Buffer Insertion with Adaptive Blockage Avoidance

[Extended Abstract]

Jiang Hu
IBM Microelectronics
11400 Burnet Road
Austin, TX 78758
jianghu@us.ibm.com

Charles J. Alpert
IBM Austin Research Lab
11400 Burnet Road
Austin, TX 78758
alpert@us.ibm.com

Stephen T. Quay
IBM Microelectronics
11400 Burnet Road
Austin, TX 78758
quayst@us.ibm.com

Gopal Gandham
IBM Microelectronics
1580 Rt. 52
Hopewell Jct., NY 12533
gopalg@us.ibm.com

ABSTRACT

Buffer insertion is a fundamental technology for VLSI interconnect optimization. Several existing buffer insertion algorithms have evolved from van Ginneken's classic algorithm. In this work, we extend van Ginneken's 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 is adaptive to the demand on buffer insertion and is incurred only when it facilitates the maximal slack solution. This approach can be combined with any performance-driven Steiner tree construction. The overall time complexity has linear dependence on the number of blockages and quadratic dependence on the number of potential buffer locations. Experiments on several large nets confirm that high-quality solutions can be obtained through this technique with little CPU cost.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance

1. INTRODUCTION

Buffer insertion is now widely recognized as a key technology for improving VLSI interconnect performance. Cong [5] 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. Buffer insertion improves the timing performance of the interconnect through two ways: (i) it regenerates the signal to increase the driving capability for long wires, and (ii) it can shield the load capacitance of non-critical sinks from the most critical source-sink path. In the buffer insertion literature, van Ginneken's dynamic programming based algorithm[20] has established itself as a classic in the field. Given a fixed Steiner tree topology and the Elmore delay model, this algorithm can compute the optimal slack solution in polynomial time.

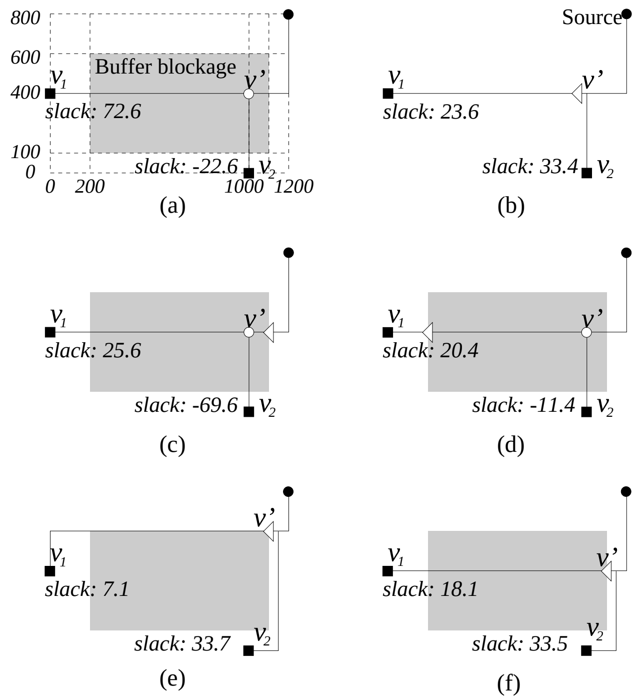

Figure 1: Steiner tree and buffer solutions on a 3-pin net with one buffer blockage.

Prior to buffer insertion, several large area chunks may be already 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, Figure 1(a) shows a Steiner tree with 3-pins and a buffer blockage. Let the required arrival times for the sinks be $rat(v_1) = 200$ and $rat(v_2) = 100$. If the blockage is ignored, one can obtain a good solution as shown in Figure 1(b). Here the buffer acts to decouple the load from the v_1 branch to the more critical sink v_2. Of course, in practice, one cannot ignore the buffer blockage and a solution other than that in Figure 1(b) must be sought. If one restricts the solution space to the existing Steiner topology, the two best solutions are shown in (c) and (d), but neither solution meets the required timing constraints.

Several works have extended van Ginneken's algorithm to handle more general formulations. In [14], Lillis *et al.* proposed several extensions including employing a buffer library, handling both inverting and non-inverting buffers and, trading off power consumption and performance. Alpert and Devgan develop a wire segmenting technique[2] to include buffer insertion points along a path between two nodes. Alpert *et al.* showed how to extend the algorithm to handle higher-order delay models [3]. These extensions all restrict candidate buffer locations to a fixed Steiner tree topology.

Other works have combined buffer insertion and Steiner tree routing into a simultaneous approach. Okamoto and Cong proposed merging A-tree and van Ginneken's algorithm into the BA-tree algorithm[16]. BA-tree is a relatively fast simultaneous approach since the tree topology it generates is limited to be a Steiner arborescence[1]. Lillis *et al.* [15] integrated the P-tree algorithm [13] with buffer insertion since they share a common dynamic programming framework. Salek *et al.* [17] embedded P-tree into a fanout tree optimization algorithm. These methods serve to extend van Ginneken's algorithm by exploring the candidate solutions over the entire Hanan grid[10], which has $O(n^2)$ grid nodes for a net with n pins, rather than a single tree and therefore become prohibitively expensive in terms of run time.

Due to these runtime inefficiencies, Salek *et al.* [18] proposed extending [17] by limiting the number of potential buffer locations. They concluded that reducing the number of buffer locations from $O(n^2)$ to $O(n)$ does not significantly reduce solution quality. Nevertheless, the tree topology itself still is constructed over the Hanan grid simultaneously with generating candidate buffer solutions. Even when restricted to $O(n)$ buffer locations, the complexity of all the works in [15, 17, 18] have at least $O(n^5)$ time complexity.

Zhou *et al.* [21] proposed an optimal algorithm on simultaneous buffer insertion and Steiner tree constructions with buffer blockages for two-pin nets. This algorithm is based on a dynamic programming framework similar to van Ginneken's algorithm, but the entire grid graph is searched instead of a fixed topology. The authors of [11] and [12] also considered blockage avoidance with buffer insertion using graph based approaches for two-pin nets. It is not clear how to extend these methods [21, 11, 12] directly to multi-pin nets.

In [7], Cong and Yuan proposed a dynamic programming algorithm, called RMP, to handle the multi-sink net buffer insertion with location restrictions. RMP is designed for the buffer block methodology[6] 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 [19] on a similar problem. While more efficient than RMP, it can optimize only the maximum sink delay rather than the minimum slack.

The simultaneous routing and buffer insertion approaches[15, 17, 18] can also be modified to handle buffer blockages. One can extend the borders of the blockages over the Hanan grid so that the algorithms are performed on the extended Hanan grid as shown by the dashed grid in Figure 1(a). Then each node in the grid graph is labeled as either blocked or free to disallow inserting buffers over blockages. If there are n sinks and k blockages, a simultaneous approach over this grid implies that candidate solutions will be explored for $O((n+k)^2)$ locations. When n or k is large, such an extensive search will inevitably be slow; a more directional search is preferred to relieve the heavy computation burden.

Difficult buffering problems occur not just with large nets but also when sink polarity constraints are present. Alpert *et al.* developed the C-tree heuristic[1] to handle these types of instances. The buffer insertion solutions that use C-tree topologies have similar timing performance to solutions from simultaneous approaches, though are generated more quickly. However, C-tree does not consider buffer blockages. One could incorporate the method of [4], which first re-routes parts of a Steiner tree to make the wires to avoid the blockages without adding much wiring. Then, this modified tree is passed to the van Ginneken's buffer insertion algorithm. For example, this approach would obtain the buffered solution in Figure 1(e). However, a carefully constructed timing-driven topology can be destroyed by this independent topology change so that the final slack could be significantly worse even though blockages are avoided and buffer insertions are enabled.

Despite all the work in this field, there is still no fast and effective solution for multi-sink nets. We seek an algorithm that can find the solution in Figure 1(f) for our example. This optimal solution for maximizing the minimum slack may be obtained by simply sliding the buffer insertion solution in Figure 1(b) to its closest legal location. While an approach like this works for this example, it fails for a larger-sized 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. Moreover, this technique does not increase the computational complexity of van Ginneken's algorithm. Experimental results show that we can obtain greater efficiency than methods that use the entire Hanan grid.

2. PRELIMINARIES

For the Steiner tree construction, let $V_{internal}$ represent the set of nodes in the tree other than the source and sinks. Our basic problem is given by[2]:

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

[1]A Steiner arborescence is a Steiner tree where every source-sink path has to be a shortest path.

[2]Although not explicitly stated in the formulation, one can trade-off performance with buffering and wiring resources. In our approach, we can achieve this by generating a set of non-dominating solutions using the technique of [14].

The formulation is similar to the formulation for RMP[7] except that a set of legal buffer locations is given in RMP instead of a set of buffer blockages. To make our formulation equivalent to that in [7], we can extend the borders of blockages over a Hanan grid and label each node on the new grid graph as either a feasible or infeasible buffer location. However, this transformation implies that there are up to $O((n+k)^2)$ legal buffer locations while the number of legal buffer locations could be either very small or very large in the RMP formulation. If we remove the restriction of the extended Hanan grid, we may consider any point out of blockage for buffer insertion. For example, in Figure 2(a), the buffer sites in RMP formulation are restricted to a few isolated points while our formulation can exploit a much larger buffer site space as in Figure 2(b) or can be reduced to (a) easily. Thus, our formulation is more general and flexible.

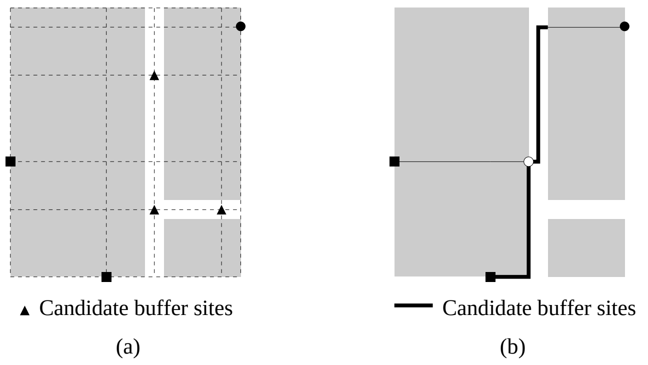

Figure 2: The candidate buffer sites in RMP formulation (a) and the potential candidate buffer sites in our formulation (b).

We adopt the Elmore delay model[9] for interconnect and an RC switch model for gate delays. We assume that the given a 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 dummy edges. For simplicity of discussion, candidate buffer sites are limited to only branch (internal) nodes; it is straightforward to extend the buffer sites to include segmenting points along a path.

Since we are extending van Ginneken's algorithm to directly handle buffer blockages, we first overview it 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 $q(v)$ at node v. We let $s=(c(v),q(v))$ denote a buffering candidate solution at v. For any two candidate solutions $s_1=(c_1(v),q_1(v))$ and $s_2=(c_2(v),q_2(v))$, we say 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 $S(v)=\{s_1,s_2,...\}$ is a non-dominating set if no candidate in this set is dominated by any other candidate in this set. During the bottom-up process of van Ginneken's algorithm, the candidate solutions at leaf node evolve through the following operations:

- $Grow(S(v),w)$: propagate candidate set $S(v)$ from node v to node w to get $S(w)$. If the wire between v and w has resistance R and capacitance C, we can get $c_i(w)=c_i(v)+C$ and $q_i(w)=q_i(v)-R(C/2+c_i(v))$ for each $(c_i(v),q_i(v))\in S(v)$, and obtain $S(w)$ from the solution pairs $(c_i(w),q_i(w))\forall i$.
- $AddBuffer(S(v))$: insert buffer at v and add the new candidate into $S(v)$. If a buffer b has input capacitance c_b, output resistance r_b and intrinsic delay t_b, we can obtain $c_{i,buf}(v)=c_b$ and $q_{i,buf}(v)=q_i(v)-r_bc_i(v)-t_b$ for each $(c_i(v),q_i(v))\in S(v)$ and add the pair $(c_{i,buf}(v),q_{i,buf}(v))\forall i$ with the maximum $q_{i,buf}$ into $S(v)$.
- $Merge(S_l(v),S_r(v))$: 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 $(c_{left}(v),q_{left}(v))$ from the left child and a solution $(c_{right}(v),q_{right}(v))$ from the right child, the merged solution $(c_i(v),q_i(v))$ is obtained through letting $c_i(v)=c_{left}(v)+c_{right}(v)$ and $q_i(v)=\min(q_{left}(v),q_{right}(v))$. The sets are merged such that the number of candidates generated is no greater than $|S_l(v)|+|S_r(v)|$.
- $PruneSolutions(S(v))$: 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 $O(n^2)$ where n is the number of potential buffer insertion locations in the routing tree.

3. THE RIATA ALGORITHM

A common strategy to solve a sophisticated problem is divide-and-conquer, e.g., 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 anticipatedly 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[1].

When considering how to incorporate blockage constraints, we need to partition it into the right phase in the design flow. Blockage avoidance is more closely tied to the request on buffering candidates than to the Steiner tree construction, i.e., it is difficult to know when it is worthwhile to make a Steiner tree avoid blockages without knowing where buffers are required. A full-blown simultaneous approach is not efficient, while the separate routing and buffer insertion approach as in [4] cannot adequately plan for blockages. However, we can find an approach somewhere in between to the middle of these two approaches, by allowing the given Steiner tree to be dynamically adjusted during van Ginneken's algorithm according to requests for buffer blockage avoidance. Unlike the simultaneous approaches[15, 21] that explore the entire grid graph and result in large complexity overhead, we seek a solution that has time complexity no worse than that of the original van Ginneken's algorithm since our goal is to efficiently optimize thousands of nets in a design.

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 [15] which explores a much larger space). These locations correspond to moving a Steiner node outside of a blockage which enables additional 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 a re-routing subpaths to avoid blockages without increasing wirelength before calling van Ginneken's algorithm. For example, Figure 3 shows a three pin net (a) before and (b) after this pre-processing step. Observe that one can always fix a given Steiner topology to avoid as much blockage as possible without changing the tree's delay or wirelength properties (assuming zero delay/resistance vias and the same technology parasitics for both the horizontal and vertical layers). This type of solution can be achieved by applying the work in [4] 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.

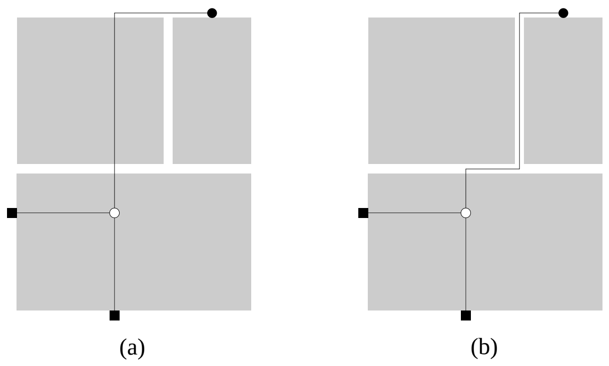

Figure 3: The path between the source and the Steiner node in (a) can be rerouted to avoid buffer blockages as in (b).

The difficult buffer blockage problems occur when a Steiner node lies on top of blockage which eliminates opportunities for decoupling non-critical 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 alternative location outside of blockage for the branching Steiner node.

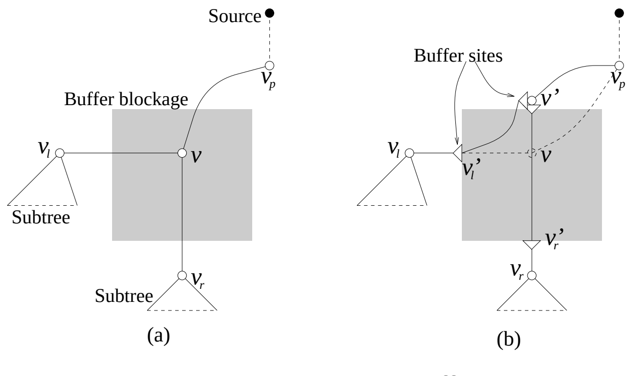

Figure 4: For a Steiner point v within a buffer blockage as in (a), the three buffer locations closest to v can be found as in (b).

Given a Steiner tree, we extend the 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 no longer restricted to a fixed topology. On the other hand, these tree adjustments are based on the request for buffer insertions. During the bottom-up propagation process, if a Steiner point does not overlap a buffer blockage, our algorithm proceeds just like van Ginneken's algorithm. The difference occurs when a Steiner point is located within a buffer blockage, as depicted in Figure 4(a). To compensate for the inability to insert buffers near the blocked Steiner point, we seek an alternative unblocked location nearby to use instead, in effect moving the Steiner point out of the blockage. In our example, Figure 4(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. The exact search area is defined by the bounding box between v and v_p.

```
Procedure: $FindCandidates(v)$
Input:  Current node $v$ to be processed
Output: Candidate solution set $S(v)$ at $v$
Global: Steiner tree $T(V,E)$
        Buffer library $B = \{b_1, b_2, ...\}$
        Rectangles $R = \{r_1, r_2, ...\}$
1. If $v$ is a sink
      $S(v) \leftarrow \{(T_v, c(v), q(v))\}$, Return $S(v)$
2. $v_l \leftarrow$ left child node of $v$
   $S(v_l) \leftarrow FindCandidates(v_l)$
3. $S_l \leftarrow Grow(S(v_l), v)$
4. If $v$ has only one child
      If $v$ is not in $r \in R$, $S_l \leftarrow AddBuffer(S_l)$
      $PruneSolutions(S_l)$, Return $S_l$
5. $v_r \leftarrow$ right child node of $v$
   $S(v_r) \leftarrow FindCandidates(v_r)$
6. $S_r \leftarrow Grow(S(v_r), v)$
7. $S_{unbuf} \leftarrow Merge(S_l, S_r)$
   If $v$ is not in $r \in R$
      $S_l \leftarrow AddBuffer(S_l)$, $S_r \leftarrow AddBuffer(S_r)$
      $S_{buf} \leftarrow Merge(S_l, S_r)$
r1.Else
r2.  $v_p \leftarrow$ parent node of $v$
r3.  If $v' = UnblockedNode(v, v_p, R)$ is found
r4.     $S_{l,buf} \leftarrow AddBuffer(Grow(S(v_l), v'))$
r5.     $S_{r,buf} \leftarrow AddBuffer(Grow(S(v_r), v'))$
r6.     $S_{buf} \leftarrow Merge(S_{l,buf}, S_{r,buf})$
8. $S(v) \leftarrow S_{unbuf} \cup S_{buf}$
9. $PruneSolutions(S(v))$, Return $S(v)$
```

Figure 5: Core algorithm of RIATA.

Other alternative candidate buffer sites v_l' and v_r' can be also be generated by segmenting the path between v and its two child nodes v_l and v_r. The candidate solution set at v_l' is propagated to both v and v', as are the candidates from v_r'. Then, once the candidates for v and v' are generated, the candidates are both grown to their potential parent at v_p and the sets of candidates are reconverged to a single set. We can treat v' as a phantom node of v in seeking of unblocked points. Since the potential locations for v' are limited to a local range, it is possible that no possible location exists. The major reason to limit the search range is to disallow the Steiner point from moving so significantly that it disrupts the given Steiner tree topology. For example, the move of v does not force an adjustment on the location of v_p.

Note that we still consider the original Steiner location and propagate candidates up the tree according to the given topology. This scheme allows us to generate a set of additional candidates corresponding to the possibility that the Steiner point be moved outside of the buffer blockage. Thus, our algorithm is guaranteed to perform at least as well as the original van Ginneken algorithm.

To implement this heuristic we need to efficiently find an alternative location v' for a node v. Given two nodes v and v_p, and a set of rectangles $R = \{r_1, r_2, ...r_k\}$ representing the buffer blockages, if node v is within a blockage $r_i \in R$, we need to find the unblocked point which is the closest to v within the bounding box defined by v and v_p. If there is no overlap between any two buffer blockages, all we need to do is to locate the rectangle r_i that overlaps v; the unblocked point closest to v must lie on the intersection of r_i and the bounding box of v and v_p. If the set of rectangles R is stored as an interval tree[8], the desired rectangle r_i can be found in $O(k)$ time in the worst case. We let $UnblockedNode(v, v_p, R)$ denote the procedure that finds such an unblocked v' if one exists.

We call our heuristic RIATA for Repeater Insertion with Adaptive Tree Adjustment (see Figure 5). The enhancements to van Ginneken's algorithm are shown in boldface; deleting the boldface steps from the figure leaves van Ginneken's original algorithm. Note that this technique is only for the Steiner points that overlap blockages. It is easy to modify any point to point connection among Steiner nodes to overlap the minimum amount of blockage without increasing wirelength *before ever calling van Ginneken's algorithm* (as in Figure 3). Additional runtime resulting from RIATA versus van Ginneken's algorithm comes from searching for an unblocked point between two nodes and additional candidates. Given a net with n pins, a buffer library B and k rectangles representing blockages, then the complexity of RIATA is $O(|B|n^2 + nk)$. Thus, compared to conventional buffer insertion algorithm, only $O(nk)$ additional operations are needed in our algorithm to avoid blockages.

4. EXPERIMENTAL RESULTS

We implemented all code in C++ and performed experiments on a SUN Ultra-10 workstation with 2GB memory. Without loss of generality, we use only one buffer type in our buffer library and sink polarity is not considered. For all experiments, we use C-tree to generate the initial timing-driven Steiner tree, whenever one is required.

4.1 Experiments on large nets

Table 1: Slack comparison between RIATA and van Ginneken's algorithm.

Net	#pins	#blk	Slack(ps)			
			NoBf	VG	RIATA	VGNB
n873	21	6	-867	110	325	542
n189	30	15	-1419	344	442	540
n786	33	15	-848	-479	6	75
n870	44	16	-2835	-98	80	144
big1	64	7	-214	733	879	1044
big2	80	7	-1560	-567	-198	-41
big3	89	21	-798	1070	1329	1575

We obtained seven large nets from industrial designs and generate buffer blockages arbitrarily. The number of pins and buffer blockages for each net are listed in column 2 and 3 in Table 1, respectively. In our experiments, we compare the tree performances of the following approaches together with RMP:

- NoBf: C-tree without any buffer insertion, which gives a baseline for comparison.
- VG: 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.
- RIATA: our algorithm that adaptively adjusts VG to consider alternative buffer insertion locations.
- VGNB: 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 2: Resource comparisons. The number of inserted buffers is denoted as b.

Net	VG		RIATA			VGNB		
	b	CPU	b	wire	CPU	b	wire	CPU
n873	3	0.16	3	5034	0.24	4	4750	0.60
n189	5	0.54	5	5846	0.87	6	5843	10.94
n786	1	0.41	3	5319	0.70	4	5318	1.79
n870	3	0.85	4	4992	1.10	9	4764	2.67
big1	7	0.87	8	9431	1.26	6	9407	4.92
big2	6	0.99	6	12576	1.55	13	12448	20.46
big3	6	2.84	9	20266	6.57	7	19669	63.45

Table 1 presents results from these experiments. We observe the following from the maximum slack results. First, buffer insertion proves to be a worthwhile operation, as all three methods significantly improve on the solution NoBf, which has no buffering. Second, in all cases, RIATA also obtains a significantly better result than VG which shows that considering the additional candidate solutions do make a significant difference. Finally, most RIATA solutions are almost as good as VGNB for which blockages are ignored.

In addition to timing performance, we show major resource utilization (total buffers, wirelength and CPU time in seconds) in Table 2. RIATA uses slightly more buffers than VG, but slightly less than VGNB. Also, RIATA does not significantly increase the wirelength of the low wirelength VG solutions. Not unexpectedly, there is an increase in CPU time of RIATA versus VG. However, VGNB uses quite a bit more CPU time than even RIATA since it has many more potential buffer insertion locations to explore since blockages are ignored. This shows that indeed, RIATA is only examining a "handful" of alternative locations.

4.2 Comparisons with RMP

Our next set of experiments compares RIATA with the RMP algorithm [7], since the two approaches share similar problem formulations. We obtained the executable of RMP from the authors of [7]. 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. 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 3.

Not surprisingly, we observe that RMP typically gives better slack results than that of RIATA though never by more than 72 ps. RIATA actually outperforms RMP by 223 ps for the case n730 and even outperforms VGNB for this instance. We speculate the reason is that by finding an alternative insertion point RIATA actually

Table 3: Comparison between C-tree+RIATA and RMP. The number of pins for each net is #p and the number of legal buffer sites is #s.

Net	#p	#s	Slack(ps)				# buf		wirelength		CPU(sec)	
			NoBf	RIATA	VGNB	RMP	RIATA	RMP	RIATA	RMP	RIATA	RMP
n071	8	19	183	467	574	539	3	5	5868	6422	0.06	10.49
m0s5	8	13	22	357	416	365	3	5	7043	10207	0.03	8.07
n313	9	33	188	469	513	529	3	5	5860	10876	0.09	0.79
n730	9	15	673	961	855	738	3	3	2348	2491	0.04	19.60
pnt3	10	19	687	939	1004	1011	5	6	7250	15093	0.09	11.72
m1s9	10	17	369	747	790	818	3	6	4608	6554	0.05	283.07
n905	11	12	284	783	851	843	3	4	3026	3379	0.09	109.35
n702	11	17	-1095	151	164	202	2	3	3583	4294	0.07	2645.26
n866	12	26	-17	477	575	523	4	7	7730	10587	0.09	1083.04

improves the original performance-driven Steiner tree construction. Since RIATA searches a much smaller space than RMP it uses orders of magnitude less CPU time. Clearly, RMP cannot be applied to thousands of nets in a physical synthesis type of optimization. In addition, RMP actually uses more buffers and significantly more overall wirelength as well.

5. CONCLUSION

We propose 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 fairly simple technique can give significant improvements over van Ginneken's original algorithm with marginal CPU cost. Further, it is much faster and nearly as effective as the RMP approach, which searches a much larger solution space. One of the keys to RIATA is that it does not significantly perturb the existing Steiner topology while generating alternative Steiner points. We believe that other schemes for carefully finding additional potential locations for buffers may be able to further improve performance without significantly impacting runtime. Our future work seeks to identify such techniques.

6. ACKNOWLEDGEMENT

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

7. REFERENCES

[1] C. Alpert, G. Gandham, M. Hrkic, J. Hu, A. Kahng, J. Lillis, B. Liu, S. Quay, S. Sapatnekar, A. Sullivan, and P. Villarrubia. Buffered Steiner trees for difficult instances. In *ISPD*, pages 4–9, 2001.

[2] C. J. Alpert and A. Devgan. Wire segmenting for improved buffer insertion. In *DAC*, pages 588–593, 1997.

[3] C. J. Alpert, A. Devgan, and S. T. Quay. Buffer insertion with accurate gate and interconnect delay computation. In *DAC*, pages 479–484, 1999.

[4] 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 CAD*, 20(4):556–562, Apr. 2001.

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

[6] J. Cong, T. Kong, and D. Z. Pan. Buffer block planning for interconnect-driven floorplanning. In *ICCAD*, pages 358–363, 1999.

[7] J. Cong and X. Yuan. Routing tree construction under fixed buffer locations. In *DAC*, pages 379–384, 2000.

[8] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational geometry: algorithms and applications*. Springer-Verlag, 1997.

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

[10] M. Hanan. On Steiner's problem with rectilinear distance. *SIAM Journal on Applied Mathematics*, 14(2):255–265, 1966.

[11] A. Jagannathan, S.-W. Hur, and J. Lillis. A fast algorithm for context-aware buffer insertion. In *DAC*, pages 368–373, 2000.

[12] M. Lai and D. Wong. Maze routing with buffer insertion and wiresizing. In *DAC*, pages 374–378, 2000.

[13] J. Lillis, C. K. Cheng, T. T. Lin, and C. Y. Ho. New performance driven routing techniques with explicit area/delay tradeoff and simultaneous wire sizing. In *DAC*, pages 395–400, 1996.

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

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

[16] T. Okamoto and J. Cong. Interconnect layout optimization by simultaneous Steiner tree construction and buffer insertion. In *ACM Physical Design Workshop*, pages 1–6, 1996.

[17] A. H. Salek, J. Lou, and M. Pedram. A simultaneous routing tree construction and fanout optimization algorithm. In *ICCAD*, pages 625–630, 1998.

[18] A. H. Salek, J. Lou, and M. Pedram. MERLIN: Semi-order-independent hierarchical buffered routing tree generation using local neighborhood search. In *DAC*, pages 472–478, 1999.

[19] X. Tang, R. Tian, H. Xiang, and D. Wong. A new algorithm for routing tree construction with buffer insertion and wire sizing under obstacle constraints. In *ICCAD*, pages 49–56, 2001.

[20] L. P. P. P. van Ginneken. Buffer placement in distributed RC-tree networks for minimal elmore delay. In *ISCAS*, pages 865–868, 1990.

[21] H. Zhou, D. F. Wong, I.-M. Liu, and A. Aziz. Simultaneous routing and buffer insertion with restrictions on buffer locations. In *DAC*, pages 96–99, 1999.