Short Papers

Buffer Insertion With Adaptive Blockage Avoidance

Jiang Hu, Charles J. Alpert, Stephen T. Quay, and Gopal Gandham

***Abstract*—Buffer insertion is a fundamental technology for very large scale integration interconnect optimization. This work presents the repeater insertion with adaptive tree adjustment (RIATA) heuristic that directly extends van Ginneken's classic algorithm to handle blockages in the layout. Given a Steiner tree containing a Steiner point that overlaps a blockage, a local adjustment is made to the tree topology that enables additional buffer insertion candidates to be considered. This adjustment adapts to the demand on buffer insertion and is incurred only when it facilitates the maximal slack solution. RIATA can be combined with any performance-driven Steiner tree algorithm and permits various solution search schemes to achieve different solution quality and runtime tradeoffs. Experiments on several large nets confirms that high-quality solutions can be obtained through this technique with greater efficiency than simultaneous approaches.**

***Index Terms*—Buffer insertion, deep submicrometer, interconnect, layout, physical design, very large scale integration.**

I. Introduction

Buffer insertion is now widely recognized as a key technology for improving very large scale integration interconnect performance. Cong [4] projects that as many as 800 000 buffers will be required for designs in 50-nm technologies. As design complexity increases, designers are relying on an increasing number of IP cores, large memory arrays, and hierarchical components, i.e., designs are becoming "chunkier." For a buffer insertion technique to be effective, it must be fully aware of its surrounding blockage constraints while also being efficient enough to quickly process thousands of nets. In the buffer insertion literature, van Ginneken's dynamic programming-based algorithm [13] has established itself as a classic in the field.

Prior to buffer insertion, several large area chunks may already be occupied by macro or IP blocks for which wires can be routed over the blocks, but buffers cannot be inserted inside the blocks. We call these regions "buffer blockages." For example, Fig. 1(a) shows a Steiner tree with three-pins and a buffer blockage. Let the required arrival times for the sinks be $\mathrm{rat}(v_1) = 200$ and $\mathrm{rat}(v_2) = 100$. If the blockage is ignored, one can obtain a good solution as shown in Fig. 1(b). Here, the buffer acts to decouple the load of the branch v_1 from the more critical sink v_2. Of course, in practice, one cannot ignore the buffer blockage and a solution other than that in Fig. 1(b) must be sought. If one restricts the solution space to the existing Steiner topology, the two best solutions are shown in Fig. 1(c) and (d), but neither solution meets the required timing constraints.

The authors of [9], [10], and [14] proposed optimal algorithms on finding a minimum delay buffered *path* with buffer blockages. In [6], Cong and Yuan proposed a dynamic programming algorithm, called recursively merging and pruning (RMP), to handle the multisink net

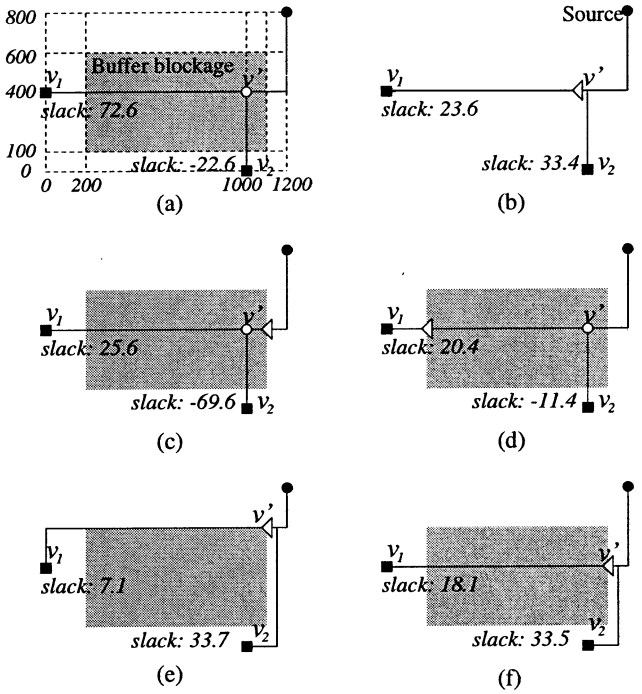

Fig. 1. Steiner tree and buffer solutions on a three-pin net with one buffer blockage.

buffer insertion with location restrictions. RMP is designed for the buffer block methodology [5] for which the number of legal buffer locations is quite limited. It works on a grid graph that is constructed by adding horizontal and vertical lines through each potential buffer locations to the Hanan grid. It not only explores almost every node on the grid in tree construction but also considers many sink combinations in subsolutions. Consequently, RMP tends to be slow when either the number of net pins or legal buffer locations is large. Nevertheless, RMP generally yields near optimal solutions in term of timing performance. More recently, Tang *et al.* suggested a graph-based algorithm [12] on a similar problem. While more efficient than RMP, it can optimize only the maximum sink delay rather than the minimum slack.

Difficult buffering problems occur not just with large nets but also when sink polarity constraints are present. Alpert *et al.* developed the "buffer-aware" C-Tree heuristic [3] to be used as a precursor to van Ginneken's algorithm. However, the method is not "blockage-aware." To solve this one could first run C-Tree, then invoke the algorithm of [2], which performs local rerouting to avoid the blockages without adding too much wiring. Then, this modified tree is passed to van Ginneken's buffer insertion algorithm. For example, this approach would obtain the buffered solution in Fig. 1(e). However, a carefully constructed timing-driven topology can be destroyed by these local topology changes, making the final slack significantly worse than not running local rerouting at all.

Hence, despite all the work in this field, there is still no fast and effective solution for multisink nets. We seek an algorithm that can find the solution in Fig. 1(f) for our example. This optimal solution for maximizing the minimum slack may be obtained by simply sliding the buffer insertion solution in Fig. 1(b) to its closest legal location. While an approach like this works for this example, it fails for a larger-sized

Manuscript received May 31, 2002; revised September 29, 2002. This paper was recommended by Guest Editor S. S. Sapatnekar.

J. Hu is with the Department of Electrical Engineering, Texas A&M University, College Station, TX 77843 USA (e-mail: jianghu@ee.tamu.edu).

C. J. Alpert and S. T. Quay are with IBM Corporation, Austin, TX 78758 USA.

G. Gandham is with IBM Corporation, Hopewell Junction, NY 12533 USA.

Digital Object Identifier 10.1109/TCAD.2003.809647

tree with multiple branching nodes because the proper buffer insertion solution for one branching node relies on the buffer solution for another branching node. Therefore, moving each buffer out of blockage individually can ruin the integrity of the buffer insertion solution. Moreover, if a buffer is moved too far away from its original location, the solution quality may degrade beyond repair.

Consequently, we propose to handle blockages during buffer insertion directly within van Ginneken's algorithm. Our technique adjusts a given tree topology according to the demand on buffer insertion, and such adjustments only occur when it facilitates the maximal slack solution. This technique can be used with any performance-driven Steiner tree algorithm. Experimental results show that we can obtain greater efficiency than methods that use the entire Hanan grid.

The remainder of the paper is organized as follows. Section II formulates the problem we wish to solve and reviews van Ginneken's algorithm. The algorithm is described in Section III. We show the experimental results in Section IV and conclude this paper in Section V.

II. Preliminaries

For the Steiner tree construction, let V_{internal} represent the set of nodes in the tree other than the source and sinks. The problem we address is formulated as follows.

Problem Formulation: Given a net $N = \{v_0, v_1, v_2, \ldots, v_n\}$ with source v_0, sinks $v_1, \ldots, v_n$, load capacitances $c(v_i)$, and required arrival time $q(v_i)$ for each sink $v_i \in N$, a set of rectangles $R = \{r_1, r_2, \ldots r_k\}$ representing buffer blockages, and a buffer library $B = \{b_1, b_2, \ldots b_m\}$, find a buffered Steiner tree $T(V, E)$ where $V = N \cup V_{\text{internal}}$ and E spans every node in V such that the required arrival time at the source is maximized.

The formulation is similar to the formulation for RMP [6] except that a set of legal buffer locations is given in RMP instead of a set of buffer blockages. To transform our formulation to that proposed in [6], we can extend the borders of blockages over a Hanan grid and label each node on the new grid graph as either a legal or infeasible buffer location.

We adopt the Elmore delay model [8] for interconnect and an *RC* switch model for gate delays. We assume that the given routing tree $T(V, E)$ is a binary tree, i.e., every internal node has no more than two children and that every sink has degree one. Any routing tree can be easily transformed to satisfy both conditions by inserting zero-length pseudo edges.[1]

Since we propose to extend the van Ginneken's algorithm to directly handle buffer blockages, we first overview the algorithm to form a basis for the remainder of the discussion. Van Ginneken's algorithm proceeds bottom-up from the leaf nodes along a given tree topology toward the source node. A set of candidate solutions is computed for each node during this process. A candidate solution at a node v is characterized by the load capacitance $c(v)$ seen downstream v and the required arrival time $q(v)$ at node v. We use a pair $s = \Big(c(v), q(v)\Big)$ to specify a buffering solution at v. For any two candidate solutions $s_1 = \Big(c_1(v), q_1(v)\Big)$, and $s_2 = \Big(c_2(v), q_2(v)\Big)$, s_1 is *dominated* by (inferior to) s_2 if $c_1(v) \geq c_2(v)$ and $q_1(v) \leq q_2(v)$. A candidate solution set $S(v) = \{s_1, s_2, \ldots\}$ is a nondominating set if no solution in this set is dominated by any other solution in this set. During the bottom-up process of van Ginneken's algorithm, the candidate solutions at leaf node evolve through the following operations.

- *Grow* $\Big(S(v), w\Big)$: Propagate candidate set $S(v)$ from node v to node w to get $S(w)$. If the wire between v and w has a resistance of R and capacitance C, we can get $c_i(w) = c_i(v) + C$ and $q_i(w) = q_i(v) - R\Big(C/2 + c_i(v)\Big)$ for each $(c_i(v), q_i(v)) \in S(v)$, and obtain $S(w)$ from the solution pairs $\Big(c_i(w), q_i(w)\Big)\ \forall\ i$.
- *AddBuffer* $\Big(S(v)\Big)$: Insert buffer at v and add the new candidate into $S(v)$. If a buffer b has an input capacitance c_b, output resistance r_b and intrinsic delay t_b, we can obtain $c_{i,\text{buf}}(v) = c_b$ and $q_{i,\text{buf}}(v) = q_i(v) - r_b c_i(v) - t_b$ for each $\Big(c_i(v), q_i(v)\Big) \in S(v)$ and add the pair $\Big(c_{i,\text{buf}}(v), q_{i,\text{buf}}(v)\Big)\ \forall\ i$ with the maximum $q_{i,\text{buf}}$ into $S(v)$.
- *Merge* $\Big(S_l(v), S_r(v)\Big)$: 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 $\Big(c_{j,\text{left}}(v), q_{j,\text{left}}(v)\Big)$ from the left child and a solution $\Big(c_{k,\text{right}}(v), q_{k,\text{right}}(v)\Big)$ from the right child, the merged solution $\Big(c_i(v), q_i(v)\Big)$ is obtained through letting $c_i(v) = c_{j,\text{left}}(v) + c_{k,\text{right}}(v)$ and $q_i(v) = \min\Big(q_{j,\text{left}}(v), q_{k,\text{right}}(v)\Big)$.
- *PruneSolutions* $\Big(S(v)\Big)$: Remove any solution $s_1 \in S(v)$ that is dominated by any other solution $s_2 \in S(v)$.

After a set of candidate solutions are propagated to the source, the solution with the maximum required arrival time is selected for the final solution. For a fixed routing tree, this algorithm can find the optimal solution in $O(n^2)$ time if there are n pins in this net. If we do wire segmenting, n should be the number of candidate insertion points.

III. Repeater Insertion With Adaptive Tree Adjustment (RIATA) Algorithms

A. Strategy

A common strategy to solve a sophisticated problem is divide-and-conquer, i.e., partitioning a complex problem into a set of subproblems in manageable scales. Such partitioning can be performed on either physical or design flow aspects. For example, a large net can be physically clustered into smaller nets as in C-Tree. Such partitioning not only speeds up the problem-solving process, but also isolates subproblems according to their natures so that scattered targets can be avoided and the optimization can be well focused. Separating the Steiner tree construction from buffer insertion procedure is an example of partitioning the design flow. An initial Steiner tree construction can limit the buffer solution search along an anticipated good direction. A directional search is obviously more efficient than the simultaneous routing and buffer insertion which is an implicitly brute-force search, even though the search may intelligently prune some unnecessary candidate solutions. This design flow partitioning is shown to be effective in the C-Tree work [3].

When considering how to incorporate blockage constraints, we need to partition it into the right phase in the design flow. Blockage avoidance is more tied with request on buffering solutions, i.e., it is hard to know how to make a Steiner tree avoid blockages without knowing where buffers are needed. A simultaneous approach is not efficient while separate routing and buffer insertion approach as in [2] cannot adequately plan for blockages. However, we can move the partitioning line to the middle of these two approaches, i.e., we can generate a Steiner tree which is allowed to be adjusted during buffer insertion construction according to dynamic requests for buffer blockage avoidance. Our key idea is to explore just a handful of alternative buffer insertion locations for which the tree topology can be modified (as opposed to an approach like buffered P-Tree [11] which explores a much larger space). These locations correspond to moving a branch node outside

[1]Note that the choice of which subtrees to group together can have an effect on solution quality. Grouping the subtrees together in a nonoptimal way generally has limited effect on timing quality, but may waste buffers that have to be inserted for decoupling. Our implementation arbitrarily groups the child nodes.

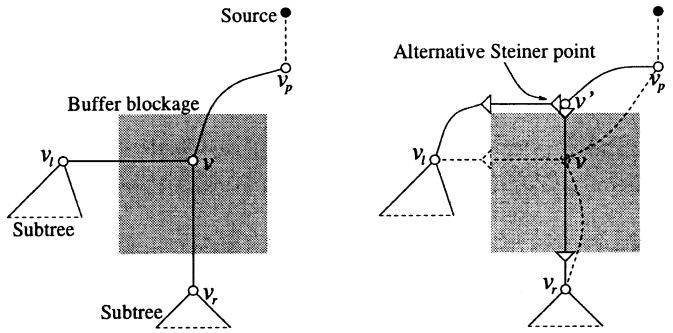

Fig. 2. For a Steiner point v within a buffer blockage as in (a), the three buffer locations closest to v can be found as in (b).

a blockage which enables opportunities for decoupling and efficient driving of long paths.

Buffer blockages along paths that do not contain any Steiner nodes can be mitigated relatively easily by allowing them to take multibend route without increasing wirelength. This type of solution can be achieved by applying the work in [2] to obtain a Steiner tree that has L-shapes and Z-bends that minimize overlap with blockages but no additional wirelength or tree topology adjustment. The difficult buffer blockage problems occur when a Steiner node lies on top of blockage which eliminates opportunities for decoupling noncritical paths and for driving long wires directly. Hence, our key idea is to consider generating alternative candidate solutions within van Ginneken's algorithm by trying an alternate location outside of blockage for the branching Steiner node.

B. Basic RIATA

Given a Steiner tree, we extend van Ginneken's algorithm so that the tree topology is adaptively adjusted during the bottom-up candidate solution propagation process, i.e., buffer insertion is not restricted to a fixed topology anymore. During the bottom-up propagation process, if a Steiner point does not overlap a buffer blockage, our algorithm proceeds in the same way as van Ginneken's algorithm. The difference occurs when a Steiner point is within a buffer blockage, as depicted in Fig. 2(a). To compensate for the inability to have possible buffer insertion candidates near the blocked Steiner point, we seek alternative unblocked sites nearby to use instead. For the sake of simplicity, the alternative point is searched only between node v and its parent node v_p. Within the bounding box between them, we search for an unblocked point which is the closest to v. Other searching schemes will be introduced in the Section III-C. In our example, Fig. 2(b) shows the result of searching for the unblocked point v' closest to v on the path between v and its parent node v_p. After we obtain this alternative Steiner point, we may generate an alternative tree topology to allow buffer insertions at the adjusted branch node v'. Before we propagate the candidate solutions from the children nodes v_l and v_r, we search for the least blocked path to their parent nodes v and v' through the technique presented in [2]. By carefully choosing the cost, this technique can provide a path connecting a node v_i and another node v_j such that the path length is the shortest and the total path length overlapping with buffer blockages is minimized. We query this method as a subroutine *LeastBlockedPath* (v_i, v_j) to find the detailed path between children node v_l and v_r, and parent node v and v' followed by wire segmenting [1]. Next, we propagate candidate solutions from v_l and v_r through the least blocked paths to both v and v'. Note that during this propagation process, more candidate solutions may be generated by inserting buffers at segmenting points along the paths *and at node* v'. Then, the candidate solutions at v and v' can be further propagated to their parent node v_p in the next stage. The adjustment on the Steiner point is a part of the candidate solutions, thus, it only actually becomes part of the new tree if the final solution is generated via this alternative path. Performing the construction in this manner also guarantees that our approach will perform at least as well as the original van Ginneken algorithm. Since the alternative Steiner points are searched along a constructed Steiner tree, the solution space is quite limited compared with the simultaneous approach.

C. RIATA+ Algorithm

In Section III-B, we introduced the basic RIATA heuristic in which only one alternative point is searched for each Steiner node between itself and its parent node. When a Steiner node and its parent node are both in the same blockage, no alternative point will be found. This is illustrated in the example in Fig. 3(a) where five Steiner nodes are within the same blockage. If we apply the basic RIATA technique, we can find the alternative Steiner point for only Steiner node v in Fig. 3(a) [the alternative unblocked node v' is shown in Fig. 3(b)]. In order to allow unblocked alternative points for other Steiner nodes in this difficult case, we need to expand the search range.

We illustrate this enhanced search scheme through the example in Fig. 4. Fig. 4(a) shows that there are two neighboring Steiner node v_i and v_j in the same buffer blockage. When van Ginneken's algorithm proceeds to node v_i, we consider four alternative points on the four sides of the blockage. They are the four crosses named as $v_{i,l}$, $v_{i,t}$, $v_{i,r}$, and $v_{i,b}$ in Fig. 4(a). We define the *expanded Steiner node set* $\tilde{V}(v_i)$ associated with v_i as $\tilde{V}(v_i) = \{v_i, v_{i,l}, v_{i,t}, v_{i,r}, v_{i,b}\}$[2]. Similar to the basic RIATA algorithm, the candidate solutions at children node v_a and v_b are propagated to *every* Steiner point $v \in \tilde{V}(v_i)$ and are merged there.

If the parent node v_j of v_i is a Steiner node in a blockage as in Fig. 4, an expanded Steiner node set $\tilde{V}(v_j)$ is generated as in Fig. 4(b). Since the candidate solutions at $\tilde{V}(v_i)$ will be propagated from five different points to five other different nodes in $\tilde{V}(v_j)$, there will be 25 combinations. However, if one of the combinations causes a path detour, the resulting solutions are generally inferior to those without path detours. For example, if we propagate candidate solutions from $v_{i,t}$ to $v_{j,b}$, a large path detour will be incurred. This observation tells us that certain combinations can be pruned out without significantly affecting the solution quality. In order to specify the pruning scheme, we define the *nearest unblocked ancestor* of a node as the first unblocked node encountered when we trace from this node upstream toward the source. For a child Steiner node v_i at location (x_i, y_i) and its parent node v_j at location (x_j, y_j), both of which are overlapped with blockages, with the nearest unblocked ancestor of v_j as node v_c at (x_c, y_c), we call the propagation from v_i to v_j as monotone if $x_j = \text{median}(x_i, x_j, x_c)$ and $y_j = \text{median}(y_i, y_j, y_c)$. We do not simply choose v_j's immediate parent node v_p as reference point, because v_p may be in a blockage and its alternative point may invalidate the monotone property defined at v_p itself. In the example in Fig. 4, the nearest unblocked ancestor of v_j is coincidentally the same as v_p. When we propagate candidate solutions from each node of $\tilde{V}(v_i)$ to each node of $\tilde{V}(v_j)$, any monotone propagation is allowed. In addition, we always allow propagation from v_i to any node in $\tilde{V}(v_j)$ to ensure there is at least one set of solutions propagated to every node in $\tilde{V}(v_j)$. Any nonmonotone propagation from a node other than v_i is disallowed. For the example in Fig. 4(c), candidate solutions at $v_{i,t}$ are not allowed to be propagated to v_j. Monotone propagations from $v_{i,t}$ to $v_{j,t}$ and from v_i to $v_{j,t}$ are illustrated in Fig. 4(c). Note that the candidate solutions from v_a in Fig. 4 may come from an expanded Steiner

[2]If v_i is not a Steiner node or it is not in any blockages, then $\tilde{V}(v_i) = \{v_i\}$.

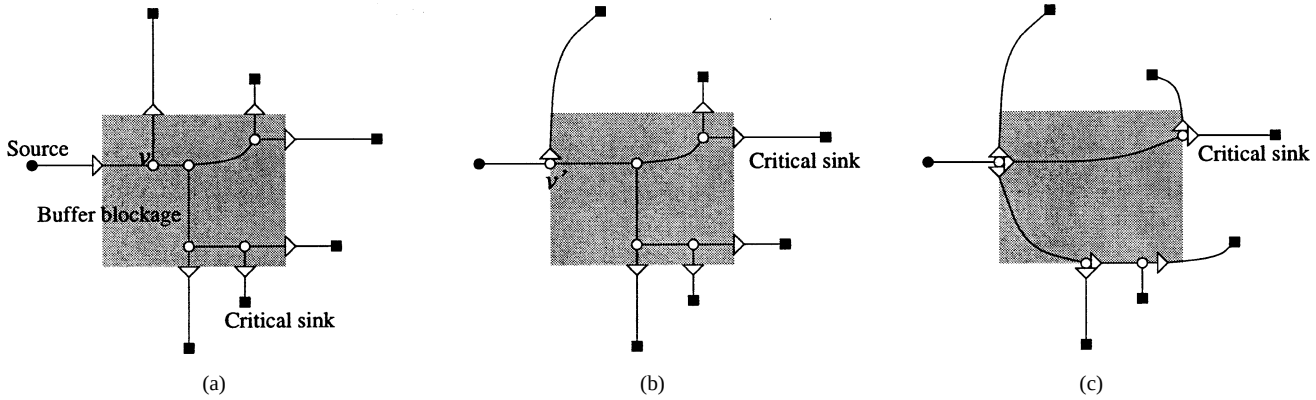

Fig. 3. Hard case that many Steiner nodes are within the same blockage. The original Steiner tree is shown in (a) with slack of -101 ps. Performing buffer insertion on this fixed topology will improve the slack to 112 ps. If we run the basic RIATA heuristic, we can obtain solution as in (b) with slack of 148 ps. If we apply the enhanced RIATA+ heuristic, we can obtain solution in (c) with slack of 369 ps.

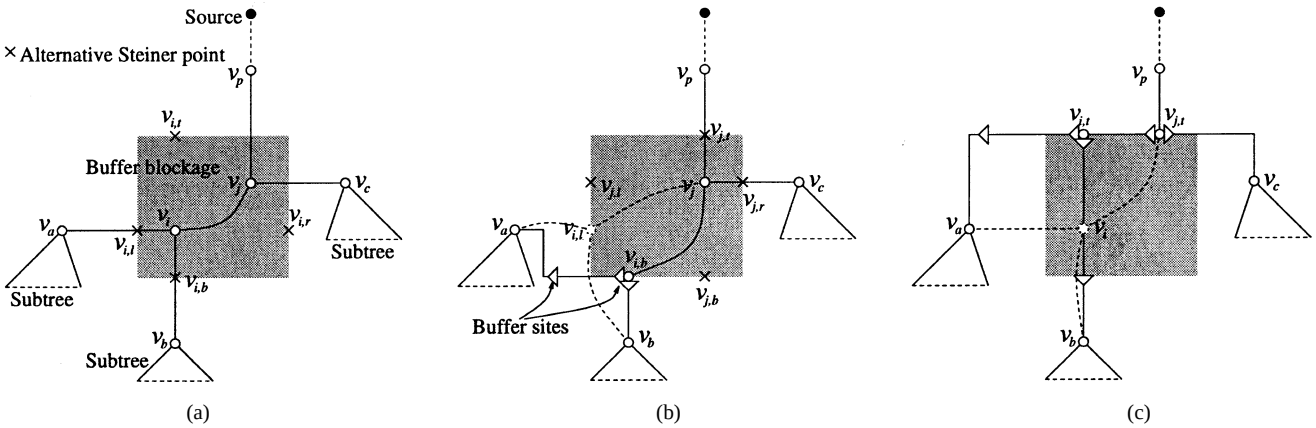

Fig. 4. (a) Example of a Steiner tree with Steiner nodes v_i and v_j in a blockage. (b) Finding alternative buffered Steiner node $v_{i,l}$ and $v_{i,b}$ for v_i. (c) Candidate solutions from $v_{i,t}$ and v_i are propagated to alternative Steiner node $v_{j,t}$.

node set $\tilde{V}(v_a)$ as well and, similarly, any nonmonotone propagation from a node in $\tilde{V}(v_a)$ except v_a to $\tilde{V}(v_i)$ is prohibited.

Searching alternative Steiner points on four boundaries of the blockage guarantees that alternative points can always be found unless the whole chip area is blocked. Furthermore, this search scheme allows Steiner nodes to be spread out around the blockage if there are multiple Steiner nodes in the same blockage as shown in Fig. 3(c). If we consider only one alternative point for each Steiner node, the alternative Steiner nodes may be crowded on the single side of the blockage. We allow candidate solutions from v_i to be propagated to every node in $\tilde{V}(v_j)$ in Fig. 4 for the same reason.

To implement this heuristic, we need to efficiently find the closest unblocked node set $\tilde{V}(v)$ for a node v. Given a node v and a set of rectangles $R = \{r_1, r_2, \ldots r_k\}$ representing the buffer blockages, if node v is within a blockage $r \in R$, we need to find the unblocked points which is the closest to v on each boundary of r. If there is no overlap between any two buffer blockages, all we need to do is to locate the rectangle r that overlaps v. If r is defined by bounding coordinates $(x_{lo}, y_{lo}, x_{hi}, y_{hi})$ and v is located at point (x_v, y_v), the unblocked points closest to v on each boundary of r are (x_{lo}, y_v), (x_v, y_{hi}), (x_{hi}, y_v) and (x_v, y_{lo}). If the set of rectangles R is stored as an interval tree [7], the desired rectangle r can be found in $O(k)$ time in the worst case. We let *UnblockedNodes* (v, R) denote the procedure that finds such an unblocked node set.

We call the enhanced heuristic RIATA+. The complete RIATA+ algorithm is described in Fig. 5. The subroutine of propagating candidate solutions from one node set to another is shown in Fig. 6. We use the symbol $\rightsquigarrow$ to represent a propagation. If there are no buffer blockages, i.e., the condition in line 2 of Fig. 5 is always false; this algorithm is essentially the same as the van Ginneken's algorithm. If we define

Procedure: *FindCandidates*(v)
Input: Current node v to be processed
Output: Candidate solution set $S(\tilde{V}(v))$ at $\tilde{V}(v)$
Global: Steiner tree $T(V, E)$
Buffer library $B = \{b_1, b_2, ...\}$
Rectangles $R = \{r_1, r_2, ...\}$

1. $\tilde{V}(v) \leftarrow \{v\}$
2. If v is a Steiner node in a blockage
   $\tilde{V}(v) \leftarrow \tilde{V}(v) \cup UnblockedNodes(v, R)$
3. $S(\tilde{V}(v)) \leftarrow \emptyset$
4. If v is a sink
   $S(\tilde{V}(v)) \leftarrow S(\tilde{V}(v)) \cup \{(c(v), q(v))\}$
   Return $S(\tilde{V}(v))$
5. $v_l \leftarrow$ left child node of v
   $S(\tilde{V}(v_l)) \leftarrow FindCandidates(v_l)$
6. $S_l(\tilde{V}(v)) \leftarrow Propagate(S(\tilde{V}(v_l)), \tilde{V}(v))$
7. If v has only one child, Return $S_l(\tilde{V}(v))$
8. $v_r \leftarrow$ right child node of v
   $S(\tilde{V}(v_r)) \leftarrow FindCandidates(v_r)$
9. $S_r(\tilde{V}(v)) \leftarrow Propagate(S(\tilde{V}(v_r)), \tilde{V}(v))$
10. $S(\tilde{V}(v)) \leftarrow Merge(S_l(\tilde{V}(v)), S_r(\tilde{V}(v)))$
11. Return $S(\tilde{V}(v))$

Fig. 5. Core algorithm of RIATA+.

Procedure: $Propagate(S(\tilde{V}(v_i)), \tilde{V}(v_j))$
Input: Candidate solutions at $S(\tilde{V}(v_i))$
Node set $\tilde{V}(v_j)$
Output: Candidate solution set $S(\tilde{V}(v_j))$ at $\tilde{V}(v_j)$

0. $S(\tilde{V}(v_j)) \leftarrow \emptyset$
1. For each node $v_{i,k} \in \tilde{V}(v_i)$
2. For each node $v_{j,l} \in \tilde{V}(v_j)$
3. If $v_{i,k} \rightsquigarrow v_{j,l}$ is monotone or $v_{i,k} == v_i$
4. $P \leftarrow LeastBlockedPath(v_{i,k}, v_{j,l})$
5. $DoSegmenting(P)$
6. $v_p \leftarrow v_{i,k}$
7. While $v_q \leftarrow$ node next to v_p in P
$S(v_q) \leftarrow AddBuffer(Grow(S(v_p), v_q))$
$v_p \leftarrow v_q$
8. $S(\tilde{V}(v_j)) \leftarrow S(\tilde{V}(v_j)) \cup S(v_p)$
9. For each node $v_{j,l} \in \tilde{V}(v_j)$
$PruneSolutions(S(v_{j,l}))$
10. Return $S(\tilde{V}(v_j))$

Fig. 6. Subroutine of propagating candidate solutions from one node set to another node set.

$\tilde{V}(v_i)$ to include only v_i and its nearest unblocked point between v_i and its parent v_j, this description applies for the basic RIATA heuristic introduced in Section III-B. Actually, there could be many other ways on defining $\tilde{V}(v_i)$ to achieve different solution quality and runtime tradeoffs. For example, we can include more alternative Steiner points in $\tilde{V}(v_i)$ or even allow nonmonotone propagations when the net is extremely timing critical or its size is small.

D. Complexity

Given a net with n insertion points and m pins, a buffer library B and k rectangles representing blockages, if the maximal candidate solution set size is g and the maximal expanded Steiner node set size is h, then the complexity of our heuristic is $O(g \cdot n \cdot |B| \cdot h^2 + m \cdot k)$. The term of $m \cdot k$ comes from the operations of searching unblocked alternative Steiner points. Obviously, $h = 2$ for the RIATA heuristic and $h = 5$ for the RIATA+ heuristic. We may assume that the capacitance value in each candidate solution can take only a polynomially bounded integer, thus, the complexity of our heuristic is pseudopolynomial.

IV. Experimental Results

We implemented all of the codes in C++ and performed the experiments on a SUN Ultra-10 workstation with 2 GB of memory. Without loss of generality, we use only one buffer type in our buffer library and no negative sink polarity is considered. For all experiments, we use C-Tree to generate the initial timing-driven Steiner tree, whenever one is required.

A. Experiments on Large Nets

We obtained seven big nets from industrial designs and generated buffer blockages arbitrarily. The number of pins and buffer blockages for each net are listed in columns 2 and 3 in Table I, respectively. In experiments, we compared the tree performances of the following approaches.

- NoBuf: This is C-Tree without any buffer insertion, which gives a baseline for comparison.
- VGNB: This is van Ginneken's algorithm on C-Tree ignoring buffer blockages completely. This serves as a type of crude upper bound on how well the other approaches are handling blockages.

TABLE I
Experimental Results on Large Nets

Net	Algo	Slack(ps)	# buf	wire	CPU(sec)
n873	NoBuf	-867	0	4750	-
	VGNB	553	4	4750	0.68
21 pins	VG	108	3	4750	0.16
	RePath	110	3	4750	0.18
6 blks	RIATA	339	3	5034	0.23
	RIATA+	481	4	6054	6.47
n189	NoBuf	-1420	0	5843	-
	VGNB	540	6	5843	3.17
30 pins	VG	264	5	5843	0.38
	RePath	344	5	5843	0.36
15 blks	RIATA	441	5	5845	0.85
	RIATA+	539	6	6029	55.27
n786	NoBuf	-848	0	5318	-
	VGNB	74	4	5318	2.27
33 pins	VG	-192	2	5318	0.24
	RePath	-190	2	5318	0.29
15 blks	RIATA	6	3	5319	0.62
	RIATA+	56	4	6555	24.44
n870	NoBuf	-2835	0	4764	-
	VGNB	179	9	4764	5.61
44 pins	VG	105	4	4764	1.09
	RePath	105	4	4764	1.51
16 blks	RIATA	128	6	5030	4.73
	RIATA+	177	10	5069	142.32
big1	NoBuf	-214	0	9407	-
	VGNB	1044	6	9407	2.45
64 pins	VG	711	7	9407	1.07
	RePath	733	7	9407	1.08
7 blks	RIATA	883	8	9441	1.62
	RIATA+	1041	7	10269	37.58
big2	NoBuf	-1560	0	12448	-
	VGNB	-41	13	12448	8.44
80 pins	VG	-567	6	12448	1.28
	RePath	-567	6	12448	1.28
7 blks	RIATA	-171	7	12765	2.28
	RIATA+	-61	13	13192	106.98
big3	NoBuf	-798	0	19669	-
	VGNB	1575	7	19669	22.25
89 pins	VG	1101	6	19669	3.95
	RePath	1101	6	19669	3.86
21 blks	RIATA	1382	8	20517	11.98
	RIATA+	1577	7	20616	713.22

- VG: This is van Ginneken's algorithm where blockage constraints are obeyed by labeling nodes that overlap blockages as infeasible. For every wire segment partially contained within a blockage, an additional buffer insertion location is considered on the point where the wire and blockage intersect.
- RePath: For a C-Tree topology, each path between two nodes v and w is rerouted if it overlaps with any buffer blockages. Note that v and w can only be either a source node, a sink node or a branch node. The paths are rerouted as in [2] such that their path lengths are not changed while the total path length overlapping with blockage area is minimized. Then, after segmenting, van Ginneken's algorithm is performed. This corresponds to the proposed flow of first constructing a Steiner tree (C-Tree), then making obvious, nonintrusive changes to the topology to avoid blockages, followed by van Ginneken buffer insertion [3].
- RIATA: This is the basic version of our algorithm introduced in Section III-B that adaptively adjusts VG to consider only one more alternate point for each Steiner node. Note that the candidate solutions generated by RIATA are a superset of VG, and, thus, RIATA is guaranteed to perform no worse than VG.
- RIATA+: This is the RIATA+ algorithm introduced in Section III-C. RIATA+ candidates are generally a superset of RIATA candidates, though not strictly so due to potential segmenting differences on alternative paths.

One can observe the following from the maximum slack results in column 5 of Table I. First, buffer insertion certainly is a worthwhile operation, as all buffer insertion methods significantly improve on the solution NoBuf, which has no buffering. Second, RIATA and RIATA+ usually yield significantly better slack results than and VG and RePath. Finally, the solutions from RIATA+ are always superior to others and are almost as good as VGNB in which blockages are ignored entirely. Actually, RIATA+ outperforms VGNB for net $big3$, because it finds a better topology than C-Tree, even though RIATA+ obeys blockages while VGNB does not.

In addition to timing performance, we show major resource expenses (number of buffers, wirelength, and CPU time in seconds) in the rightmost three columns of Table I. We observe that RIATA uses roughly the same number of buffers as VG or RePath while the number of buffers used by RIATA+ is close to VGNB. Also, RIATA does not significantly increase the wirelength of the low wirelength VG solutions while the wirelength increase from RIATA+ is greater. There is a moderate increase in CPU time of RIATA versus VG, but they are virtually in the same order. Not unexpectedly, RIATA+ consumes significantly larger CPU time, since it explores a much larger solution space. Therefore, RIATA and RIATA+ provide different solution quality and runtime tradeoff. In the Section V, we will show that even RIATA+ is much faster than the simultaneous approach.

B. Comparisons With RMP

Besides VG and RePath, our next set of experiments compare RIATA/ RIATA+ with the RMP algorithm [6], since its problem formulation is almost same as ours. We obtained the executable of RMP from the authors of [6]. As RMP is designed for relatively small nets, we perform comparisons on sets of industrial nets with fewer sinks than those considered in the previous experiments. In addition, we run RMP in quick mode which is its faster heuristic version. We randomly generate blockages and then construct the corresponding extended Hanan grid for each test case. In order to compare to RMP's formulation, we intentionally marked some legal candidate buffer sites on the Hanan grid as illegal to reduce the solution space since otherwise RMP cannot complete in a reasonable amount of time. Comparisons for RIATA and RMP both use the same set of possible buffer locations. Comparisons giving slack and resource utilization are shown in Table II.

RIATA+ outperforms RMP in six of the nine testcases on slack results. On net n730, the slack from RIATA+ is greater than that from RMP by 116 ps. Furthermore, RIATA+ shows superior advantage on CPU time versus RMP. The best case for RMP is on net n313 where it consumes about the same CPU time as RIATA+. When the number of pins/buffer sites increase, the RMP's CPU time quickly increases to magnitude greater than RIATA/ RIATA+. Clearly, RMP cannot be applied to thousands of nets in a physical synthesis type of optimization. In addition, RMP usually uses more overall wirelength as well.

TABLE II
EXPERIMENTS ON SMALL- AND MIDDLE-SIZED CIRCUITS

Net	Algo	Slack(ps)	# buf	wire	CPU(sec)
n071	NoBuf	183	0	5868	-
	VG	393	1	5868	0.04
8 pins	RePath	393	1	5868	0.05
	RIATA	452	3	5919	0.11
19 buf	RIATA+	559	5	6767	0.77
sites	VGNB	574	6	5868	0.17
	RMP	548	4	7771	11.61
m0s5	NoBuf	22	0	6278	-
	VG	208	1	6278	0.04
8 pins	RePath	294	3	6278	0.05
	RIATA	369	4	7130	0.09
13 buf	RIATA+	454	7	9279	0.87
sites	VGNB	416	7	6278	0.17
	RMP	365	5	10207	8.07
n313	NoBuf	188	0	5547	-
	VG	188	0	5547	0.04
9 pins	RePath	448	2	5547	0.05
	RIATA	491	4	6052	0.10
33 buf	RIATA+	512	6	7204	0.84
sites	VGNB	513	6	5547	0.16
	RMP	529	5	10876	0.79
n730	NoBuf	673	0	1487	-
	VG	694	1	1487	0.05
9 pins	RePath	694	1	1487	0.05
	RIATA	694	1	1487	0.06
15 buf	RIATA+	854	4	1524	0.24
sites	VGNB	941	6	1487	0.22
	RMP	738	3	2491	19.60
pnt3	NoBuf	687	0	5291	-
	VG	787	3	5291	0.06
10 pins	RePath	804	3	5291	0.07
	RIATA	987	3	7250	0.09
19 buf	RIATA+	1028	8	8911	1.00
sites	VGNB	1031	5	5291	0.12
	RMP	1011	6	15093	11.72
m1s9	NoBuf	369	0	4222	-
	VG	493	1	4222	0.05
10 pins	RePath	661	2	4222	0.09
	RIATA	760	4	4648	0.12
17 buf	RIATA+	821	6	5091	1.32
sites	VGNB	837	6	4222	0.23
	RMP	818	6	6554	283.07
n905	NoBuf	284	0	2233	-
	VG	696	1	2233	0.08
11 pins	RePath	713	2	2233	0.10
	RIATA	780	3	3018	0.10
12 buf	RIATA+	811	5	3033	2.11
sites	VGNB	863	6	2233	0.48
	RMP	843	4	3379	109.35
n702	NoBuf	-1095	0	3567	-
	VG	-29	1	3567	0.07
11 pins	RePath	-8	1	3567	0.09
	RIATA	153	3	4197	0.13
17 buf	RIATA+	179	3	5494	0.60
sites	VGNB	200	6	3567	0.42
	RMP	202	3	4294	2645.26
n866	NoBuf	-17	0	6840	-
	VG	353	2	6840	0.10
12 pins	RePath	356	2	6840	0.11
	RIATA	469	3	7730	0.14
26 buf	RIATA+	604	7	8205	2.09
sites	VGNB	613	8	6840	0.31
	RMP	523	7	10587	1083.04

V. CONCLUSION

We propose RIATA/ RIATA+, an adaptive tree adjustment technique that is integrated directly into van Ginneken's classic buffer insertion algorithm to handle buffer blockage constraints. Our experiments show that this simple technique can give significant improvements over van Ginneken's original algorithm at a reasonable CPU cost. Further, it is much faster and nearly as effective as the RMP approach, which searches a much larger solution space.

ACKNOWLEDGMENT

The authors would like to thank X. Yuan and J. Cong for providing the RMP code.

REFERENCES

[1] C. J. Alpert and A. Devgan, "Wire segmenting for improved buffer insertion," in *Proc. ACM/IEEE Design Automation Conf.*, 1997, pp. 588–593.
[2] 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 Trans. Computer-Aided Design*, vol. 20, pp. 556–562, Apr. 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 Trans. Computer-Aided Design*, vol. 21, pp. 3–14, Jan. 2002.
[4] J. Cong. (1997) Challenges and opportunities for design innovations in nanometer technologies. SRC Design Sciences Concept Paper. [Online]. Available: http://www.src.org

[5] J. Cong, T. Kong, and D. Z. Pan, "Buffer block planning for interconnect-driven floorplanning," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 1999, pp. 358–363.
[6] J. Cong and X. Yuan, "Routing tree construction under fixed buffer locations," in *Proc. ACM/IEEE Design Automation Conf.*, 2000, pp. 379–384.
[7] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*. New York: Springer-Verlag, 1997.
[8] W. C. Elmore, "The transient response of damped linear networks with particular regard to wideband amplifiers," *J. Appl. Phys.*, vol. 19, no. 1 , pp. 55–63, 1948.
[9] A. Jagannathan, S.-W. Hur, and J. Lillis, "A fast algorithm for context-aware buffer insertion," in *Proc. ACM/IEEE Design Automation Conf.*, 2000, pp. 368–373.
[10] M. Lai and D. F. Wong, "Maze routing with buffer insertion and wiresizing," in *Proc. ACM/IEEE Design Automation Conf.*, 2000, pp. 374–378.
[11] J. Lillis, C. K. Cheng, and T. Y. Lin, "Simultaneous routing and buffer insertion for high performance interconnect," in *Proc. Great Lakes Symp. VLSI*, 1996, pp. 148–153.
[12] 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 *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 2001, pp. 49–56.
[13] L. P. P. P. van Ginneken, "Buffer placement in distributed RC-tree networks for minimal elmore delay," in *Proc. IEEE Int. Symp. Circuits Syst.*, 1990, pp. 865–868.
[14] H. Zhou, D. F. Wong, I.-M. Liu, and A. Aziz, "Simultaneous routing and buffer insertion with restrictions on buffer locations," in *Proc. ACM/IEEE Design Automation Conf.*, 1999, pp. 96–99.

Global and Local Congestion Optimization in Technology Mapping

Davide Pandini, Lawrence T. Pileggi, and Andrzej J. Strojwas

***Abstract*—In this era of deep submicrometer technologies, interconnects are becoming increasingly important as their effects strongly impact the integrated circuit (*IC*) functionality and performance. Moreover, logic block size is no longer determined exclusively by total cell area and is often limited by wiring area. However, synthesis optimization objectives are focused on minimizing the number and size of library cells. Methodologies that incorporate congestion within the logic synthesis objective function have been proposed in the past. Nevertheless, we will demonstrate that predicting the true congestion prior to layout is not possible, and the effectiveness of any congestion minimization approach can only be evaluated after routing is completed within the fixed die size. In this paper, we propose a practical, complete methodology which first performs congestion-aware technology mapping using a global weighting factor for the technology-dependent synthesis cost function and then applies incremental localized unmapping and remapping on layout congested areas. This complete approach addresses the problem that one global factor is not suited for all layout regions of the design, which might have very different routing demands. Most importantly, through the application of this methodology to industrial examples, we will show that any attempt at a purely top–down single-pass congestion-aware technology mapping is merely wishful thinking.**

***Index Terms*—Congestion estimation, logic synthesis, physical design, placement, routability, routing, technology mapping, wiring congestion.**

I. Introduction

In deep submicrometer (DSM) technologies, interconnects play a crucial role in the overall performance of very large scale integration (VLSI) systems [1]. For technologies of 0.25 μm and below, interconnect capacitance becomes dominant with respect to gate capacitance, thus rapidly increasing the interconnect induced delay (as a percentage of the overall path delay). Therefore, the impact of interconnects on performances has to be carefully evaluated in order to satisfy the design constraints during all phases of the traditional application specified integrated circuit (ASIC) top-down design flow. The interconnect models used in timing-driven layout tools are essentially based on fan-out loading and predefined net configurations. However, a fan-out-based model for delay estimation can be highly inaccurate for modeling the actual interconnect delay prior to layout, since, by not considering the actual topology of the wires, it cannot accurately predict the distributed *RC* effects [3]–[5]. As a consequence, many iterations between logic synthesis and physical design are usually necessary to achieve the timing closure. Unfortunately, this iterative process does not have any guarantee of convergence, and significant changes to the high-level description of the circuit may be necessary, thus, introducing a critical bottleneck for tight time-to-market targets. While the problems of timing convergence due to these inaccuracies are well studied, of equal importance is the impact of wiring on defining the block or chip size, in particular for logic synthesis, which traditionally has attempted to minimize cell area in order to minimize the area of a logic block or *IC*. Traditionally, optimization focused on the total cell area for

Manuscript received June 1, 2002; revised September 20, 2002. This work was supported by the Central Research and Development of STMicroelectronics, Inc. This paper was recommended by Guest Editor C. J. Alpert.

D. Pandini is with the Central Research and Development, STMicroelectronics, Agrate Brianza, 20041, Italy (e-mail: davide.pandini@st.com).

L. T. Pileggi and A. J. Strojwas are with the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213 USA (e-mail: pileggi@ece.cmu.edu; ajs@ece.cmu.edu).

Digital Object Identifier 10.1109/TCAD.2003.809646