$. For example, when v_i is a branch node, we set $P(v_i)$ to be the 3×3 array of tiles centered at g_k.[1] Of course, if g_k lies on the border of the entire tile graph, $P(v_i)$ may include fewer tiles. Figure 4(a) gives an example of the plate corresponding to Steiner node v_4. The plate indicates any of the possible location which the Steiner node may be moved to. This example shows the Steiner node being moved up one tile and right one tile.

The search for alternate wiring paths is limited to the minimum bounding box covering the plates of two end nodes. In Figure 4, such bounding boxes are indicated by the thickened dashed lines. Therefore, the size of plates define the search range for both Steiner nodes and wiring paths. Certainly, a plate may be defined to include more tiles or to even be a irregular shape. For example, if the plate size is 1×1, our algorithm will basically reduce to a length based/van Ginneken style buffer insertion algorithm along a fixed topology. However, one could also choose the plate to be the entire tile graph – this yields an algorithm similar to the S-Tree embedding [9]. By choosing a plate of size larger than one but smaller than the entire grid graph, we still obtain the ability to modify the topology to move critical Steiner points into low-porosity regions while also capping the runtime penalty. Hence, our experiments use a 3×3 tile size to capture this tradeoff. An example of how a new Steiner topology might be constructed from an existing topology is demonstrated in Figures 4(a)-(c).

Adopting a length-based buffer insertion scheme [2] makes the adjustment simpler and thereby faster. Since this stage only produces a Steiner tree before buffering, the purpose of including buffering at this stage is for estimation purposes, hence a simplified scheme should be adequate.

Alpert et al. [2] suggest that buffer insertion may be performed in a simple way by just following a rule of thumb: the maximal interval between two neighboring buffers is no greater than certain upper bound. The interval between neighboring buffers is specified in term of number of tiles. In this work, this constraint is modified to be the maximum load capacitance U a buffer/driver may drive, so that sink/buffer capacitance can be incorporated. To keep the succinctness of the tile-based interval metric in [2], we discretize the load capacitance in units equivalent to the capacitance of wire with average tile size. If the average tile boundary length is λ and the wire capacitance per unit length is $\hat{c}$, then a load capacitance is expressed in the number of $\beta = \lambda\hat{c}$. This modification allows us to use non-uniform sized tiles, even though we still assume such non-uniformity is very limited.

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 plate-based adjustment.

Each intermediate buffer solution is characterized by a 3-tuple $s(v, c, w)$ in which v is the root of the subtree, c is the discretized load capacitance seen from v, and w is the accumulated porosity cost. A solution $s_i(v, c_i, w_i)$ is said to be dominated by another solution $s_j(v, c_j, w_j)$, if $c_i \geq c_j$ and $w_i \geq w_j$. A set of buffer solutions $S(v)$ at node v is a non-dominating set when there is no solution in $S(v)$ dominated by another solution in $S(v)$. Usually $S(v)$ is arranged as an array $\{s_1(v, c_1, w_1), s_2(v, c_2, w_2)...\}$ in the ascending order of load capacitance. The basic operations of the length-based buffer insertion are:

- $AddWire(s_i(v, c_i, w_i), u)$: grow solution s_i at v to node u by adding a shortest distance wire between them. Node u is either within the same tile as v or in the neighboring tile of v. If the wire has capacitance C, we can get $c_j(u) = c_i(v) + \frac{C}{\beta}$ and $w_j(u) = w_i(v) + w(v, u)$ where $w(v, u)$ is the rectilinear distance between v and u times the placement density.

- $AddBuffer(s_i(v, c_i, w_i))$: insert buffer at v. If buffer b has an input capacitance c_b, output resistance r_b and intrinsic delay t_b, then the buffered solution $s_j(v, c_j, w_j)$ is character-

[1] Note that the size of the plate depends on the quality of solution/runtime tradeoff desired by the user. Using a finer tile graph will require a larger plate to push Steiner nodes outside of high density regions. One could alternatively use say a 5×5 array or even an alternate shape such as a diamond of tiles all within distance 2 of the original tile.

ized by $c_j(v) = c_b/\beta$ and $w_j(v) = w_i(v) + p(g)$ in which $p(g) = d^2(g)$ is the porosity cost for tile g where node v locates.

- $Prune(S(v))$: remove any solution $s_i \in S(v)$ that is dominated by another solution $s_j \in S(v)$.
- $Expand(s_i(v, c_i, w_i), u)$: grow $s_i(v)$ to node u by $AddWire$ to get solution $s_j(u, c_j, w_j)$, insert buffer for $s_j(u)$ to obtain $s_k(u, c_k, w_k)$. Add the unbuffered solution s_j and buffered solution s_k into solution set $S(u)$ and prune the solutions in $S(v)$.
- $Merge(S_l(v), S_r(v))$: merge solution set from left child of v to the solution set from the right child of v to obtain a merged solution set $S(v)$. For a solution $s_{i,l}(v, c_{i,l}, w_{i,l})$ from the left child and a solution $s_{j,r}(v, c_{j,r}, w_{j,r})$, they are merged to $s_k(v, c_k = c_{i,l} + c_{j,r}, w_k = w_{i,l} + w_{j,r})$ only when $c_k \leq U$.

Procedure: $FindCandidates(v)$
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
   $S(v) \leftarrow \{(v, 0, 0)\}$
   $S(P(v)) \leftarrow \{S(v)\}$
   Return $S(P(v))$
2. $v_l \leftarrow$ left child of v
   $S(P(v_l)) \leftarrow FindCandidates(v_l)$
3. $S_l(P(v)) \leftarrow Propagate(S(P(v_l)), P(v))$
4. If v has only one child
   Return $S_l(P(v))$
5. $v_r \leftarrow$ right child of v
   $S(P(v_r)) \leftarrow FindCandidates(v_r)$
6. $S_r(P(v)) \leftarrow Propagate(S(P(v_r)), P(v))$
7. $S(P(v)) \leftarrow Merge(S_l(P(v)), S_r(P(v))$
8. Return $S(P(v))$

Figure 5: Core algorithm.

Similar to the van Ginneken's algorithm, starting from the leaf nodes, candidate solutions are generated and propagated toward the source in a bottom-up manner. Before we propagate candidate solutions from node v_i to its parent node v_j, we first find both plate $P(v_i)$ and plate $P(v_j)$ and define a bounding box which is the minimum sized array of tiles covering both $P(v_i)$ and $P(v_j)$. Then we propagate all the candidate solutions from each tile of $P(v_i)$ to each tile of $P(v_j)$ within this bounding box. Note that branch nodes are allowed to be moved in a neighborhood defined by the plate. Since the branch nodes are more likely to be buffer sites due to the demand on decoupling non-critical branch load from the critical path, allowing branch nodes to be moved to less congested area is especially important. Moreover, such move is a part of a candidate solution, thus such move will be committed only when its corresponding candidate solution is finally selected at the driver. Therefore, such move is dynamically generated and selected according to the request of the final minimal porosity cost solution. The complete description on the core algorithm is given in Figure 5 and the subroutine of solution propagation is in Figure 6.

Procedure: $Propagate(S(P(v_i)), P(v_j))$
Input: Candidate solutions at $P(v_i))$
Expansion region $P(v_j)$
Output: Candidate solution set $S(P(v_j))$

0. $S(P(v_j)) \leftarrow \emptyset$
   $B \leftarrow$ bounding box of $P(v_i)$ and $P(v_j)$
   $Q \leftarrow \emptyset$
1. For each tile $g_k \in P(v_i)$
   $Q \leftarrow Q \cup S(g_k)$
2. While $Q \neq \emptyset$
3. $s \leftarrow$ min cost solution in Q
   $g_k \leftarrow$ tile where s locates
4. For each tile g_l adjacent to g_k
5. If $g_l \in B$
   $Expand(s(g_k), g_l)$
6. Return $S(P(v_j))$

Figure 6: Subroutine of propagating candidate solutions from one plate to another plate.

3.3 Stage 3: Local Blockage Avoidance

After the tree has been modify to route through the tiles with high porosity, the tree still may overlap blockages. This is especially true if one uses a sparse grid graph to begin with. This occurs because the plate-based adjustment in Stage 2 is performed with respect to a given tile graph, but large blockages themselves are managed as part of a tile. For example, if a blockage occupies half the area of a given tile, but the remaining half of the tile is empty, the tile's density is 50%. Of course, a net that traverses through this tile should be routed in the space not occupied by the blockage wherever possible. Thus, this stage performs local cleanup within tiles to achieve blockage avoidance without changing tile assignment.

The local blockage avoidance is similar to the algorithm proposed in [1]. The main idea is to divide each Steiner tree into a set of non-overlapping *2-paths* and then perform local blockage avoidance on each 2-path. A 2-path is a path of nodes for a given Steiner tree such that both endpoints are either the source, a sink, or a Steiner node. Thus, every internal node has degree two.

Each 2-path of a given Steiner tree is rerouted over an extended Hanan grid generated by drawing horizontal and vertical lines through each pin and extending the borders of each rectangle shaped buffer blockage. The extended Hanan grids are shown by the dashed lines in Figure 7. In this grid, if an edge does not overlap with any blockage, its cost is its geometric length. Otherwise, edge cost will be geometric length timed by a penalty coefficient greater than one. A minimal cost path is searched over this grid to replace original 2-path to achieve blockage avoidance. The value of the penalty coefficient controls the tradeoff between blockage avoidance and total wirelength. A greater value of the penalty coefficient implies a stronger desire to avoid the blockage and a greater wiring detour is allowed. A small penalty coefficient will restrict the total wirelength with a reduced blockage avoidance ability. Such tradeoff is illustrated in Figure 7.

During seeking the minimal cost path in [1], the entire region covering the two endpoints are expanded properly and are searched. For the example in Figure 3(a), when the 2-path between v_3 and v_5 is rerouted, the search region is indicated by the thickened dashed lines in Figure 3(a). In our case, since the Steiner tree after stage 2 is already in high porosity regions, we may refine the search region

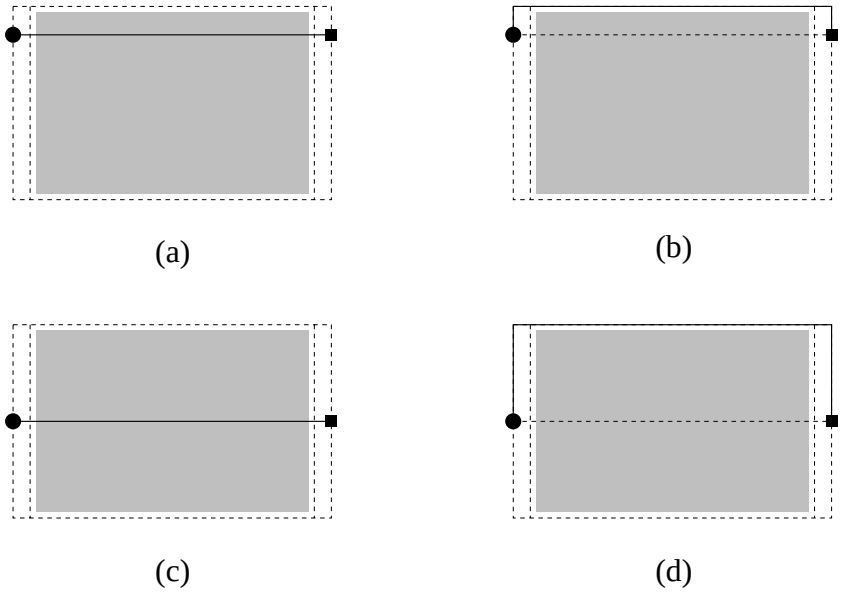


Figure 7: A path overlaps with a buffer blockage as in (a) can be moved out of the blockage with a small detour as in (b) with a small penalty coefficient on cost. Only a greater penalty coefficient can allow a large detour like from (c) to (d).

to be the tiles the 2-path passing through as shown by the thickened dashed lines in Figure 3(b). In this work, we define the penalty coefficient to be 1.01 so that the wire detour from this rerouting will be very small (no more than 1% wirelength increase) and the disturbance to the timing performance of the tree is very limited. In Figure 3, the rerouted path after blockage avoidance is shown in (b).

4. EXPERIMENTAL RESULTS

For our experiments, we implement our porosity-aware buffered Steiner tree algorithm into an industrial physical synthesis system called PDS [6]. The system begins with a placed netlist and performs several operations on critical paths to reduce timing closure, such as gate sizing, pin swapping, logic transforms, and of course buffer insertion. At the end of physical synthesis, PDS reports the following statistics measuring the success of a run:

1. Slack: the minimum slack among all the timing paths,
2. # Neg: the number of timing paths with negative slack,
3. FOM(Figure of Merit): a measure of the cumulative slack of all negative slack cells in the design and a greater value of FOM is desired,
4. TWL: total wirelength,
5. CPU: total CPU time in seconds on an RS6000 595 machine with 4 Gb or RAM.

For each of three industrial cases, we ran physical synthesis two different ways:

1. **Baseline**: just Stages 1 and 4 in Section 3 are run.
2. **Porosity**: the porosity-based modifications of Stage 2 and 3 are included on top of the baseline.

We make the following observations. For each case, using the porosity algorithm reduced the number of nets with negative slack and also the FOM. For two of the three test cases, some difference in the worst slack in the design was observed, though not enough to be significant. The total penalty for wirelength is less than half a percent. Finally, the CPU degradation was roughly a factor of 1.7 to 2.5.

To truly measure how effective the porosity algorithm is, we should hopefully see improvements in routing results as well, as the placement of buffers should cause the design to be more spread out. We were unable to obtain this data in time for the submission deadline.

In some sense, the ability to measure the effectiveness of the algorithm is only as good as the design data. For truly chunky semi-hierarchical designs, this technology should have a much bigger impact than for totally flat designs, since it is the alley space problems that can significantly impact the ability to close timing.

Table 2 isolates the runtimes of just the buffer insertion parts of the physical synthesis run. First, observe just how efficient the baseline buffer insertion algorithm can be. It can perform buffer insertion on roughly 5 to 30 nets per second. Using porosity aware buffer insertion certainly is less efficient, though it is still efficient compared to simultaneous approaches. For example, for test1 it can process about seven nets per second, while test2 is about one net per second and test3 is two nets per second.

Table 2: Detailed runtime analysis of just buffer insertion in PDS.

test	baseline		porosity				
case	#nets	CPU	#nets	CPU	Stage 1,4	Stage 2	Stage 3
test1	31736	1081	47119	6989	1810	3794	1385
test2	37093	1630	37514	31101	2747	23601	4753
test3	12851	3247	12623	25543	3122	22190	231

When one breaks down the total runtime of the porosity algo-

Table 1: Physical synthesis on industrial designs.

testcase	#cells	#blockages	grid size	algorithm	slack(ns)	#Neg	FOM	TWL	CPU
test1	155K	209	24×24	baseline	-1.60	36827	-10995	122.5	9803
				porosity	-1.50	29112	-8224	122.7	17300
test2	334K	848	32×32	baseline	-0.96	14885	-5209	205.4	19816
				porosity	-0.98	14115	-4896	206.2	49162
test3	293K	18	32×32	baseline	-1.32	59525	-25870	111.4	16155
				porosity	-1.29	51743	-22213	111.7	38714

rithm into its various stages, we observe that the runtime is dominated by Stage 2. We believe that there are significant speedups that can be integrated into this stage to achieve runtimes comparable to that of Stages 1, 3, and 4. In particular, the wavefront expansion is currently stored as a linked list; using a priority queue should result in significant savings. Stage 3 can also be sped up by performing pre-processing on the tile graph and blockage map so that the grid graph does not have to be constructed from scratch during the re-routing of each 2-path. Ideally the runtime for porosity aware buffer insertion should be about a factor of two worse than the baseline.

Finally, Figure 8 and 9 illustrate the kind of impact the algorithm can have. Figure 8 shows the baseline for an example 17-pin net for test2, and 9 shows the net layout for the porosity-aware algorithm. The circles along the topology indicate possible buffer insertion points. Observe that the layouts are similar, but the porosity-aware layout is able to better avoid blockages while maintaining the same integrity of the timing-driven Steiner tree. After inserting buffers, the worst case slack on the tree in Figure 9 is $150ps$ better than that of the tree in Figure 8.

5. CONCLUSION

This paper is the first to address the problem that not just blockages, but the porosity of the entire layout affecting the quality of buffer insertion algorithms, especially within a physical synthesis environment. We presented a four stage algorithm to address buffered Steiner tree design. The key idea in Stage 2 was to use a plate to limit the solution space exploration for Steiner nodes during tree adjustment, thereby permitting a fairly efficient algorithm. We demonstrated the effectiveness of this technique in a physical synthesis system. In future work, we seek to improve this approach by tweaking the cost function to improve performance and to achieve additional speedups via alternative data structures and pruning schemes.

6. REFERENCES

[1] C. J. Alpert, G. Gandham, J. Hu, J. L. Neves, S. T. Quay, and S. S. Sapatnekar. A Steiner tree construction for buffers, blockages, and bays. *IEEE Transactions on Computer-Aided Design*, 20(4):556–562, April 2001.

[2] C. J. Alpert, J. Hu, S. S. Sapatnekar, and P. G. Villarrubia. A practical methodology for early buffer and wire resource allocation. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 189–194, 2001.

[3] C.J. Alpert, G. Gandham, M. Hrkic, J. Hu, A.B. Kahng, J. Lillis, B. Liu, S.T. Quay, S.S. Sapatnekar, and A.J. Sullivan. Buffered Steiner trees for difficult instances. *IEEE Transactions on Computer-Aided Design*, 21(1):3–14, January 2002.

[4] J. Cong. Challenges and opportunities for design innovations in nanometer technologies. SRC Design Sciences Concept Paper, 1997.

[5] J. Cong and X. Yuan. Routing tree construction under fixed buffer locations. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 379–384, 2000.

[6] W. Donath, P. Kudva, L. Stok, P. Villarrubia, L. Reddy, A. Sullivan, and K. Chakraborty. Transformational placement and synthesis. In *Proceedings of Design, Automation and Test in Europe Conference*, pages 194–201, 2000.

[7] W. C. Elmore. The transient response of damped linear networks with particular regard to wideband amplifiers. *Journal of Applied Physics*, 19:55–63, January 1948.

[8] M. Hrkic and J. Lillis. Buffer tree synthesis with consideration of temporal locality, sink polarity requirements, solution cost and blockages. In *Proceedings of the ACM International Symposium on Physical Design*, pages 98–103, 2002.

[9] M. Hrkic and J. Lillis. S-tree: A technique for buffered routing tree synthesis. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 578–583, 2002.

[10] J. Hu, C.J. Alpert, S.T. Quay, and G. Gandham. Buffer insertion with adaptive blockage avoidance. In *Proceedings of the ACM International Symposium on Physical Design*, pages 92–97, 2002.

[11] A. Jagannathan, S.-W. Hur, and J. Lillis. A fast algorithm for context-aware buffer insertion. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 368–373, 2000.

[12] M. Lai and D.F. Wong. Maze routing with buffer insertion and wiresizing. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 374–378, 2000.

[13] J. Lillis, C. K. Cheng, and T. Y. Lin. Optimal wire sizing and buffer insertion for low and a generalized delay model. *IEEE Journal of Solid-State Circuits*, 31(3):437–447, March 1996.

[14] J. Lillis, C. K. Cheng, and T. Y. Lin. Simultaneous routing and buffer insertion for high performance interconnect. In *Proceedings of the Great Lake Symposium on VLSI*, pages 148–153, 1996.

[15] X. Tang, R. Tian, H. Xiang, and D.F. Wong. A new algorithm for routing tree construction with buffer insertion and wire sizing under obstacle constraints. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 49–56, 2001.

[16] L. P. P P. van Ginneken. Buffer placement in distributed RC-tree networks for minimal elmore delay. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 865–868, 1990.

[17] H. Zhou, D. F. Wong, I-M. Liu, and A. Aziz. Simultaneous routing and buffer insertion with restrictions on buffer locations. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 96–99, 1999.

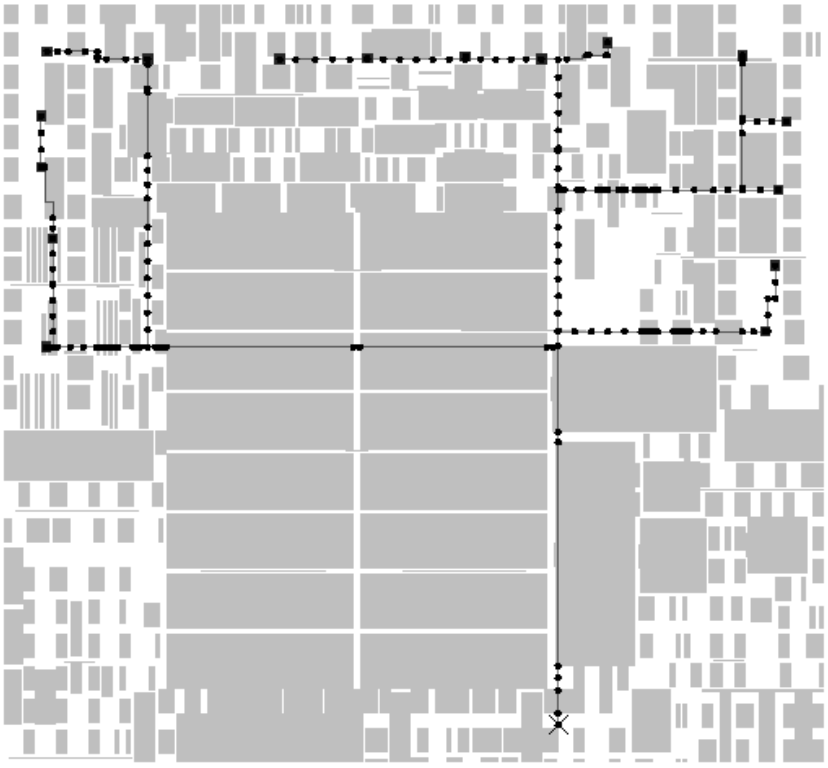


Figure 8: An example of Steiner tree from baseline methodology. The shaded rectangles are buffer blockages. The source is indicated by a cross.

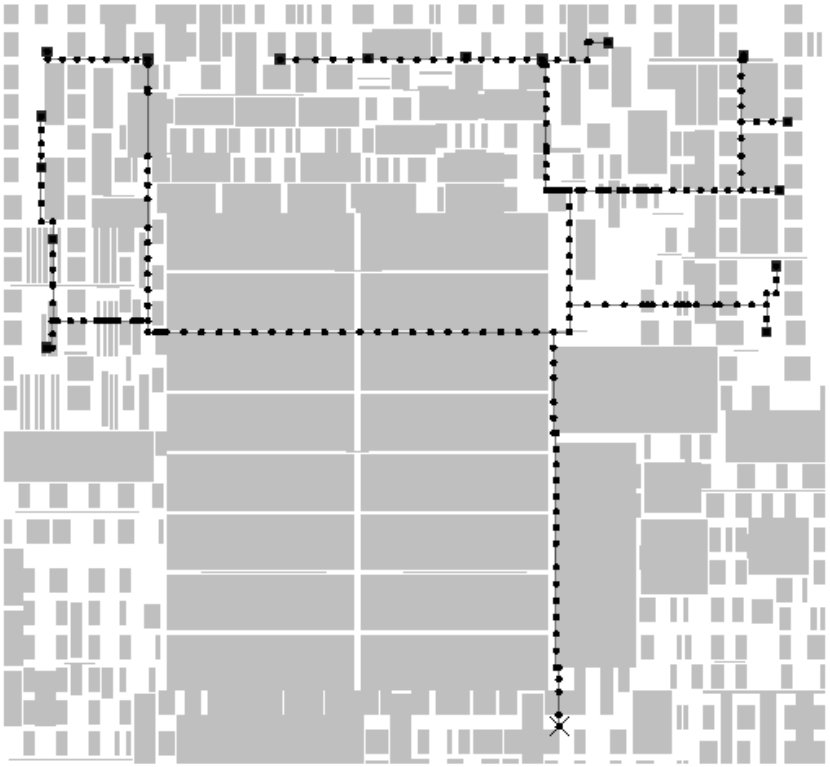


Figure 9: The Steiner tree obtained through porosity aware method on the same net as in Figure 8.